JS-EDEN ADM Plugin Documentation

Contents

1 Introduction

This documentation describes the design decisions and implementation of the JS-EDEN ADM plugin created for this coursework project. It also describes the example models created and what was discovered creating these.

2 Plugin Design

The aims of the plugin are to bring the ADM up-to-date with the current state of EM tools and explore ideas relating to parameterisation and the human perspective. One of the main tools currently being used to develop EM models is JS-EDEN, a JavaScript implementation of an EDEN interpreter that can be used in a web browser[1]. JS-EDEN allows the development of plugins to add additional functionality to the tool. It was decided to implement a version of the ADM as a JS-EDEN plugin, which encouraged JavaScript to be the main language of implementation.

One motivation in the design of the plugin was to make it accessible and usable to people unfamiliar with the environment. Because of this it was decided to use guided templates to allow the user to create entity templates to be instantiated. This includes input boxes for the template name, definitions and actions, which are of the form:

```
guard --> action sequence;
```

All code entered should be JS-EDEN code, except for special constructs that were created for instantiating and deleting entities. It was decided to use the α character to represent an action on an entity, as this is a special character not likely to be used in normal JS-EDEN code and so distinguishable. This is inspired by the definition factory model[2]. Actions to instantiate entities are of the form:

```
α_instantiate(template_name(parameter_values) as entity_name)
```

or α_i for short. Actions for removing entities are of the form:

```
α_remove(entity_name)
```

or α_r for brevity. A separate view is then given for instantiation of templates with parameters values. Again this contains separate input boxes for each parameter in order to make instantiation clearer.

[1]http://jseden.dcs.warwick.ac.uk/master/

[2]http://www2.warwick.ac.uk/fac/sci/dcs/research/em/teaching/cs405-1213/lab9

Although creating entities in this way is simple and clear it is very time consuming, so a language was also defined for more advanced users. This language is based on languages defined by previous ADM translators and tries to make instantiation clear. Here is an example template creation as defined in this language:

```
{
  name: entity_name;
  definitions: {
    eden_definition;
  }
  actions: {
    eden_guard --> eden_action or entity_action; ...;
  }
}
```

After templates have been created they may be instantiated as entities as following, using a similar format to instantiating entities within actions:

```
template_name(parameter_values) as entity_name;
```

After entities are created and instantiated they need to be displayed to the user so actions can be chosen and computation carried out. For these purposes a separate human perspective needs to be created, within which available actions will be listed below entity names and these selected for each step. For the motivations of this human perspective please see the project report. Multiple actions should be selectable to enable concurrent computation. In the case that the action is part of a sequence of actions and there are more to follow this should be made visible to the user. As the design is just a simple prototype it was decided to do this very basically by showing a '...' following the action.

It is also useful to be able to view details of all entities which are currently instantiated so as to have an understanding of the agents within the system. For these purposes a separate view was created to show the definitions and actions for each entity.

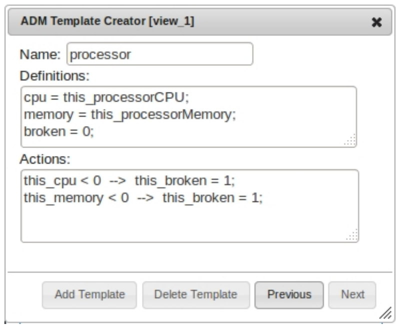
3 Plugin Development

As described above the core functionality of the ADM was primarily implemented using JavaScript, as originally encouraged by the JS-EDEN plugin creation procedure. For each of the windows described in the design section a "view" was implemented, using HTML and CSS for the actual view and JavaScript for the functionality. The development of each of these views will be described in turn below.

Testing was carried out as development progressed and Github used for version control[3]. Functionality was added in an iterative manner, with the template creator originating as an instantiator at the same time. The JavaScript objects required to create the views were adapted from the objects used in the Symbol List plugin, to make development easier and also to give continuity with other plugins within JS-EDEN.

3.1 Template Creator

This view consists of three input boxes, as described in the design section, for entering a name, definitions and actions for the template. There are buttons for adding the template and scrolling through and deleting previously created templates. Any parameters should be named in parenthesis after the template name. When referring to variables defined by itself the "this" keyword can be used, which will be replaced by the entity name on instantiation. The view for this can be screen in figure 1 below.



Name: processor

Definitions:

```
cpu = this_processorCPU;
memory = this_processorMemory;
broken = 0;
```

Actions:

```
this_cpu < 0  -->  this_broken = 1;
this_memory < 0  -->  this_broken = 1;
```

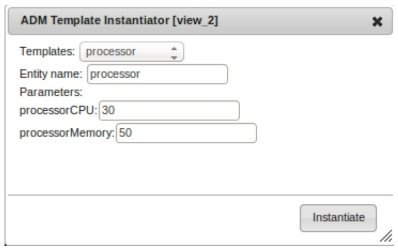
Figure 1: Template creator view.

When a template is created it is added to a JavaScript array containing current templates and added to the template instantiator view if it exists.

3.2 Template Instantiator

The purpose of this view is to allow users to instantiate previously defined templates with specific values for any parameters. Templates can be selected from a drop down list and then instantiated using the text boxes to specify a unique entity name and values for all parameters. This view can be seen in figure 2, where a template for "processor" is being instantiated.

[3]`https://github.com/RuthRainbow/js-eden`



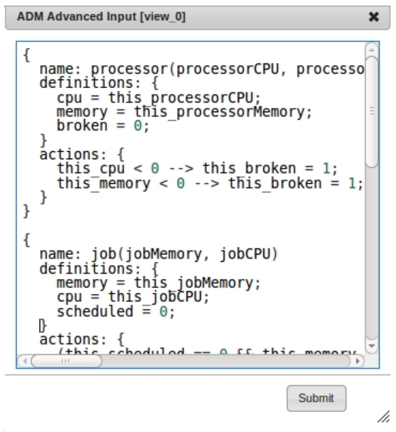
```
Templates: processor
Entity name: processor
Parameters:
processorCPU: 30
processorMemory: 50
```

Figure 2: Template instantiator view.

When a template is instantiated all “this” keywords are replaced by the entity name. The parameter values are then matched up to the parameter names specified in the template and defined as JS-EDEN variables, being prefixed by the entity name specified. Definitions are also processed in a similar way, being prefixed by the entity name and then defined as a JS-EDEN variable. For each guard a JS-EDEN variable is created which depends on the boolean statement given as the guard. Guard action pairs are then added to an array, where the guard refers to the JS-EDEN boolean variable. An array of instantiated entity objects is maintained for reference in JavaScript. The human perspective and entity list views are also updated if they exist.

3.3 Advanced Input

This view is simply a code editor that allows the user to enter ADM style code to create and instantiate templates. The view can be seen in figure 3.



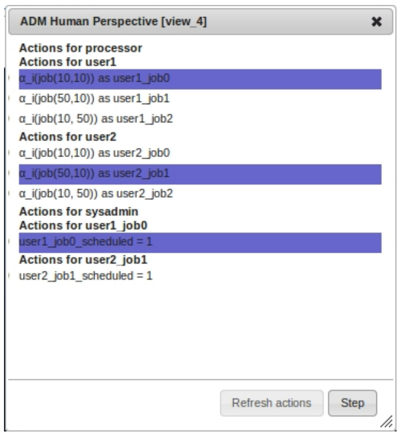
```
{
  name: processor(processorCPU, processo
  definitions: {
    cpu = this_processorCPU;
    memory = this_processorMemory;
    broken = 0;
  }
  actions: {
    this_cpu < 0 --> this_broken = 1;
    this_memory < 0 --> this_broken = 1;
  }
}

{
  name: job(jobMemory, jobCPU)
  definitions: {
    memory = this_jobMemory;
    cpu = this_jobCPU;
    scheduled = 0;
  }
  actions: {
    (this_scheduled == 0 && this_memory
```

Figure 3: Advanced input view.

When the “submit” button is clicked the code is parsed and templates created and entities instantiated in a similar way to the template creator and instantiator views, ensuring that each has a template or entity unique name.

3.4 Human Perspective

The human perspective is perhaps the most important view of the plugin as this is where the human user chooses the actions to execute at each step of computation. This view consists of a list of actions with true guards for each entity in the system. When actions are selected these are highlighted. If the action is not the final action of a sequence a '...' is displayed after the action to indicate this. The view can be seen in figure 4.



```
Actions for processor
Actions for user1
α_i(job(10,10)) as user1_job0
α_i(job(50,10)) as user1_job1
α_i(job(10, 50)) as user1_job2
Actions for user2
α_i(job(10,10)) as user2_job0
α_i(job(50,10)) as user2_job1
α_i(job(10, 50)) as user2_job2
Actions for sysadmin
Actions for user1_job0
user1_job0_scheduled = 1
Actions for user2_job1
user2_job1_scheduled = 1
```

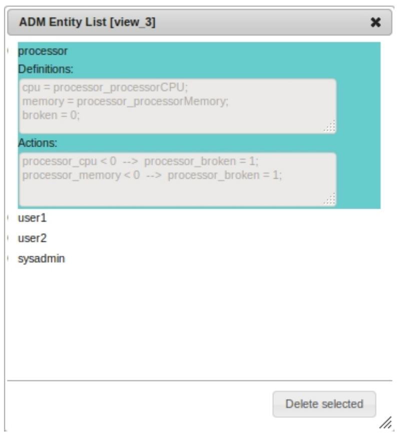
Figure 4: Human perspective view.

The "Refresh actions" button exists to check the state of all guards and update the available actions accordingly in case these have been changed directly through JS-EDEN code input.

When the "step" button is pressed all highlighted actions are executed. This is done by looping through each in turn. The spirit of the ADM suggests that these should be done truly in parallel, however this is not possible as JS-EDEN runs only on a single thread. The actions are also executed deterministically in the order selected, although in theory the order should be nondeterministic. It is assumed that the human user has not created a conflict so no check is carried out. EDEN definitions are simply processed by executing JS-EDEN statements. Entity instantiations are parsed in a similar way to in the advanced input view. Entity deletions are parsed and then the specified entity is removed from the JavaScript array of current entities. Once the actions have redefined the state, the status of guards, which will have been updated via their EDEN dependences, is rechecked and the human perspective view updated according to actions which are now available. If selected actions were part of a sequence the next action in the sequence will become available, allowing the sequence to be interleaved asynchronously. The entity list view is also updated accordingly, if it exists.

3.5 Entity List

This view exists so users can view the definitions and actions of all entities currently instantiated in the system. When views are clicked they are expanded and highlighted, with their definitions and actions being shown as for processor in figure 5.



```
cpu = processor_processorCPU;
memory = processor_processorMemory;
broken = 0;
```

```
processor_cpu < 0  -->  processor_broken = 1;
processor_memory < 0  -->  processor_broken = 1;
```

Figure 5: Entity list view.

This view also provides a button for deleting selected entities. When this is clicked the entity is removed from the JavaScript array of currently instantiated entities and this view updated. The human perspective view is also updated if it exists to remove this entity and any available actions it had previously.

4 Example Models

In this section we discuss the example models included with the prototype. The code for these is in individual directories along with a README file. Further information about each model can be found within the corresponding README files, as well as what is described below.

4.1 Job Model

The simple job model was the first example created for the plugin, and does not include any graphics. Creating an example model helped the development effort in that it uncovered many bugs within the system.

This model creates a system where user entities can submit jobs, with specified CPU and memory requirements. These jobs are instantiated as new entities. Jobs can be scheduled from the human perspective if there is enough CPU and memory. However if too many jobs are scheduled simultaneously the processor may break. If the processor breaks the sysadmin entity is able to remove the existing processor entity and instantiate a new entity which isn't broken.

This very simple example illustrates a parallel system which can be controlled using the human perspective. It illustrates how the parameterisation is useful - processor entities can be created with differing amounts of CPU and memory. It also illustrates how the available actions depend on the state of the system, as jobs cannot be scheduled if the processor has broken due to overload. It illustrates how actions instantiating and removing entities are useful, as can be seen in the role of the sysadmin entity.

However the example also illustrates some limitations of the system. Only one item in a sequence can be executed in each atomic step, but the original state allowing the action to occur may have changed. Also all actions need to be selected by the human user to occur, even though they may be inherent. An example of this is the processor breaking - the user can choose to go over CPU and memory and not cause the processor to break and continue choosing actions for jobs to execute.

4.2 CatFlap Model

This model was based on Bridge's LSD specification[4], with my own modifications made through empirical experimentation. A very simple animation is also provided to give the user a greater understanding of the position of the cat within the model

The human perspective allows the cat to act non-deterministically as mentioned in the original LSD specification, according to how the modeller or "superuser" chooses for the cat to act. For example the cat may intend to go inside, move halfway under the flap and then change its intentions. With the human perspective situations such as this can be created and the consequences of this on other entities in the model observed.

[4]`http://www2.warwick.ac.uk/fac/sci/dcs/research/em/teaching/cs405-0708/concsys/concsyslecture4/lsdexamples/catflapiblsd`

4.2.1 Changes from Bridge's LSD Specification

In this example I made a few changes from Bridge's LSD specification based on my own experimentation and issues with the implementation. These changes and the motivation behind them are discussed below.

Firstly JS-EDEN does not implement enums to my knowledge, so integer values were used instead to indicate the various states of the cat flap and the cats intention etc. In general -1 refers to outside, 0 refers to indifferent/inbetween, and 1 refers to inside.

JS-EDEN also does not implement power sets to my knowledge. To overcome this instead the set of set lock positions is changed to two definitions - one for the outside lock, one for the inside lock. These are part of the flap and so belong to that entity. The actions for "man" are also updated correspondingly, to be able to flip the status of the lock as long as the flap is currently in the vertical position (0).

The specification for the cat suggests that it should only be able to stop pushing if it is not currently engaged, i.e. next to the cat flap. This was changed as the cat should be able to stop pushing at any time.

Movement conditions on the cat were also changed so it could move in either direction if away from the flap, regardless of its intentions, indicating that it is free to move in either the inside or the outside world. This was also required to allow the cat to move towards the flap from its starting position.

The actions for the flap were also changed so the action to change flap position was only available if the flap was not currently in that position.

The guard for the cat to begin pushing in a direction was changed so this can only happen if it is currently engaged. This was probably needed because of the addition of the cat being allowed to move fully away from the flap.

An extra guard was also added to the action which moves the flap back to the vertical position so that it can't close whilst the cat is underneath it. For this an extra definition was also added to the cat to indicate when it is underneath. This was probably needed as the original specification did not allow the cat to be in a position that was directly beneath the flap, and instead the cat moved atomically through in one step.

4.3 Systolic Array

For this example there was no previous LSD specification to adapt from, and the model was mainly constructed using the description in "Parallel Computation in Definitive Models" along with the advice of Dr. Meurig Beynon. This model illustrates that the current prototype is capable of "classical" ADM execution where all actions with true guards are executed at each time step, in a machine oriented fashion. It is assumed that the programmer has not included actions that may conflict, as in a standard programming environment where the existence of a conflict may cause undefined behaviour. Currently the prototype is able to model an autonomous system if all true guards are selected at each computational step.

This model requires two templates: one to model "time" through the system as the number of completed processor steps and the other as a single processor in the array. This illustrates the usefulness of having parameterised templates - each processor can be instantiated with α and β which indicate the index of the processor in the 2D array.

The entity representing time is very simple - it just consists of a time value, t, which should be incremented after each processor step. It was decided not to include the time_changed definition as suggested in the paper as with the human perspective this is not necessary.

The entities representing processors consist of three definitions (i, j, k) as well as the parameters α and β. These three definitions are defined as described in the paper have a dependency on the current value of time. At each time step, if the time is the correct value modulo 3 for this processor to function, a step of computation is carried out on the output matrix. This computation affects a single element in the output matrix using the current values of i, j and k as indexes for which elements to use from the input matrices.

Once a matrix multiplication has been carried out the model can be easily reset using the Eden input window, by setting time_t = 0; and resetting the matrix C to all 0s. A and B can also be redefined if a different multiplication is desired.

Although this model is not fully correct it succeeds in illustrating how the prototype ADM can easily be adapted to model autonomous systems. It also illustrates how the human perspective is useful in understanding concurrent systems, as the effect of each selected action can be observed manually. It also allows a greater level of interaction and experimentation, for example with the human perspective it is possible to create situations where a number of processors fail to function.

5 Creating ADM Code

This section describes some methods for creating models in the ADM. Conversion from pre-existing ADM code or LSD notation will be briefly described. To create a new model in the ADM it may be useful to write an LSD account first of the model you wish to create. This will help you split the "agents" in the model into well defined entities with state and a protocol for transitions over this. Once you have created a preliminary model in the ADM you are able to refine it empirically through experimentation. This is aided by the human perspective which clearly shows you the state changes caused by your actions.

5.1 Converting from ADM

Code should be quite portable from old ADM models, with slight changes of syntax. Firstly there are some differences in the placement of braces and colons. Secondly when an entity is referring to the definitions of another entity it needs to be prefixed with "entityname_", or "this_" if referring to its own definitions.

One main issue will be converting the actual eden and associated notations into JS-EDEN, in particular any line drawings made during Donald. An automatic translator from Donald to JS-EDEN code would be very useful here.

5.2 Converting from LSD notation

The complexity of LSD conversion depends on the complexity of the LSD account itself. If the account refers to function that are not available in JS-EDEN or refers to all of one particular type of entity it may need adaptation. Only simple principles for conversion are described here.

Oracles and handles are not used by the ADM, as entities have access to all definitions in the definition store. The state should be directly converted into definitions for the relevant template. The derivative should also be added as definitions, each depending on definitions already belonging to the template or to other predefined templates. The protocol should correspond to the actions of the template, which should be directly translatable as this should already include guards.