Applying Umple to the rover control challenge problem: A case study in model-driven engineering

Timothy C. Lethbridge and Abdulaziz Algablan

Electrical Engineering and Computer Science, University of Ottawa, K1N 6N5, Canada
timothy.lethbridge@uottawa.ca, aalga075@uottawa.ca

Abstract. We demonstrate the use of the textual modeling language called Umple on a software modeling challenge problem that requires participants to create an executable model of a robotic rover that must follow another rover at a safe distance. We used a wide variety of features of Umple in the solution, including class models, state machines, aspects, and mixsets (feature models). We implemented two separate solutions to the problem. Our first solution, in which pulses of full power of varying lengths alternate with direction changes, performs significantly better than our second solution in which power and direction are blended together and varied in amplitude. Overall Umple proved very versatile at exploring various dimensions of this problem.

Keywords: Umple, autonomous control, mixins, state machines, case study.

1 Introduction

In this paper we present a solution, written in Umple, to the MDETools 2018 workshop challenge problem [1]. The problem involves arranging for a rover to follow a randomly moving leader rover in a simulated environment. The challenge is to create a model that can generate code whereby the follower stays neither too far (15 simulation distance units by default) from the leader nor too near (12 units) to it, and turns appropriately to follow it. Follower speed and direction primarily has to be controlled by independently varying power to left- or right-side sets of wheels; braking is also available.

Our solution involved static and dynamic modeling. The static model was a class model of the two types of rovers, with various attributes and associations. For the dynamic part of the problem – autonomous following – we used two distinct models. The first we call 'frequency modulation'. It involves applying pulses of 100% forward thrust to both sides, as well as turning pulses (one side 100% forward, the other side 100% reverse). The pulses are varied in duration (and hence frequency), with longer pulses being needed when greater corrective action is needed. This mimics the way a human might manually operate the follower rover; indeed it was inspired by spending time using a manual interface provided by the workshop organizers.

The second approach we call 'amplitude modulation'. It uses a state machine that adjusts the power of left side and right side wheels at regular intervals. The amount of power applied is varied depending on the need for corrective action. Similarly, the

greater the need for turning, the greater the difference between power applied to left and right wheels.

While it was easy to tune the system to give good following results for the frequency modulation approach, we were not able to effectively tune the amplitude modulation approach. The main lessons from this paper are, however, the effectiveness of Umple for modeling the problem and developing the solution. Umple features such as state machines, mixsets, its dual textual-diagram nature, and the detailed feedback it gave to point out model weaknesses all facilitated rapid development of our solution.

In the next section we give an overview of Umple. Following that, we give a detailed narrative of the steps we followed to develop the solutions, and our lessons learned.

2 Brief overview of Umple

Umple [2] [3] is an open-source textual language for representing software models and blending them with programming-language code. It is intended to make modeling easy for developers, presenting them with a familiar-looking syntax and freedom to use the toolchain of their choice. It can be used as through the UmpleOnline website, in Eclipse, or on the command line. Umple has proved particularly useful among modeling educators [4] [5].

Umple's features are described in its 530-page user manual [6], with over 430 examples, as well as in 35 peer-reviewed publications and 14 theses. Some of its modeling constructs include classes, interfaces, attributes [7], enumerations, associations [8], constraints, patterns, methods, state machines [9], generation templates, active objects [10] and trace directives [11]. Umple incorporates a subset of UML features and follows UML semantics for those features; it does not, however, attempt to include all of UML. Beyond UML, it supports several approaches to separation of concerns that are suited to its textual nature, including mixins, traits [12], aspects, mixsets [13], and filters. It can generate, incorporate, and/or be incorporated in Java, C++ and Php.

Although it can be made to 'play nicely' with models and tools that are part of the Eclipse/Papyrus/EMF ecosystem, it does not by design have any dependencies on that ecosystem.

In the remainder of the paper we will present and explain a few snippets of Umple as we illustrate the work we performed on the challenge problem. The reader is invited to view our repository [14] for complete details of the model.

3 Method used to develop the solutions

We followed an agile approach to developing our solutions. This involved small iterations with testing and re-planning after each iteration.

Each iteration is outlined in this section. We have documented in detail the process we followed to help others understand experiences in using Umple, and to allow comparison of tools. The resulting code, at various stages has been placed in a Github repository umple/roverChallenge [14]. Releases of the jars have been created so the reader can check out and run the simulation at any stage.

3.1 Iteration 0: Converting to Umple and getting a basic controller working

We created the first version of our system in the following three sub-steps:

Operationalizing the provided manual program. First we ensured we could run the provided (manual-controller) version of the challenge problem [1]. This includes three executables: 1) The rover simulator, a visual environment made with Unity. 2) The Observer environment, a Java application; and 3) the provided RoverController that allows the user to manually use keyboard keys to give commands to the follower. These three communicate using TCP/IP. The challenge problem requires replacement of only the third component with an autonomous controller.

The provided application also comes with a configuration file allowing changing various parameters, such as how erratic the leader should be in terms of speed and changes of direction. Although we experimented informally with adjusting these parameters, we elected to leave them exactly as they were provided for the remainder of our work. This will allow other models to be compared to ours more easily.

We encountered some initial obstacles getting the provided setup working on our Macintosh. However these turned out to be caused by omissions in the instructions for manually controlling the follower, and were fixed after consulting the organizers.

After getting the manual controller working, we spent approximately 30 minutes controlling the follower manually in order to better understand the behavior of the leader and strategize about how we would model an autonomous follower. Manual control using the provided interface involves pressing the 'up arrow' key to apply full power to the wheels; releasing the key causes the rover to 'coast' over the simulated gravel environment. We found that the best way to cause the follower rover to maintain an optimal distance from the leader is to pulse the power by pressing the key for short intervals (about once a second for half a second), but more often and for longer if the follower rover falls behind, and ceasing the pulses if the follower gets too close. Manual control for steering is by pressing the left or right arrow keys; this causes full power to be applied to the wheels on one side (e.g. left to turn right) and full *reverse* power to be applied to the opposite set of wheels. We found that pulsing this power by pressing these keys for brief intervals, between forward pulsing, best enables course correction. These observations became the basis for our Frequency Modulation solution discussed in Section 3.2 below.

We committed the original version of the code to our repository [14].

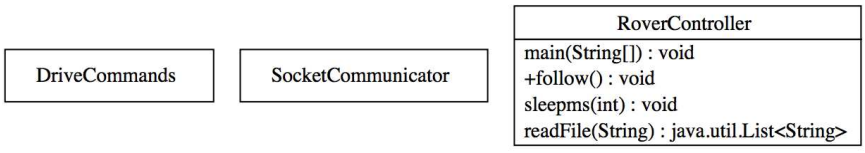

Fig. 1. Umple-generated diagram of version 1 of the solution to the challenge problem with a basic dumb follow method

Umplification of the provided manual controller. Umplification [15] [16] is a key process recommended for many adopters of Umple that are faced with an existing code base. It involves taking code written in a language such as Java and converting it into Umple. This allows developers of legacy software to gradually and painlessly convert their code so that it becomes model-based, all the while making sure tests pass at every stage. The 'Umplificator' tool [16] would have allowed this to be done automatically, although for this relatively small example, we did it manually, since there were only about 5 lines of code to change.

The provided manual controller code consists of three files. SocketCommunicator.java, DriveCommands.java and RoverController.java. We elected to initially only umplify the third of these, and leave the others as 'external libraries', in order to demonstrate an important feature of Umple: interoperation with other tools, in this case plain Java. The umplification of RoverController.java was a one-minute process, since the Umple compiler accepts Java syntax in almost as-is. The result was a file called RoverController.ump. We simply had to convert Java 'import' statements into Umple 'depend' statements and declare the other two files as external Java classes using Umple's 'external' keyword. We then used the manual rover controller user interface, described above, to verify that the resulting program (compiled now by Umple) behaved in the same way as the originally-provided program.

We used the command-line version of the Umple compiler for this work, as well as UmpleOnline for working with diagrammatic views of the models.

Stripping the manual control and adding dumb controller logic. The next iteration involved stripping all the manual control logic, including the user interface that accepted keystrokes, from the system. Control of the rover was replaced by a single short method called 'follow()' that simply drives straight, makes a turn and drives straight again. In other words it does not actually pay any attention to the leader, but was written to ensure that the system still could be compiled and executed.

It is worth noting at this stage that although the RoverController class is written in Umple, it is still not at this stage using any of Umple's modeling capabilities. The follow() method is just plain Java. One of the key features of Umple is that plain Java can be embedded in Umple. However a diagram can nonetheless be drawn of the system by Umple, and this appears in Figure 1.

The end-result of this step was committed to the repository [14] and tagged as V1. A release of the V1 jar is also available.

3.2 Version 2: The 'Frequency Modulation' (FM) solution

Our next step was to start altering the Umple model to enable autonomous control. In the RoverController.ump file, we defined an abstract Rover class that has attributes and methods common to the leader and follower. We added subclasses for the latter two. Implementations of methods allow instances of these classes to be updated with current information about their position by querying the connected servers.

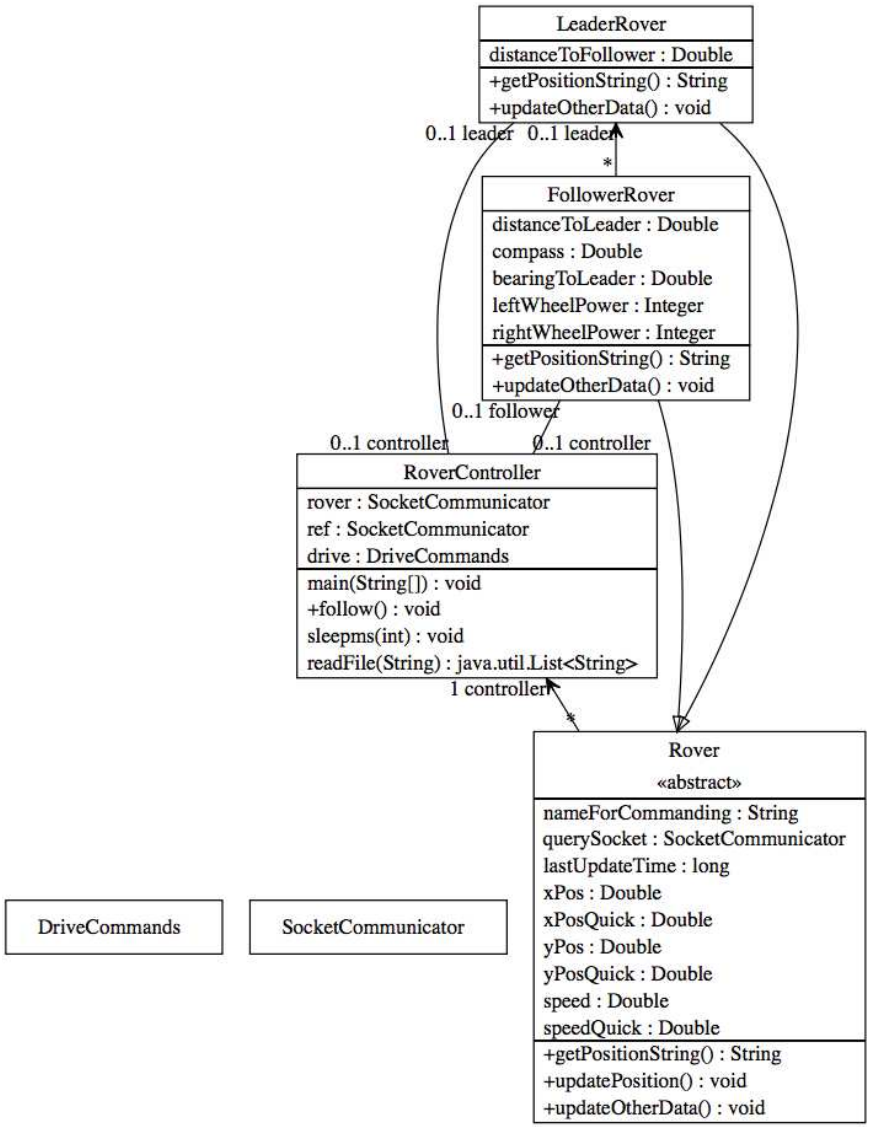

Fig. 2. Class diagram from version 2 with a moderately intelligent follow method

Of particular interest is a method that computes the bearing from the follower to the leader, enabling it to know what course it must optimally follow. This is line 151 in RoverController.ump and is as follows:

```
bearingToLeader= Math.toDegrees(Math.atan2(leader.getXPosQuick()
- getXPosQuick(),leader.getYPosQuick() - getYPosQuick()));
```

Central to the following approaches we present later is turning the rover such that the bearingToLeader is within 5 degrees of the follower's compass heading.

```
1   class Rover {
2     abstract;
3     String nameForCommanding;
4     SocketCommunicator querySocket;
5     long lastUpdateTime = 0L;
6     * -> 1 RoverController controller; // association
7     Double xPos = 0.0;
8       before getXPos {updatePosition();} // get from server
9     Double xPosQuick = {xPos}; // get from cache
10    Double yPos = 0.0;
11      before getYPos {updatePosition();}
12    Double yPosQuick = {Pos};
13  }
```

Fig. 3. Extract of Umple model for the Rover class

Figure 2 is a class diagram generated by Umple. Figure 3 is a sample of Umple text describing the Rover class. Some features of Figure 3 are worth pointing out:

- Line 2 has the keyword 'abstract' this is Umple's syntax to declare a class as abstract. We call such keywords 'stereotypes' following UML conventions.
- Lines 3,4,5,7 and 10 define various attributes. They are like variable declarations but have more sophisticated semantics, aligning with UML conventions, but also allowing 'aspects' to systematically adjust their generated code.
- Lines 8 and 11 are simple aspects that ensure that before accessing the data, it is updated from the server.
- Lines 9 and 12 allow bypassing this update, if the developer knows it has recently been performed.

After tuning, we were able to ensure that follower follows the leader in all circumstances, but it errs on the side of following too far (following too close can lead to crashes). Table 1 shows that version 2 of the follower on average manages to stay in the required zone 53.2% of the time when tested with the provided testing module UnityObserver.jar. Version 2 of the RoverController jar is available in our repository.

Version 2a: Improving model quality and exploring parameter variation. Our next step was to refactor and improve the Umple model. Unlike Java and many other languages, Umple does not force the user to keep one class per file. In fact, a typical Umple model would have a several-to-several relationship between files and class fragments. Files in Umple should be seen as units of human-understandable functionality. Umple's ability to distribute class fragments among multiple files might hypothetically make it hard to find needed code or model elements. However in practice, we have found this is not an issue: Javadoc generated from Umple helps the user locate the needed elements, as do diagrams generated by Umple and simple searching.

Figure 4 presents the File model for distribution of functionality after refactoring. We left the pure Umple class model into RoverController.ump. We moved utility logic

such as code that obtains data from servers and calculates values such as bearings to RoverController_code-updateData.ump, and the follow() method to RoverController_FMfollow.ump. We also extracted hard-coded parameters needed for the control algorithm to RoverController_FMparameterSets.ump.

As a final step to create version 2a of our controller, we experimented with changing values of key algorithm parameters. In particular we had originally decided to allow the lower bound for the time-period of power pulses, and also the coasting period between pulses, to be 500 ms, mimicking our manual control method discussed in Section 3.1. To make the controller more responsive we reduced this to 250ms. Table 1 shows this resulted in much better in-zone following performance. We tried to reduce these parameters to be 125ms, or other values between 125 and 250ms, however we ran into problems: Messages sent to the simulation and the RoverController were sometimes taking more than 125ms for the round trip (message sends plus processing), and this led to erratic behavior. This is classic deadline-miss behavior common in real-time systems. We settled on 250ms as the shortest time that resulted in stable behavior.

Table 1. Test results for FM versions 2 and 2a (10 tries each)

	Version 2, 500ms pulses min				Version 2a, 250ms pulses min		
	Percent in zone	Percent too close	Percent too far		Percent in zone	Percent too close	Percent too far
Mean	53.2%	6.6%	40.3%		65.5%	6.3%	28.1%
Std. Dev.	21.0%	4.1%	20.0%		14.4%	4.0%	14.4%
Max	82.9%	13.8%	67.1%		85.8%	10.4%	57.9%
Min	27.5%	0.0%	14.6%		42.1%	0.0%	10.0%

- RoverController.ump Class Model: RoverController, Rover, LeaderRover, Follower Rover
  - Uses: DriveCommands.ump: Originally provided umplified code
  - Uses: RoverController_code-updateData.ump: Updates Data
  - Uses: RoverController_code-general.ump: Utilities
  - Uses if mixset FM: RoverController_FMfollow.ump: Loop follow method
    - Uses RoverController_FMparameterSets.ump
  - Uses if mixset AM: RoverController_AMfollow.ump: State machine
    - Uses RoverController_AMparameterSets.ump

Fig. 4. File and feature model

3.3 Version 3: The 'Amplitude Modulation' (AM) solution

For version 3, we attempted to create a totally different algorithm. Whereas in version 2, we had pulsed full power for varying amounts of time, with full 100%/-100% power application for direction changes if needed. In version 3, we tried instead adjusting power at values *less than 100%*. We also attempted to turn by varying the power to left and right wheels – with greater difference when more turning is needed.

This was achieved by using a pair of state machines to control distance and direction. The diagram for the latter appears in Figure 5, and the Umple code appears in Figure 6.

Full details are in the repository. Textual state machines in Umple can be identified as a state machine name followed by a brace, so line 2 of Figure 6 marks the start of a state machine. Line 3 marks the start of a state. Line 5 indicates an action to take on entry to the state. Line 6 specifies a guarded transition, with the guard in square brackets, and the target state appearing after the -> symbol. The words in all-caps are tuning parameters that can be adjusted in the file RoverController_AMparameterSets.ump.

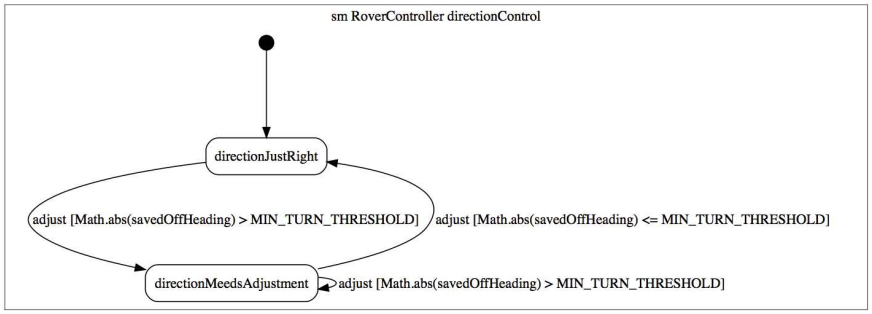

Fig. 5. Direction control state machine as a diagram

```
1   class RoverController {
2     directionControl {
3       directionJustRight {
4         // Set wheels to be equal
5         entry / {setLrDifference(0);}
6         adjust [Math.abs(savedOffHeading) > MIN_TURN_THRESHOLD] ->
7            directionNeedsAdjustment;
8       }
9       directionNeedsAdjustment {
10        // Set wheels to be different
11        entry / {
12          setLrDifference(-(int)savedOffHeading *
13            DIRECTION_MULTIPLIER);
14        }
15
16        // Direction is OK now
17        adjust [Math.abs(savedOffHeading) <= MIN_TURN_THRESHOLD]
18           -> directionJustRight;
19
20        // re-check direction every cycle
21        adjust [Math.abs(savedOffHeading) > MIN_TURN_THRESHOLD] ->
22           directionNeedsAdjustment;
23      }
24    }
25  }
```

Fig. 6. Extract of the direction control state machine as Umple text

Although we spent much time trying to fine-tune the parameters for this AM approach, we were never able to get it to perform as well as the FM approach.

We have nonetheless left the AM approach in the code base, but have switched it off by commenting out the mixset (feature) that activates it in the feature model. A user

running version 3 of our system will achieve the same results as for version 2a unless they uncomment RoverController.ump line 22 and comment out line 21.

Results of executing the Amplitude Modulation solution are found in Table 2.

Table 2. Test results for AM version 3 (10 tries)

	Version 2, 500ms pulses min		
	Percent in zone	Percent too close	Percent too far
Mean	39.9%	7.0%	53.1%
Std. Dev.	15.0%	2.3%	15.5%
Max	71.3%	9.6%	72.5%
Min	21.3%	3.3%	22.1%

4 Key lessons learned

We wrote 732 lines of Umple model/code. Generated Java was about double that. We did not find any bugs in Umple in this project, and were able to complete it in a day.

Some Umple features that proved particularly useful were the following:

- The Umple compiler's error messages were precise and allowed us to rapidly solve the few syntax or semantics errors we made. It usually just took minutes from the time of completion of writing any update of the model/code to the time when we were able to get the new version running. The majority of development time was spent strategizing about the model, tuning model parameters, and running the simulation. Debugging of Umple or embedded Java was less than 5% of total time spent.
- It was nice to be able to see both textual and graphical views of the model, as they were able to serve as quality checks on each other.
- Embedding of algorithmic code (as needed for calculations) in the model helped simplify development. These were added as derived attributes, state machine actions, and ordinary methods.
- Injection of code into generated methods, such as to load data from the server and send control messages, using Umple's 'before' and 'after' constructs, proved very useful and effective.
- Umple's mixset capability, allowing alternative versions (AM and FM agorithms) of our product line served well in the solution to this problem.

The 'FM' version of the solution, using pulses of full power, worked better than the 'AM' version, that varied power amplitude. The software that was provided for the challenge did not facilitate the automation of parameter tuning because each run would take several minutes. Nonetheless, we were able to develop a solution by manually adjusting parameters that kept the follower at the desired distance 65.5% of the time.

5 Conclusions and future work

From the perspective of the authors, Umple's features discussed in the last section made it both useful and easy to use for this project. They allowed us to effectively solve the core modeling problem, which was to enable a rover to autonomously follow another.

We tried two dynamic autonomy models. The one that worked best involved pulsing power to the wheels. We suspect this is because the follower needs to apply full power much of the time in order to keep up with the leader, and perhaps because pulsing might help prevent skidding, in the same manner as pulsing car brakes stops skidding.

As future work it would be nice to instrument the simulation so the parameters can be learned via machine learning. Many variations of the model could also be tried.

References

1. MDETools Workshop, "Challenge Problem", https://mdetools.github.io/mdetools18/challengeproblem.html (Visited July 2018)
2. University of Ottawa, "Umple Home Page", http://www.umple.org (Visited July 2018)
3. Github, "Umple Model-Oriented Programming", https://github.com/umple/umple (Visited July 2018)
4. Lethbridge, T.C. "Teaching Modeling Using Umple: Principles for the Development of an Effective Tool", CSEE&T 2014, IEEE Computer Society, Austria, pp 23-28 (2014)
5. Agner, Luciane T. W. and Lethbridge, T.C., ("A Survey of Tool Use in Modeling Education", Models 2017, IEEE Computer Society, pp 303-311 (2017)
6. University of Ottawa, "Umple User manual", http://manual.umple.org (Visited July 2018)
7. Badreddin, O, Forward, A., and Lethbridge, T.C., "Exploring a Model-Oriented and Executable Syntax for UML Attributes", SERA 2013, Springer SCI 496, pp. 33-53
8. Badreddin, O, Forward, A., and Lethbridge, T.C., "Improving Code Generation for Associations: Enforcing Multiplicity Constraints and Ensuring Referential Integrity", SERA 2013, Springer SCI 496, pp. 129-149.
9. Badreddin, O., Lethbridge, T.C., Forward, A., Elasaar, M. Aljamaan, H, Garzon, M., "Enhanced Code Generation from UML Composite State Machines", Modelsward 2014, Portugal, INSTICC, pp. 235-245 (2014)
10. Husseini-Orabi, M., Husseini-Orabi, A., and Lethbridge, T.C. "Concurrent Programming using Umple", Modelsward 2018, pp. 575-585 (2018)
11. Aljamaan, H., Lethbridge T.C. "MOTL: a Textual Language for Trace Specification of State Machines and Associations", Cascon 2015, ACM, 101-110 (2015).
12. Abdelzad, V, and Lethbridge, T.C. "Promoting Traits into Model-Driven Development", Software and Systems Modeling, 16:997–1017 (2015)
13. Lethbridge., T.C. and Algablan, A. "Using Umple to Synergistically Process Features, Variants, UML Models and Classic Code", ISOLA, October, Springer (2018).
14. Github, " Umple case study implementing an autonomous rover", https://github.com/umple/roverChallenge (Visited July 2018)
15. Lethbridge, T.C., Forward, A. and Badreddin, O. "Umplification: Refactoring to Incrementally Add Abstraction to a Program", WCRE, Boston, October 2010, pp. 220-224
16. Garzon, M., Lethbridge, T.C., Aljamaan, H., and Badreddin, O. "Reverse Engineering of Object-Oriented Code into Umple using an Incremental and Rule-Based Approach", Cascon, ACM (2014)