The magmaOffenburg 2023 RoboCup 3D Simulation Team

Tobias Biehl, Nico Bohlinger, Hannes Braun, Klaus Dorer, Stefan Glaser, Patrick Grommelt, Markus Portugall, Jannes Scholz, Louis Weiss, Maren Wolffram[1]

Hochschule Offenburg, Elektrotechnik, Medizintechnik und Informatik, Institute for Machine Learning and Analytics, Germany

Abstract. Team description papers of magmaOffenburg are incremental in the sense that each year we address a different topic of our team and the tools around our team. In this year's team description paper we focus on the architecture of the software. It is a main factor for being able to keep the code maintainable even after 15 years of development. We also describe how we make sure that the code follows this architecture.

1 Introduction

One of the key challenges in developing a team's code is to keep the code maintainable over years. One of the most important ingredient into this is to have an architecture defined that ensures a clean structure of the software without dependency cycles. A second, evenly important ingredient is to make sure that the code is following this architecture and to avoid architectural drift. In this paper we show how the architecture of the magmaOffenburg team is designed and which tools are used to enforce this architecture.

2 Architecture

2.1 First- and second-level

Figure 1 shows the first- and second-level architecture of team Magma. Module kdolib is our own algorithms and machine learning library. Modules util and base contain code that can generally be used by any autonomous system. In fact, they are used by our autonomous driving team taco (see right side). RoboCup specific code is only contained in the agent module and, to a minor degree, in the wrapper of the RoboCup SimSpark environment for OpenAI gym in the deeplearning Python module. Dependencies are top down so that lower components are reusable.

In the second level of agent there is a clear separation of code that is only used during learning (learningbase, geneticlearning, deeplearning and monitor), code that resembles the tools used for developing and debugging and code that is finally used at runtime during games (magmaagent and common). There are

no cyclic dependencies and runtime code may never depend on learning or tools code.

All elements in Figure 1 and the dependencies are implemented as Maven modules. This makes sure, that no undesired dependencies can be introduced into the software by accidentally importing a class from a module, on which it should not depend. While Maven modules are a very effective measure to enforce architecture, the overhead of creating them is inhibitive to also use them on lower architectural levels.

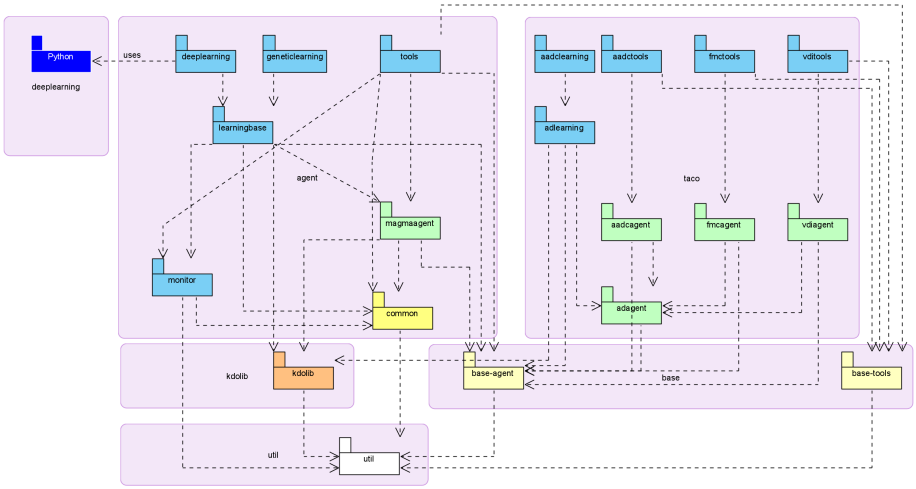

Fig. 1: First- and second-level architecture of Magma and Taco.

2.2 Third- and fourth-level

Figure 2 shows the third- and fourth-level architecture of the magmaagent component that contains the major parts of the RoboCup sepcific runtime elements for playing soccer.

Package communication contains all code for receiving perceptions from the robot (in Magma case from the server) and sending actions to the robot. Package model contains all the agent knows about itself (agentmodel and agentmeta), what it knows about the world around (worldmodel and worldmeta) and what it can conclude from that knowledge (thoughtmodel). The -meta packages contain knowledge known before a game starts, like body part information of the robot model in agentmeta or field sizes in worldmeta. Package decision contains the decision makers and behaviors. Agentruntime is the creator for these components and contains the main loop of agent with its sequential decision architecture.

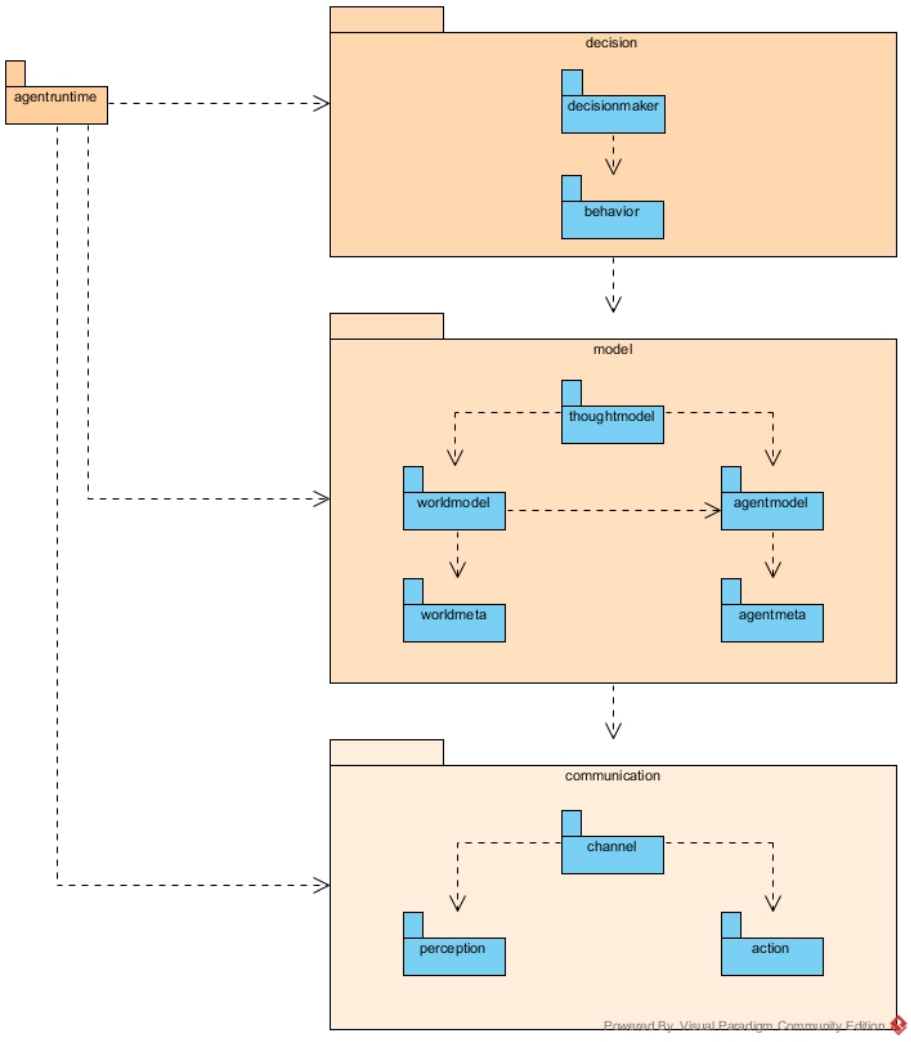

Fig. 2: Third- and fourth-level architecture of the magmaagent component.

This layered architecture allows, for example, that we can release only the communication layer to other teams, or the communication and the model layer without the decision layer. Lower layers always work and compile without the layers above them.

To ensure that our code follows the architecture in levels three and below, we use a not so well known but very powerful framework called ArchUnit[1]. With its fluent API it allows to very easily specify rules that must hold for dependencies between architectural layers or packages in general.

[1] https://www.archunit.org/

Listing 1.1 shows an example of a rule that completely ensures the access rules for the architecutral layers of level three. It is, of course, mandatory in general, that the package structure of the code follows the architecture.

```
@ArchTest
public static final ArchRule checkLayers = layeredArchitecture()
   .layer("decision").definedBy("..decision..")
   .layer("model").definedBy("..model..")
   .layer("communication").definedBy("..communication..")
   .layer("runtime").definedBy("..agentruntime..")
   .layer("robotsMain").definedBy("..robots")

   .whereLayer("communication")
   .mayOnlyBeAccessedByLayers("model", "runtime")
   .whereLayer("model")
   .mayOnlyBeAccessedByLayers("decision", "runtime", "robotsMain")
   .whereLayer("decision")
   .mayOnlyBeAccessedByLayers("runtime", "robotsMain");
```

Listing 1.1: Rule defining the access of the four architectural layers of third level architecture in magmaagent.

Listing 1.2 gives an example of how to define architectural rules for components in general. Specifically this rule makes sure that no code in agent, the package containing only code that does not depend on a specific robot type, depends on robot-specific code located in package robots and subpackages.

```
@ArchTest
public static final ArchRule agentRobots =
   noClasses().that().resideInAPackage("..agent..")
   .should().accessClassesThat().resideInAPackage("..robots..");
```

Listing 1.2: Rule to check that no robot-unspecific classes of agent and subpackages should access robot-specific classes in robots and subpackages.

3 Conclusion

Although it is additional effort to design and maintain an architecture, it pays back in multiple ways especially in complex autonomous systems:

- Components may be reused as our base components that are used for Magma, our real robot Sweaty and autonomous driving in Taco.
- Components can be more easily exchanged with other components
- Components can be tested independently
- Components may be released on various architectural levels
- Finally, it simplifies the introduction of new team members that get a better overview of the software.