Comparison Between Different Abstraction Level Programming: Experiment Definition and Initial Results

Janne Merilinna and Juha Pärssinen

VTT Technical Research Centre of Finland, P.O. Box 1000, 02044 Espoo, Finland
{Janne.Merilinna, Juha.Parssinen}@vtt.fi

Abstract. Domain-specific languages and especially domain-specific modeling languages (DSML) are mentioned to achieve 5-10 times performance gains compared to traditional software development practices due to raising the level of abstraction. The data for the cases where these gains have been witnessed is usually not available. Therefore, in this paper, we introduce a simple but comprehensive and affordable experiment framework which can be utilized for measuring the benefits and drawbacks of DSMLs in an open fashion, i.e. publishing the data and the results and enabling a possibility to repeat the experiment by others. In this paper, we also present our own experiences and initial results about the benefits and drawbacks. We found the benefits of DSML to be clear if the applications have to be implemented daily and especially if the platform continues to evolve and the existing applications have to be updated to correspond to the changes. The benefits of utilizing DSML in a domain where the platform continues to evolve came as a surprise and needs further study.

1 Introduction

Quite often domain-specific languages (DSL) and domain-specific modeling languages (DSML) are mentioned to attain 5-10 times performance gains compared to traditional software development practices [1]. Usually, the gains witnessed in industrial cases have to be trusted blindly as no data is available for further study. In addition, in order to achieve a fair and reliable comparison, the cases have to be performed by utilizing both approaches, i.e. implementing the system manually and by utilizing DSML, which may not be so in true industrial cases. Therefore, this paper strives to set an initial starting point for investigating and comparing the benefits of traditional software development and DSML in a reliable fashion by introducing a simple but comprehensive enough experiment framework that can be utilized for the comparison.

The selected environment for the programming experiment is the Lego Mindstorms NXT Kit which enables the creation of thousands of different kinds of automatons consisting of LEGO elements, a 32-bit programmable microcontroller, a set of sensors and servo motors, and Bluetooth and USB connections [2]. In this experiment, we created a warehouse keeper (WaKe) tripod which moves by utilizing its servo motors and monitors and senses its environment through its sensors. The purpose of the WaKe automaton is to deliver packages by following routes painted on the warehouse floor. The route which the automaton takes is programmable by the application developer.

This paper also presents the researchers' experiences in implementing software for the WaKe automaton by applying traditional means, i.e. manually coding, and by utilizing custom made DSML for the automaton. Experiences gathered in implementing the automaton forms the foundation for comparing the development approaches.

This paper is structured as follows. First, a selected programming experiment is introduced by discussing the elements existing in the warehouse domain, in addition to what the automaton should do and how the automaton is built. Second, implementation of the automaton including presentation of the DSML for the automaton is presented. Third, an evaluation framework for comparing the approaches in developing the automaton is presented. After that, the approaches are briefly compared followed by discussion about the experiment. Final remarks close the paper.

2 Lego Mindstorms as an Experiment

2.1 Domain Analysis

In this experiment, a simplified WaKe automaton was created which delivers packages from point A to point Z via zero to n (where n is zero or more) waypoints. The starting point, end point and waypoints are connected to each other with a *route* which is painted on the floor of a warehouse. The *route* must be followed strictly, i.e. no waypoints should be skipped and no curves straightened. A small variation in tracing the *route* is allowed but in general, the *route* should be followed as precisely as possible in order to avoid collisions with the environment or other automatons in the warehouse.

As discussed, the main objective of the automaton is to take a package from point A (points are later referred to as points of interest, *POI*s) to *POI* Z by following a *route*. The main *scenario* for the automaton may contain several (independent) sub package delivery *scenario*s, called *task*s, which as a whole may consist of tens of *POI*s. These independent *task*s are usually combined together in order to form a complete *scenario* for that day. Next in this section, concepts of the domain are described more precisely.

Scenarios and Tasks

The delivery duties are called *scenario*s which consist of sub scenarios called *task*s. Let's take an example of a *scenario* and its *task*s. Consider *task*s A and B. The objective for *task* A is to take a package and deliver it to somewhere in the warehouse. The purpose of *task* B is to drive the automaton back to the same place (not necessarily following the same *route*) where *task* A started. *Scenario* AB then encapsulates these two *task*s.

Point of Interest and Route

A *task* consists of a few *POI*s all of which have to be visited in some specific order to complete the *task*. When the automaton reaches the *POI*, it performs its actions set for the *POI* and continues (or ends) the *task*. No physical *POI* should be skipped.

There are 1 to 4 *route*s for each *POI*. The *route*s enter the *POI* from orthogonal directions, i.e. from the north, east, south and west. No half-cardinal points exist. Considering the *route*, a *POI* must exist at every cross-road. During the *route* a few events may occur which must be reacted to correctly.

Events and Actions

Next, all the actions and triggers that can occur in *task*s are discussed:

- There are three actions that can be performed at the *POI*:
  - taking the package,
  - releasing the package, and
  - no action.
- In order for the automaton to start moving from the *POI*, the automaton waits for three kinds of signals:
  - two 'beep' sound signals,
  - an obstacle being closer than some certain specified distance, e.g. an external supervisor waving hand on front of the automaton, and
  - a period of time.

- While on the move, i.e. on *route*, two kinds of signals trigger the automaton to stop immediately:
  - a single 'beep' sound signal, and
  - an obstacle being closer than some certain specified distance.
- If the automaton is halted on *route* two signals trigger the automaton to start moving again:
  - if the automaton is stopped by encountering an obstacle, removing the object enables the automaton to start moving again, and
  - if sound signals have stopped the automaton, two 'beep' sound signals enable the automaton to start moving again.

2.2 The Automaton

The automaton is a two-wheel driven tripod with a third wheel at the back for stabilizing the automaton. Yaw and speed are controlled by two independently driven servo motors which are at the left and right sides of the automaton. The automaton also has a jaw controlled by a servo motor at the front which is utilized for grasping objects.

The automaton is equipped with four sensors [2]:

- A touch sensor, which is located and pointed to the front of the automaton between the jaws, giving the automaton a sense of touch.
- A light sensor, which is located beneath the touch sensor and is pointed downwards to the ground, enabling the automation to distinguish between light and dark, i.e. it reads light intensity, not the actual colour.
- An ultrasonic sensor, which is located on top of the touch sensor and is pointed forwards, enabling the automaton to measure distances between itself and obstacles ahead.
- A sound sensor, located on top of the NXT brick, enabling the measurement of sound pressure, i.e. enabling the automaton to hear.

In addition to these sensors, the automaton is equipped with a timer. The following bullets summarize the events triggered by the sensors and the timer and actions to be taken during different phases of *task*s, and the responsibilities of the servo motors.

- Wheels
  - Two servo motors attached to the wheels of the automaton are responsible for controlling speed and yaw.
- Jaw
  - The jaw can be opened and closed by controlling the attached servo motor.
- Touch sensor
  - If pushed while the robot is waiting for a package, it closes its jaws and grasps the package.
- Sound sensor
  - If a single 'beep' sound signal is detected, the automaton stops its current action immediately.
  - The automaton continues its actions if two 'beep' sound signals are detected.
- Light sensor
  - The automaton utilizes its light sensor in order to track the *route*.
  - The light sensor is utilized to notice *POI*s on the ground.
- Ultrasonic sensor
  - The automaton senses objects in its path while travelling. If its path is blocked, the automaton comes to a standstill until the object is removed.
  - When at a *POI*, the ultrasonic sensor is utilized to trigger the automaton to leave the *POI* and continue the *task*.
- Timer

- When at a *POI*, after a certain specified time, the automaton is triggered to leave the *POI*.

2.3 An Application for the Automaton

The example application consists of *task*s A and B where both *task*s consist of three *POI*s. Figure 1 depicts the geometrical layout of the *scenario*. In figure 1, the visiting order of the *POI*s are A, B and C in *task* A. In *task* B, the order is C, B and A. Next, *task*s are discussed more precisely.

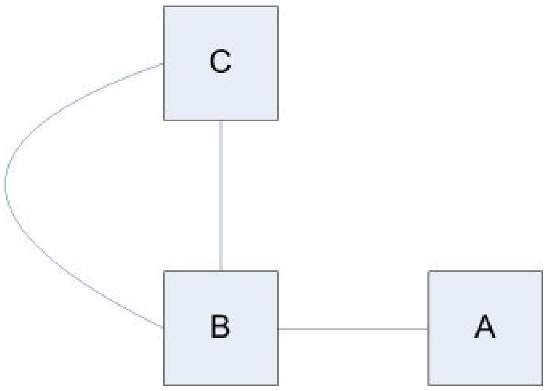

Figure 1. Scenario for the experiment.

Task A

The automaton faces to the west at the starting position. In the beginning, the automaton takes the package and waits for either 2000 milliseconds or specific sound signals, i.e. two 'beep' sound signals. When the trigger arrives, the automaton exits the *POI* to the west and begins its journey to the next *POI*. While on *route*, the automaton monitors the environment with its ultrasonic sensor in order to notice obstacles. Next, the automaton enters the second *POI* from the east. At the second *POI* the automaton takes a turn to the north (turns right, as it entered this *POI* from east) and then waits for an exit signal. In this *POI*, exiting the *POI* is triggered by waving a hand as an "everything ok" signal in front of the automaton from an external supervisor. On *route* to the final *POI* in this *task*, the automaton can be stopped by encountering an obstacle similarly as it was in the first leg of this *task*. The automaton enters the final *POI* from the south and releases its package, ends this *task* and moves to the next one automatically.

Task B

The *task* begins from the same *POI* where *task* A ended. In this *task*, the main objective is to go back to the starting *POI* of *task* A but using a different *route*. The automaton waits for 1000ms and then exits to the west (turns left, as it entered this *POI* from the south) and begins its journey to the next *POI*. While on *route*, the automaton monitors the environment similarly as before but also listens for sound signals. The automaton enters the *POI* from the west and exits after waiting for 1000ms to the east towards the final *POI*. In the final leg of this *task*, the automaton rushes to the starting point of *task* A and finishes the *scenario*.

3 Implementation of the Automaton

The purpose is to implement software which makes possible to program different *scenario*s for the automaton. The idea is that the application developer could program the automaton in an easy way without knowing every detail of the underlying platform, i.e. Lego Mindstorms NXT.

Two approaches in developing WaKe automaton were tested: use of

- Open-source language Not eXactly C [3] and IDE as a traditional counter-part in this experiment. NXC is textual DSL for the Lego Mindstorms environment, and is close to C programming language.
- DSML programming environment applying MetaEdit+ instead of utilizing Mindstorms's included LabVIEW programming environment which is a graphical language where the programming is performed by connecting boxes together like Lego bricks. LabVIEW can be considered as a general purpose graphical language for the Lego Mindstorms whereas this custom-made DSML is tailored for one particular domain of our automaton, and in this sense it is more abstract than LabVIEW and NXC.

As is common in the realm of DSMLs, the purpose is not to generate all the native source code but rather generate code on top of a pre-implemented framework or library [4]. Therefore, in this experiment, this approach is also taken. Figure 2 depicts the layering structure of the aspects that will be implemented manually or with the help of a DSML and a code generator. As shown, NXC is lying at the bottom of the stack being common to both development approaches. The WaKe library is also common to both approaches but unlike NXC it has to be implemented first. It is clear that much of this experiment depends on this library as the application developer is expected to implement his or her own *scenario* code on top of this layer.

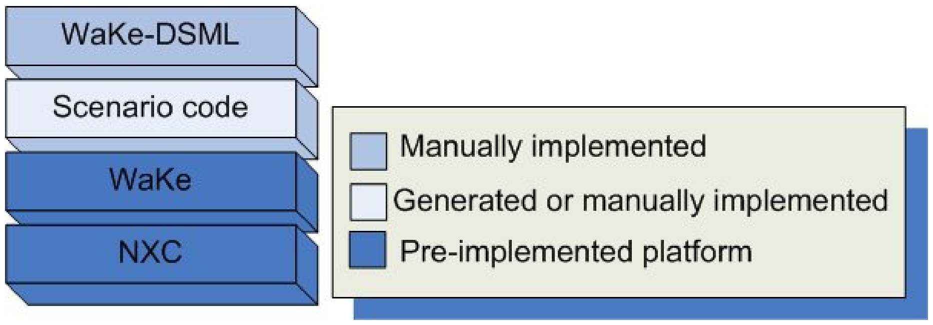

Figure 2. Layering structure for the source code of the automaton.

3.1 Application Programming Interface for the Automaton

As there was no previous experience with the Lego Mindstorms or with NXC, a familiarization period of a couple of days was conducted. In this period, experimenting and observing the automaton was conducted in order to gain an idea of its capabilities. Next, specifications for the automaton were iteratively and incrementally formed whilst the basic functionality for the automaton was developed. No up-front design and analysis for the library was performed, therefore the library was implemented more in a learn-as-you-go mode rather than carefully considering what aspects would be generated and how.

The resulting library consists of two parts:

- a runtime platform responsible for monitoring the sensors and moving the automaton, and
- an interface exposed for application developers.

The runtime platform consists of a few simultaneously running tasks, i.e. threads, each responsible for e.g. monitoring the sound sensor, detecting obstacles and controlling movement. The runtime makes the data available for the application programming interface (API) through global variables rather than actively throwing data to the API.

The API abstracts all low-level details making development of *scenario*s very easy. For instance, in order to make the automaton move and follow the *route*, the application developer writes a single line, startWalking(). Turning left, right and around is also simple. For instance, turnRight() function turn the automaton circa 90 degrees right and then finds the nearest *route*. The application developer does not need to worry about the fact that the automaton may not be directly on top of the *route* after taking a turn. The following code snippet (Figure 3) exemplifies the *scenario* code.

```
takePackage();
setObstacleDistance(10);
until(obstacleAhead()==true);
straight();
startWalking();
setObstacleDistance(10);
while(onPOI()==false)
{
   if(obstacleAhead()==true && onPOI()==false)
   {
      stopWalking();
      until(obstacleAhead()==false);
      startWalking();
   }
}
stopWalking();
```

Figure 3. An example of the scenario code.

3.2 A DSML for the Automaton

Implementing the DSML, named WaKe DSML, was conducted mainly after the implementation of the API and the runtime. Considering domain analysis, a domain expert's concepts [5] approach was a natural choice for defining *scenario*s for the automaton. Therefore, the WaKe DSML is composed of *POI*s, in this case a start *POI*, a stop *POI* and waypoint *POI*s, a *route* which connects the *POI*s, and actions and events that can occur while on *POI* or *route*. The resulting language implementing the *scenario* defined in Section 2.3 is depicted in figures 4, 5 and 6.

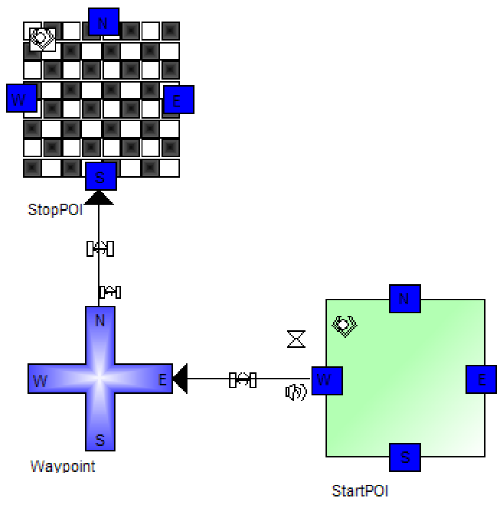

Figure 4. Defining a deliver package task for the automaton.

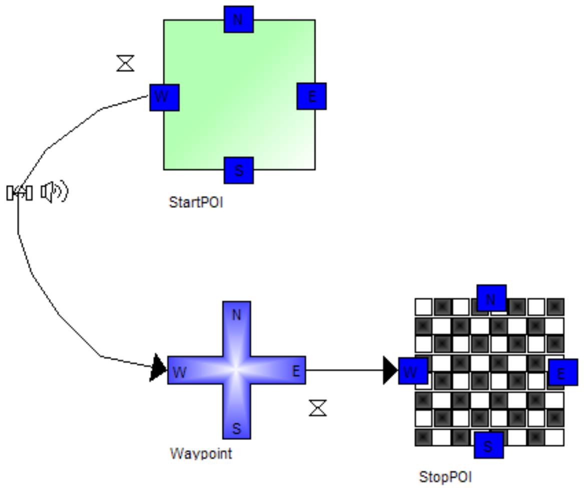

Figure 5. Defining a return home task for the automaton.

In the WaKe DSML, the application developer creates a start *POI* (green rectangle in figures 4 and 5) and a stop *POI* (checked rectangle) and a sufficient amount of waypoint *POI*s (the cross-shaped symbol), and then connects *POI*s with one-way arrows. In this language, *POI*s and arrows, i.e. the *route*, are relative in the sense that the location of the *POI*s in the diagram is not taken into account nor is the curvature of the arrows. However, ports (small rectangles at the edge of *POI*s) describe the direction in which the automaton heads from the *POI* and the direction from which the *POI* is entered. Actions taken at the *POI*s are described with small rectangles on the *POI*s. Events that trigger the automaton to exit the *POI*s are described at the start point of the arrows and events that trigger the automaton to stop while moving are described somewhat in the middle of the *route*.

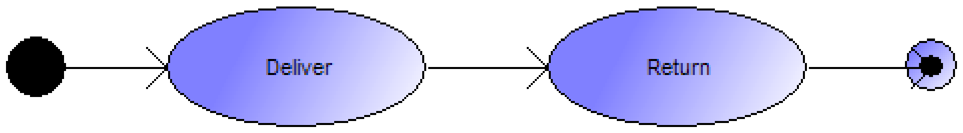

Figure 6. Defining a scenario that encapsulates the tasks.

Figure 6 describes a *scenario* structure. The *scenario* begins with a start symbol (filled circle) and ends with a stop symbol (bull's-eye) and in between there are the *task*s.

4 Comparison of the Development Approaches

4.1 Framework for Comparing the Development Approaches

In the context of this experiment, there are two aspects that can be divided further into a number of sub aspects (Table 1). These can be assessed for comparing the development approaches when the platform already exists. This in addition to comparing the time to develop the DSML and implement *scenario*s with it compared to implementing *scenario*s manually.

Table 1. Aspects to consider when comparing the development approaches

Experiment Framework		
Application	Scenario	Task Task order
	Task	POI Event
Platform	Constraints Elements Features New platform	

As depicted, at the application level one can manipulate *scenario*s by adding, removing and arranging the *task*s, and at the *task* level *POI*s and *route*s can be modified. One can measure and experiment with multiple aspects including at least:

- Time spent to learn the DSML vs. API,
- Time spent to develop a new *scenario* (of various sizes),
- Time spent to modify existing *scenario*s including reusing pre-implemented *task*s, and
- Reliability and intelligibility of the *scenario* code.

Where things get more complicated is when the underlying platform, i.e. WaKe in figure 2, starts to evolve and existing *scenario* code has to be adapted to the changed situation. In the context of WaKe, there are four aspects that can change:

- New constraints, e.g. *POI* can be exited if and only if the external supervisor waves a hand in front of the automaton,
- New elements, e.g. new types of *POI*s,
- New features, e.g. new sensors, and
- A new platform, i.e. replacing the existing WaKe API and runtime with another.

Now, one can measure at least:

- The time spent to implement new constraints, elements, and features into the code generator and DSML and the time spent to adapt (with a code generator) existing *scenario*s to the evolved platform vs. time spent to adapt existing *scenario*s manually.
- Reliability that the new constraints are actually implemented in the *scenario* code,
- Time spent to implement a new code generator for the new platform and adapt existing *scenario*s vs. teach the new platform to application developers and adapt existing *scenario*s.

4.2 Initial Experiment Results

The total time spent for implementing the library for the automaton was not calculated precisely as learning the NXC, making specifications for the automaton, experimenting with it and implementing the API and the runtime were performed simultaneously. However, relative workload can be defined where the total time spent was about two weeks. Most of the time spent (50%) was related somehow to the light sensor which is applied for following the *route* and for noticing *POI*s on the *route*. The logic for both of these is very simple but adjusting thresholds, i.e. defining the color of the *route*, *POI* and the environment, was time consuming. Implementation of the API and the runtime was fairly simple and thus took circa 20% of the time. The rest of the time (30%) was spent defining specifications and, especially, experimenting with the automaton.

As Metaedit+ was not previously known, the total time spent for developing WaKe DSML and the code generator for it also includes the learning phase of the tool. Developing the WaKe DSML was guided and coached by an experienced modeler who was familiar with the Metaedit+. Development took circa 10h while implementing the code generator took 15h.

The *scenario* code of various sizes (3-10 *POI*s, 1-4 *task*s) was implemented in both approaches several times while developing the automaton and after three months from the development. In this three-month period, both the API and the WaKe DSML were partially forgotten. It was significantly easier to re-familiarize ourselves with the WaKe DSML and implement *scenario* code with it compared to implementing the code manually albeit that both are extremely easy to utilize. The advantages of WaKe DSML when reusing existing *task*s in rather large *scenario*s was clear as one didn't have to think how to combine the *task*s, i.e. didn't have to think in which direction the automaton entered the last *POI* and what was the next heading. Overall, WaKe DSML was the natural choice when implementing any code.

Considering platform evolution, adding new constraints, elements and features were experimented with while implementing the WaKe DSML. The code generator being template-based, it was straightforward to add some e.g. documentation into the *scenario* code instead of updating existing *scenario*s manually. Overall, modifying the WaKe DSML and its code generator was considered worthwhile as after the updates, it was clear that all the new constraints etc. were actually followed. It was clear that spending some time in modifying aspects in one (or two) places and generating changes to 100 places is more worthwhile than implementing the changes manually.

The results presented so far were backed up by letting another person implement the API and runtime of her own without guidance thus resulting in a different implementation. Unfortunately, in this experiment, there was no time to re-implement the code generator and re-generate the *scenario* code with it. However, we can posit that this would also be worthwhile compared to porting the existing *scenario*s manually.

5 Discussion

Implementing *scenario* code for the automaton was conducted by manually coding and by means of DSML on top of a pre-implemented WaKe library. The WaKe library provides an opportunity for implementing the *scenario* code in a script-like fashion thus there is little need for other than direct function calls to the library. One could even say that the library on its own provides a DSL for the automaton. From this one could argue that what has been compared in this paper is not the traditional means vs. DSML but rather DSL vs. DSML. Ultimately, however, this is not true as with this DSL one could do a lot more than just implement *scenario* code since it does not restrict the application developer. In addition, DSML for this DSL would not look like the WaKe DSML if the abstraction level remains the same. DSML for this DSL would probably look more like a UML state machine diagram, i.e. a graphical representation of the *scenario* code. Wake DSML sees "domain as the real world" rather than "domain as a set of systems" [6].

Considering the Lego Mindstorms NXT Kit as an experimentation platform for comparing DSMLs (or UML) and traditional software development can be seen as worthwhile. It provides an open and especially affordable platform for experimentation thus enabling repetition of experiments, which is usually not the case in industrial scenarios. Thus, we encourage readers of this paper to repeat our experiment with a large number of attendees and report the results.

6 Conclusions

In this experiment, we built a warehouse keeper automaton from Lego Mindstorms NXT Kit and developed DSML for it in order to set a framework for comparing a traditional software development approach to that of DSML. We found that DSML provides a lot more opportunities and benefits than it does downsides even when the platform on which the code is generated was evolving. The traditional means for developing software for the automaton only proved to be better if it was intended to implement a limited number of applications and the platform remains unchanged. The benefits of the DSML approach were clear if the applications were to be implemented daily and especially if the platform kept evolving and the existing applications had to be updated to correspond to the changes.

References

[1] MetaCase website, URL: http://www.metacase.com [Visited at 2.5.2007]

[2] Lego, NXT Technology Overview, URL: http://mindstorms.lego.com/Overview [Visited at 2.5.2007]

[3] Next Byte Codes & Not eXactly C, URL: http://bricxcc.sourceforge.net/nbc/ [Visited at 2.5.2007]

[4] Roberts, D., Johnson, R., Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, Proceedings of Pattern Languages of Programs, 1996

[5] Luoma, J., Kelly, S. and Tolvanen, J., Defining Domain-Specific Modeling Languages: Collected Experiences, 4th Domain-Specific Modeling Workshop (DSM'04), 10p.

[6] van Deursen, A., Klint, P. and Visser, J., Domain-specific languages: An annotated bibliography, ACM SIGPLAN Notices, 35(6):26–36, June 2000.