CASTLE Documentation

Release 1.0

Lachlan Birdsey

Oct 08, 2017

Contents

Documentation for the CASTLE framework.

Github link for CASTLE: https://github.com/CASTLE-FWK

Main CASTLE issue tracker: https://github.com/CASTLE-FWK/CASTLE/issues

Current known major issues

(I apologise for the lack of decent code highlighting, this will be fixed at some point in the future.)

CHAPTER 1

Introduction to CAS, CASL, & CASTLE

CAS, CASL & CASTLE

Introduction

With the advent of highly connected and complicated systems permeating today's world, such as the internet, social networks, and smart cities, studying these systems to analyze or discover properties can provide great benefits. To study such systems, *models* are constructed. These models represent a certain facet of the system under-study and may not always include key features. There are several approaches to creating models such as using discrete event system specifications (DEVS), agent-based modeling (ABM), and complex networks (CN). These models are then *simulated* which allows a user to study their outputs, behaviors, and properties.

Many of the systems studied can be considered 'complex systems'; systems that contain a large number of complex interacting entities. Complex systems exist in many facets of reality, and have been studied exstensively. However, complex systems ignore properties that are in these systems such as self-organization, adaptation, emergence, and feedback loops. To study these properties, the term 'complex adaptive systems' is used.

Study of 'complex adaptive systems' has been around for several years, in particular focused on by John Holland. However, existing modeling and simulation approaches are used to study them, which can result in incomplete, or incorrect models that lack key properties, hard to reuse models, and high domain specificity.

The Complex Adaptive Systems Language (CASL) and framework, CASTLE, were designed with several benefits in mind. Firstly, a modeling language that can be used by a large range of researchers, with widly different coding abilites. Secondly, a language that considers properties such as self-organization and emergence to be key. Thirdly, a unified framework that allows domain experts to contribute their analysis tools directly. Finally, a simulation back-end that can execute large scale simulations for extended periods of time on commodity hardware.

CASL and CASTLE are developed by Lachlan Birdsey for his PhD project, with the assistance of his supervisors Dr. Claudia Szabo, and Prof. Katrina Falkner.

Yet Another Modeling Language

There are quite a few modeling languages around for complex system modeling, some of which have been used for complex adaptive system modeling. This section consists of a brief description of some popular languages and how they compare with CASL. In addition, simple models will be presented in each language to compare how they look with their companion CASL model.

CHAPTER 2

Getting Started with CASTLE & CASL

Getting Started

To get started using CASL and CASTLE there are a few things that need to be done:

Dependencies

- Repast Simphony
- MongoDB (optional)
- XText (this can be resolved automatically by Eclipse)

Installation

1. Download and install Repast Simphony
2. Add http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/ to `Available Software Sites` in Repast
3. Update Repast (Eclipse Neon.3 is the minimum version required)
4. Clone the CASTLE distribution repository to somewhere useful
5. In Repast:

```
Help -> Install New Software -> Add... ->
Archive... "Point to CASL.zip in the cloned repository"
```

Install, and wait for the remaining dependencies to be resolved.

Usage

As CASL is in constant development, please make sure you have the current version of the plugin and `CASTLE` library. Simply `git pull` the cloned repo. Once you load Repast Simphony, selection Check For Updates from the Help menu, and it will look for the updated CASL plugin to update.

Using CASL to creating simulation code for use in Repast Simphony requires 2 stages, firstly, creating the model and generating the code. Secondly, moving the code into a Repast Simphony project:

1. Create your CASL project. Currently, this is just an Eclipse general project.
2. Create your `.casl` file. There are some templates located here.
3. Clone the `commons.casl` file located here into the project. This is also currently required for every CASL project.

When you create a CASL model file and save, code generated for use in Repast will be located in a directory called `src-ge`

1. Create a Repast Simphony project with the same name as your CASL project.
2. Copy and paste the directories and files located in `src-gen` into the Repast project. Copy and paste the file `MODEL_NAME.rs` file from `src-gen` to the Repast Simphony project's `MODEL_NAME.rs` directory. Allow for replacement.
3. Add `CASTLE.jar` to the project's build path.
4. Before running, select `MODEL_NAME Model` from the Run Configuration menu and add the `CASTLE.jar` to the build path of the project.
5. To run, select `MODEL_NAME Model` from the Run menu. This will load the Repast Simphony simulator. Right-click `Data Loaders` and select `Set Data Loader`. Select `Custom ContextBuilder Implementation` and this should automatically select the main CASL generated System class. Click next and then finish.
6. To inspect and/or change the simulation parameters, you can click the parameters tab in Repast Simphony.
7. When you are ready to execute the simulation, press the Play button near the top of the window.

For more information, follow the guides on how to setup a Repast simulation located at the main Repast site

CHAPTER 3

Complex Adaptive Systems Language

CASL

Introduction

Complex Adaptive Systems Language (CASL) is a declarative agent-based modeling language for create large scale and complicated complex adaptive systems models.

A note: the term CASL by itself refers to both CASL and CASL-SG. CASL-SG only refers to the additional features of CASL-SG. In the case of a CASL (and not CASL-SG) description, it will be stated as such.

Overview

For CASL, each model consists of a single `SYSTEM` block, at least one `AGENT` block, and at least one `ENVIRONMENT` block. The most basic CASL file looks like:

```
import cas.test.commons.*;
SYSTEM: {
        name: "CASL EXAMPLE";
        description: "";
        ruleset: {
                type: lenient;
                inspection_level: none;
                lenient_exceptions: diversity modularity adaptation; //This is␣
↪current inactive
                semantic_groups: disable;
        };
        parameters: {
                var * int:terminationStep = 1000;
        };
        functions: {};
        agent_types: {
                anAgent;
        };
        group_types: {
                ;
        };
        environment_types: {
                anEnvironment;
        };
        end_conditions: {
                condition STEPS terminationStep;
        };
};

AGENT anAgent: {
        description: "";
        parameters: {};
        functions: {
                def initialize()(): {};
        };
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
};

ENVIRONMENT anEnvironment: {
        description: "";
        environment_rules: {
                type: implicit
                attributes: virtual
                layout_type: BOUND;
        };
        parameters: {
                var LayoutParameters:layoutParameters;
        };
        functions: {
                def initialize()(): {};
        };
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
        group_interactions: {}; //This will soon not be necessary
};
```

CASL Basics

The Language

CASL supports common declarative language constructs such as `if/else` statments, loops, recursion, etc.

Naming Rules & Conventions

Naming rules in CASL are the same as in Java. These are:

- All variable and entity names must start with an alphabetic character
-

Reserved Keywords

There are several reserved keywords in CASL, they are as follows:

```
if
else
then
var
agt
grp
env
for
foreach
endif
endfor
```

Declaration and Assignment

Variable declarations must be prefaced with `var`, for example:

```
var int:anInteger = 39;
var Vector2:aVector;
var List<Item>:itemList;
```

However, if you wish to store references to entities you must use `agt`, `grp`, or `env` for Agents, Groups, and Environments respectively. For example:

```
agt Cell:aCell;
grp Room:myRoom;
env Forest:theForest;
```

Types

A variable in CASL consists of either primitive or non-primitive types. The primitive types are:

```
bool
int
string
float
null
```

Non-primitive types in CASL are defined in `.casl` library files. By default, CASL comes with:

```
Vector2<descriptor: type>
List<descriptor: type>
Queue<descriptor: type>
PQueue<descriptor: type>
Neighbors<descriptor: type>
LayoutParameters<descriptor: type>
Grid<descriptor: type>
```

IF/ELSE

`IF` statments in CASL are similar to C and Java based languages. An example is:

```
if (condition) then
        //do something
else if (another condition) then
        //do something else
else
```

```
        //do something else
endif;
```

Loops

CASL supports `for` and `foreach` loops. These mostly follow Java conventions, with a couple of exceptions. For example:

```
//For-loop incrementer must be of form x = x + k (until issue is resolved)
for (var int:i = 0; i < 10; i = i +1) do
        //Do something
endfor

foreach (Item item : ItemList) do
        //Do something
endfor
```

Self Reference

To reference a parameter in the same entity, the `self` keyword must be used. For example:

```
var int:newNum = self.oldNum * 2;
```

`self` behaves similarly to the `this` keyword in Java. However, when referencing parameters from the same entity, the `self` keyword must be used. The CASL editor will warn you if this is not done.

Component Reference

These are:

```
FUNCTION.functionName();
BEHAVIOR.behaviorName();
INTERACTION.interactionName();
AGT_INTERACTION.interactionName();
ENV_INTERACTION.interactionName();
GRP_TRANSMISSION.transmissionName();
ADAPTATION.adaptationName();
```

Model Structure

The follwing section describes the functionality of each CASL component. A visual representation of the hierarchical structure of CASL is below **MAKE IMAGE**

SYSTEM

The `SYSTEM` component of a CASL model allows the user to define basics, initialization criteria, system wide functions, entity types and termination conditions.

Name

This is the name of the CASL model and the name the generator gives to the Repast Simphony project as well as the file that controls the simulation. Any character is accepted. Blank names or names only with whitespace are not allowed.

Description

(Optional)

This allows the designer to add a helpful description to the model. Newlines are allowed.

Ruleset

- `type`:
- `inspection_level`:
- `lenient_exceptions`: .. diversity modularity;
- **`semantic_groups`:** This has controls if the model is a CASL or CASL-SG model. It takes two values, either `enable` or `disable`. To learn more about CASL-SG...

Parameters

This stores the `SYSTEM` parameters. Some examples:

```
var int:numberOfCells = 800;
var string:UserConfigurationPath = "some/path/to/a/file";
```

To create parameters that can be altered for initialization, you can include a `*`, for example:

```
var * int:numberOfCells = 400;
var * string:userNamePrefix;
```

This currently only works with primitve types. If you assign a value to a variable, the value will become the initialization variables default.

Functions

This stores the `SYSTEM` functions. An `initialize` function is required, otherwise the `SYSTEM` will not be able to start. For example:

```
def initialize(var int:numAgents)(): {
        self.
};
```

Some examples:

```
//Double a number and return
def doubleNumber(var int:num)(var int:newNum): {
        newNum = num * 2;
};

//Set the position of this SYSTEM
def setPosition(var Vector2:pos)(): {
        self.position = pos;
};

//Get the position of this SYSTEM
def getPosition()(var Vector2:pos): {
        pos = self.position.
};
```

More about Functions

Agent_Types

This is a list of all the agent types that will be in the model. For example:

```
Pigeon,
Eagle,
Dove;
```

Environment_Types

This is a list of all the environment types that will be in the model. For example:

```
Road,
Hospital,
University,
Business;
```

End_Conditions

This stores a list of termination conditions that once met will cause the simulation to end. Typically, this will be a step number. Example:

```
condition STEPS terminationStep;
```

This `end_condition` variable must be declared in the `Parameters` section above.

AGENT

An Agent in CASL and CASL-SG has the following structure:

```
AGENT theAgentsName: {
        description: "";
        parameters: {};
        functions: {
                def initialize()(): {};
        };
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
};
```

An `AGENT` block starts of with `AGENT` and the agents name

Description

(Optional)

This allows the designer to add a helpful description to the model. Newlines are allowed.

Parameters

This stores the `AGENT` parameters. Some examples:

```
var bool:Alive = false;
agt Cell:aNeighbour;
```

More about Parameters

Functions

This stores the `AGENT` functions. Each `AGENT` requires an `initialize` function to be declared and the CASL Editor will warn you if it is missing. Some examples:

```
//Double a number and return
def doubleNumber(var int:num)(var int:newNum): {
        newNum = num * 2;
};

//Set the position of this AGENT
def setPosition(var Vector2:pos)(): {
        self.position = pos;
};

//Get the position of this AGENT
def getPosition()(var Vector2:pos): {
        pos = self.position.
};
```

More about Functions

Behaviors

This stores the `AGENT` behaviors. An example:

```
changeStateToDead[SELF][DELAYED](): {
        FUNCTION.setState(false);
};
```

More about Behaviors

Interactions

This stores the `AGENT` interactions. An example:

```
checkNeighboursVelocity[AGENT][INSTANT](): {
        BEHAVIOR.adjustVelocity(neighbour.AGT_INTERACTION.getVelocity());
};
```

More about Interactions

Adaptation

This stores the `AGENT` adptations or adaptive processes. An example:

```
adaptState[IMPLICIT][NONE](var int:numNeighbors): {
        if (numNeighbors > 3) then
                BEHAVIOR.die();
        endif;
}
```

More about Adaptations

Subsystems

This stores the AGENT subsystems. In here you can declare multiple AGENT types. The parent type and other subsystems can interact.

More about Subsystems

ENVIRONMENT

An ENVIRONMENT in CASL has the following structure:

```
ENVIRONMENT anEnvironment: {
        description: "";
        environment_rules: {
                type: implicit
                attributes: virtual
                layout_type: BOUND;
        };
        parameters: {
                var LayoutParameters:layoutParameters;
        };
        functions: {
                def initialize()(): {};
        };
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
};
```

In CASL-SG, an ENVIRONMENT has an extra component called Group_Interactions:

```
group_interactions: {};
```

Description

(Optional)

This allows the designer to add a helpful description to the model. Newlines are allowed.

Environment_Rules

There are 3 rules that have to be set for each ENVIRONMENT, these are type, attributes, layout_type.

Type

Attributes

Layout_Type

Parameters

This stores the ENVIRONMENT parameters.

More about Parameters

An ENVIRONMENT must have the layoutParameter variable. In addition, a layout representation variable must be declared that matches the layout_type. For example, if the layout_type is set to GRID:

```
var Grid<Cell>:cellGrid;
```

Functions

This stores the ENVIRONMENT functions. Each ENVIRONMENT requires an initialize function to be declared and the CASL Editor will warn you if it is missing. Some examples:

```
//Double a number and return
def doubleNumber(var int:num)(var int:newNum): {
        newNum = num * 2;
};

//Set the position of this ENVIRONMENT
def setPosition(var Vector2:pos)(): {
        self.position = pos;
};

//Get the position of this ENVIRONMENT
def getPosition()(var Vector2:pos): {
        pos = self.position.
};
```

More about Functions

Behaviors

This stores the ENVIRONMENT behaviors. An example:

```
changeStateToDead[SELF][DELAYED](): {
        FUNCTION.setState(false);
};
```

More about Behaviors

Interactions

This stores the ENVIRONMENT interactions. An example:

```
checkNeighboursVelocity[ENVIRONMENT][INSTANT](): {
        BEHAVIOR.adjustVelocity(neighbour.AGT_INTERACTION.getVelocity());
};
```

More about Interactions

Adaptation

This stores the ENVIRONMENT adptations or adaptive processes. An example:

```
adaptState[IMPLICIT][NONE](var int:numNeighbors): {
        if (numNeighbors > 3) then
                BEHAVIOR.die();
        endif;
}
```

More about Adaptations

Subsystems

This stores the ENVIRONMENT subsystems. In here you can declare multiple ENVIRONMENT types. The parent type and other subsystems can interact.

More about Subsystems

SEMANTIC GROUP

A semantic group in CASL-SG has the following structure:

```
GROUP aGroup: {
                description: "";
                group_rules: {
                        layout_type: BOUND;
                };
                parameters: {
                        var LayoutParameters:layoutParameters;
                };
                functions: {
                        def initialize()(): {};
                };
                behaviors: {};
                external_interactions: {};
                internal_interactions: {};
};
```

Parameters

This stores the GROUP parameters.

More about Parameters

An GROUP must have the layoutParameter variable. In addition, a layout representation variable must be declared that matches the layout_type. For example, if the layout_type is set to GRID:

```
var Grid<Cell>:cellGrid;
```

Functions

This stores the GROUP functions. Each GROUP requires an initialize function to be declared and the CASL Editor will warn you if it is missing.

More about Functions

Behaviors

This stores the GROUP behaviors.

More about Behaviors

Interactions

This stores the GROUP interactions. An example:

```
checkNeighboursVelocity[GROUP][INSTANT](): {
        BEHAVIOR.adjustVelocity(neighbour.AGT_INTERACTION.getVelocity());
};
```

More about Interactions

External_Interactions

Internal_Interactions

CASL Components

Parameters

Structure

A parameter in CASL consists of a descriptor, type, name, and optional assignment. Generic parameter structure:

```
descriptor type:name = assignment;
```

A descriptor can be either `var`, `agt`, `grp`, or `env`, which refers to a variable, `AGENT`, `GROUP`, or `ENVIRONMENT` respectively.

Types

Types can either be primitives or non-primitives. Example:

```
var int:aNumber = 19; //A primitive
var Vector2:aVector; //A non-primitive
var Grid<agt.Cell>; //A paramaterized non-primitive
```

Functions

Structure

A function in CASL consists of `def`, a function name, at least zero input parameters, and an optional single output parameter. An example is:

```
def function(var bool:in)(var int:out): {
        if (bool) then
                out = 10;
        else
                out = 3;
        endif;
};
```

Return types are implicit in functions and only occur if a output variable is declared. If no output variable is declared, the function will not return anything.

Input and output parameters

Examples

Behaviors

The primary driver of all entities in CASL.

Structure

A behavior in CASL consists of a name, contact type, trigger type, and input parameters. Two simple examples:

```
//Generic
behaviorName[SELF][INSTANT](input parameters): {
        //Do something
};

//Occur each step
behave[SELF][REPEAT (1)](): {
        //Do something
};
```

Contact Types

There are 2 contact types, namely, `SELF`, and `AFFECT`

- `SELF`:
- `AFFECT`:

If you select the wrong type, the CASL Editor will warn you and prevent your model code from generating.

Trigger Types

There are 4 trigger types, namely, `INSTANT`, `DELAYED`, `STEP`, `REPEAT`.

- **`INSTANT`:** This triggers the behavior as soon as it is called.
- **`DELAYED`:** This delays the triggering of the behavior until the CLEANUP phase of the same step.
- **`STEP (x)`:** This triggers the behavior after `x` steps, where `x` is greater than 0.
- **`REPEAT (x)`:** This causes the behavior to occur every `x` steps, where `x` is greater than 0.

Input Parameters

Any input parameter is allowed.

Examples

```
//Every step, interact with neighbors
doStep[AFFECT][REPEAT(1)](): {
        INTERACTION.checkNeighbors();
}
```

Interactions

An Interaction allows for entities to communicate with one another and therefore fulfilling the basic idea of agent based modelling.

Structure

A CASL interaction consists of a name, an interaction type, a trigger type, and an optional input parameter. An example is:

```
interact[QUERY][STATE](var int:num): {
        //The interaction
};
```

Interaction Type

There are 3 types of interactions, namely, QUERY, COMMUNICATION, and INDIRECT.

- QUERY
- COMMUNICATION
- INDIRECT

Trigger Type

There are 4 types of trigger types: STEP, STATE, PARAMETER, INPUT.

- STEP
- STATE
- PARAMETER
- INPUT

Input Parameters

Any input parameters are allowed.

Examples

Reference interactions differs depending on the context. The following examples show the three 3 main contexts.:

```
//An entity triggering its own interaction
//This is usually how a COMMUNICATION interaction is performed
INTERACTION.interactWithFriend(agt Friend:f): {

};

//An entity triggering the interaction in another entity
//This is usually how a QUERY interaction is performed
agt Friend:aFriend; //A reference to an agent
var int:num = aFriend.AGT_INTERACTION.getInformation();
```

Adaptations

Subsystems

Subsystems allow for one or more AGENT entities to exist and operate inside another AGENT. This is also extended to the ENVIRONMENT entity. Subsystems are simply declared as an AGENT or ENVIRONMENT depending on the parent type. An example for a subagent is as follows:

```
//Other parent agent sections
subsystems: {
        AGENT subAgent: {
                description: "";
                parameters: {};
                functions: {};
                behaviors: {};
                interactions: {};
                adaptation: {};
                subsystems: {};
        };
};
```

These subentities can interact with other subentities in the same parent entity, and the parent entity can interact with their own subentities. Subentities are not allowed to interact with subentities from other entities, unless through an interaction of the parent entity.

Examples

CASL System Calls

Occasionally, and predominantly at initialization time, other entities may need to access `PARAMETERS` or `FUNCTIONS` from the `SYSTEM`. This can be acheieved by using a *system call*, which takes the form of:

```
SYSTEM.parameterName;
```

For example, for a variable `initialPopulationSize` stored in an `ENVIRONMENT`:

```
var int:initialPopulationSize = SYSTEM.initialPopulationSize;
```

Macros

CASL contains several useful macros for essential simulation function. The most important ones are `POPULATE` and

POPULATE

The `POPULATE` macro allows SYSTEMS, ENVIRONMENTS, and GROUPS to create populations of entities. It takes the form of:

```
CASL.POPULATE[theLayoutVariable](layoutInitializationParameters)[aRange,
↪theTypeOfEntityToBePopulated](entityInitializationParameters);
```

An example of populating `Cells` into a `CellCity` where the layout variable is called `cellGrid`

```
CASL.POPULATE[cellGrid](SYSTEM.sizeOfGrid)[SYSTEM.numberOfCells, AGENT.Cell](self);
```

For each Entity type, you are required to execute one `POPULATE` per type.

LOGGER

In the `initialize` function for the `SYSTEM`, you can define how logging will work across the model. The `LOGGER` macro takes the form of:

```
CASL.LOGGER(mute, toConsole, toFile, ["filePath"], infoToFile, infoToConsole,␣
↪infoToDB);
```

An example of configuring the CASL Logger to only send to the console:

```
CASL.LOGGER(false, true, false, false, false, false);
```

An example of configuring the CASL Logger to send to both the console and a file:

```
CASL.LOGGER(false, true, true "/users/aUser/simulations/output.txt", false, false,␣
↪false);
```

An example of configuring the CASL logger and execution data writer for data farming (that is send all execution information to the data base while maintaining console printing of explcit logs):

```
CASL.LOGGER(false, true, false, "", false, false, true);
```

To send data to the Logger, in any function or feature simply:

```
CASL.LOG("information to log");
CASL.LOG("the state: "+FUNCTION.getState());
```

METRIC

The `METRIC` macro turns on tracking of that particular feature. To use this, simply:

```
CASL.METRIC[true]
```

You can also send further information to be tracked:

```
CASL.METRIC[true](agentState, "aString");
```

GET_ID

Every Entity in CASL has an object called `entityID()` which is generated on creation and assigns each Entity a unique ID. While it's not always useful to have direct access to the ID, situations may occur when you need it. To access an Entity's ID, simply:

```
var EntityID:id = CASL.GET_ID();
```

COUNT

The `COUNT` macro simply counts the number of elements in a `List` that possess a particular value. For example:

```
//Count the number of neighbors that are alive
CASL.COUNT[neighborsList](FUNCTION.getState());
```

CASL-SG

The large scale extension to CASL introduces a new entity type called `GROUP`. This new entity allows several new approaches to constructing models, as well as allowing for much more optimized simulations.

Overview

A generic CASL-SG model example is below:

```
SYSTEM: {
        name: "CASL-SG EXAMPLE";
        description: "";
        ruleset: {
                type: lenient;
                inspection_level: none;
                lenient_exceptions: diversity modularity;
                semantic_groups: enable;
        };
        parameters: {};
        functions: {};
        agent_types: {
                anAgent;
        };
        group_types: {
                aGroup;
        };
        environment_types: {
                anEnvironment;
        };
        end_conditions: {
                condition STEPS terminationStep;
        };
};

AGENT anAgent: {
        description: "";
        parameters: {};
        functions: {};
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
};


GROUP aGroup: {
        description: "";
        group_rules: {
                layout_type: BOUND;
        };
        parameters: {};
        functions: {};
        behaviors: {};
        transmissions: {};
        internals: {};
};

ENVIRONMENT anEnvironment: {
        description: "";
        environment_rules: {
                type: implicit
                attributes: virtual
                layout_type: BOUND;
        };
        parameters: {};
        functions: {};
        behaviors: {};
        interactions: {};
        adaptation: {};
        subsystems: {};
        group_interactions: {};
};
```

Aside from the extra entity, there are few changes. The `ENVIRONMENT` gains a new component called `group_interactions`.

CHAPTER 4

Observation Tool

Observation Tool

The Observation Tool takes data from the simulation and allows the user to run a variety of metrics and analysis tools on the data. The Observation Tool allows metrics to run on data that is either being generated from a currently executing simulation, i.e. *live*, or from a previously executed and stored simulation i.e. *post-mortem*.

Metrics

CASTLE allows users to implement their own metrics to run on data from a simulation.

CHAPTER 5

Contact

If you find any issues, please place them in the Documentation Github issue tracker: https://github.com/CASTLE-FWK/Documentation/issues

Of course, there are many issues overall and n+1 things to be done. Please use the main CASTLE issue tracker: https://github.com/CASTLE-FWK/CASTLE/issues

CHAPTER 6

Publications

Publications

This is a manually curated list of papers, as such it will be updated rather infrequently.

- L. Birdsey, C. Szabo, K. Falkner: CASL: A Declarative Domain Specific Language For Modeling Complex Adaptive Systems, Winter Simulation Conference 2016
- L. Birdsey, C. Szabo, K. Falkner: Large-Scale Complex Adaptive Systems using Multi-Agent Modeling and Simulation, AAMAS 2017

CHAPTER 7

Indices and tables

- genindex
- modindex
- search