Lego Mindstorms: an mbeddr Case Study

Markus Voelter[1] and Bernd Kolb[2]

[1] independent/itemis
[2] itemis

Abstract. This case study looks at the first large-scale demo case built with mbeddr: a set of C extensions for programming Lego Mindstorms robots. We have developed several robots (and the respective software) based on a common set of C extensions. Since C-based Mindstorms programming is based on the OSEK operating system, this case study has relevance beyond Lego.

Last Changed: Feb 1, 2012

1 Contacts for more Info

The Lego Mindstorms case study has been developed by the mbeddr team, in particular Bernd Kolb (`kolb@itemis.de`) and Markus Voelter (`voelter@acm.org`).

2 What is the problem/domain your system addresses?

Lego Mindstorms provides the ability to program lego models. The system comes with an ARM processor and a number of sensors and actuators to build various kinds of computer-controlled machines. The software can be written in various ways, from a visual programming language provided by Lego, a version of Java, as well as C. In this case study we use the C-based way of programming mindstorms.

The C-based way of programming Mindstorms is based on the OSEK operating system . OSEK is used a lot in automotive embedded software, which is why this case study is relevant beyond Lego. The OSEK implementation used for Mindstorms in Lejos OSEK .

3 Why was mbeddr/MPS chosen as the basis?

mbeddr was used as the basis because this case study was explicitly intended to be an mbeddr demo.

4 How does the system integrate/use mbeddr's existing capabilities?

For example, a interface `DriveTrain` supports a high-level API for driving the robots. We use pre- and postconditions as well as a protocol state machine to define the semantics of the interface.

```
exported c/s interface DriveTrain {
  void driveForwardFor(uint8 speed, uint32 ms)
    pre(0) speed <= 100
    post(1) currentSpeed() == 0
    protocol init(0) -> init(0)
  void driveBackwardFor(uint8 speed, uint32 ms)
    pre(0) speed <= 100
    post(1) currentSpeed() == 0
    protocol init(0) -> init(0)
  void driveContinouslyForward(uint8 speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init(0) -> new forward(1)
  void driveContinouslyBackward(uint8 speed)
    pre(0) speed <= 100
    post(1) currentSpeed() == speed
    protocol init(0) -> new backward(2)
  query uint8 currentSpeed()
  void stop()
    post(0) currentSpeed() == 0
    protocol *(-1) -> init(0)
  void turnLeft(uint8 turnDeltaSpeed)
    protocol init(0) -> init(0)
  void turnRight(uint8 turnDeltaSpeed)
    protocol init(0) -> init(0)
}
```

Components then implement the interfaces and keep global state. For example, the `DriveTrainImpl` component provides the `DriveTrain` interface and keeps track of the current speed, something that cannot easily be queried from the robot itself.

The motors are encapsulated into interfaces/components as well. This way, we can provide dummy/mock implementations to simulate the robot without using the Mindstorms hardware and API.

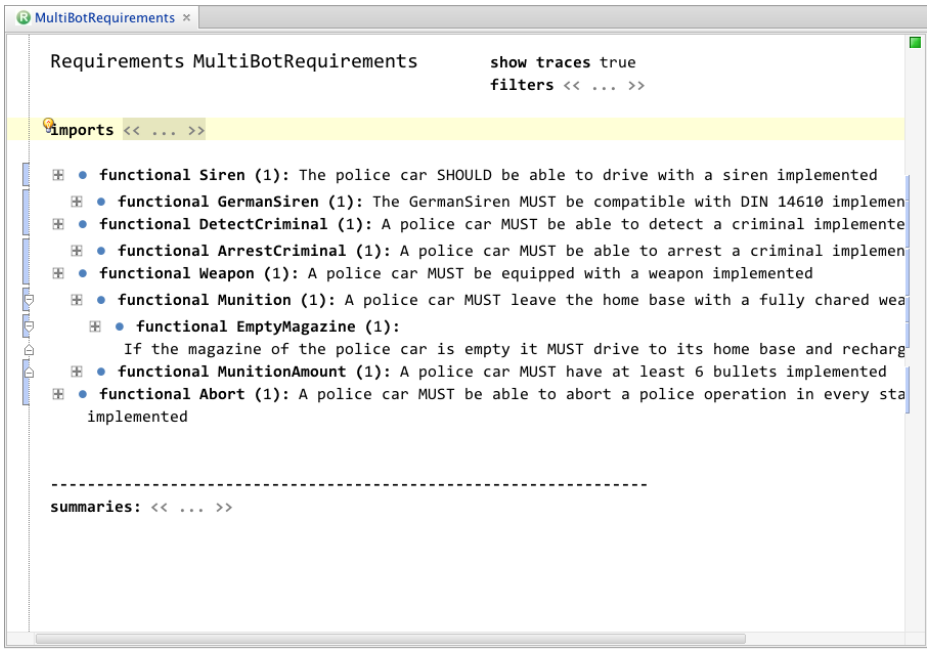
Other convenience components such as the `Orienter` use the drive train underneath and provide a high-level approach to orienting the robot based on the compass sensor. The compass sensor itself requires a non-trivial sequence of operations to retrieve an actual heading. This is encapsulated in the component.

```
void orienter_orientTowards(int16 heading, uint8 speed, DIRECTION dir)
     <- op orienter.orientTowards { int16 currentDir = compass.heading();
  if ( dir == COUNTERCLOCKWISE ) {
    motorLeft.set_speed(-1 * ((int8) speed));
    motorRight.set_speed(((int8) speed));
    while ( !(currentDir in [heading - 4 .. heading + 4]) ) {
      currentDir = compass.heading();
    }
  } else {
    motorLeft.set_speed(((int8) speed));
    motorRight.set_speed(-1 * ((int8) speed));
    while ( !(currentDir in [heading - 4 .. heading + 4]) ) {
      currentDir = compass.heading();
    }
```

```
  }
  motorLeft.stop();
  motorRight.stop();
}
```

State Machines: One of the robots is configured to drive around an obstacle course. It has an ultrasound sensor that can detect obstacles and a a set of bumper sensors at the front and at the back. Whenever the robot detects an obstacle with one of these sensors, it backtracks, turns, and tries again. It uses slightly different strategies depending on which sensors report a (possible) collision. The logic that decides where to go when is implemented as a state machine. The state machine calls out to the above mentioned components to effect the necessary changes in direction or speed.

Requirements Tracing: For demo purposes we have defined a set of requirements for the robot. They are captured with the mbeddr requirements language:



We then use tracing from various implementation artifacts to connect these to requirements:

```
[connect gun.motor to gunMotor.motor ] implements Weapon
[connect gun.util to util.util ] implements Weapon
connect policeCar.pcDriver to policeCarDriver.pcd
connect policeCarDriver.driveTrain to drivetrain.dt
[connect policeCarDriver.gun to gun.gun ] implements Weapon
[connect policeCarDriver.siren to siren.siren ] implements Siren
```

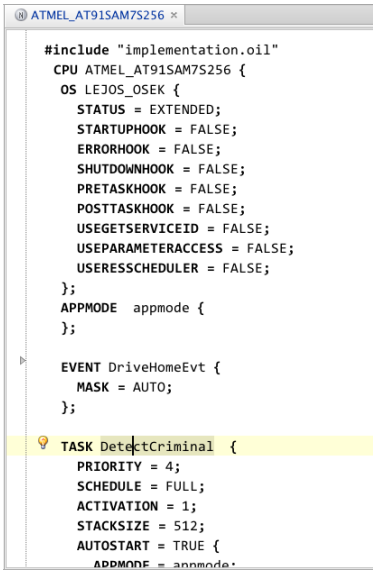
Product Line Variability: Lego robots are particularly useful for building hardware product lines — you can just plug on/off various parts of the robot. In our case, for example, the bumpers and the compass were easily removable. The software of the robot has to take into account this variablity. Also, there is an optional display that is available, so display output must be removed if it is not configured. An example is below.

```
message list PoliceCarDriverMessages {
  {logging}
  [INFO sirenStarted() active: Siren started ] implements Siren
  {logging}
  [INFO sirenStopped() active: Siren stopped ] implements Siren
  {logging}
  [INFO drivingToOperationArea() active: Driving to op area ] implements DetectCriminal
  {logging}
  [INFO criminalDetected() active: Criminal detected ] implements DetectCriminal
  [WARN shoot() active: Shoot! ] implements ArrestCriminal, Weapon
}
exported component PoliceCarDriverImpl extends nothing {
  [requires Siren siren ] implements Siren
  provides PoliceCarDriver pcd
  {withDisplay && logging}
  requires EcRobot_Display display
  requires DriveTrain driveTrain
  [requires Orienter orienter optional ] implements DetectCriminal
  [requires EcRobot_Sonar sonar ] implements DetectCriminal
  [requires Gun gun ] implements Weapon
  [void pcd_startSiren() ⬅ op pcd.startSiren {                    ] implements Siren
    {withDisplay && logging}
    display.showStringAt("siren on", 0, 0);
    {logging}
    report(0) PoliceCarDriverMessages.sirenStarted() on/if;
    siren.on();
  [} runnable pcd_startSiren                                      ]
```

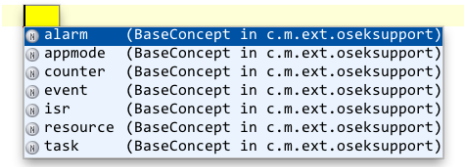
5 How was mbeddr extended?

The extensions developed in this system related to first-class support for OSEK abstractions. These include:

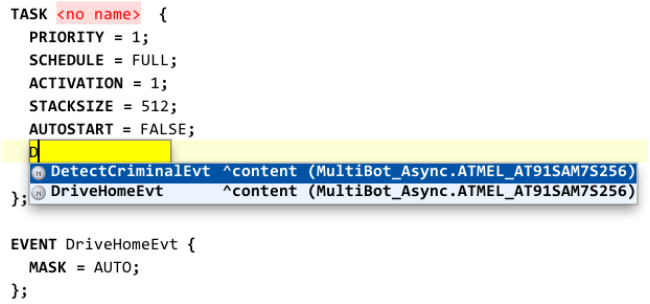
OIL Files: The OSEK operating system has to be configured for each application. This configuration determines various aspects of the OS instance including tasking, scheduling, and memory allocation. To be able to define these OIL files, we have implemented the OIL language in MPS. This is rather trivial, since OIL files are essentially nested name-value pairs. Note however, that OIL files themselves are *not* C extensions, they are a separate, stand-alone ("external") DSL. An example is below.



However, our language implementation does know about the various possible entries and their properties. The code completion menu below shows the various options available in OIL files.



After selecting a `task`, the properties required by a task are pre-initialized; Events referenced by the task can be referenced directly.



C Extensions for Tasks and Events A major reason why the OIL file declares tasks and events is that the operating system instance generated from an OIL file then schedules taks and manages events for the programmer. However,

it is of course necessary to provide *implementations* of tasks (i.e. specify what should happen as the task executes). We have implemented a new top level content `task` that serves this purpose:

```
task (SirenTask) {
  if ( siren.isOn() ) {
    siren.playOnce();
  }
  TerminateTask();
}
```

`SirenTask` in the code above is actually a reference to the task node declared in the OIL file of the particular system. This way it is ensured that you can only define task implementations for tasks declared in the OIL file. Conversely, the system reports a warning in the OIL file if there is no task implementation referring to a particular task declaration in the OIL file.

The OSEK API provides various functions for managing events; the code below uses a few of them.

```
task (Shoot) {
  while ( true ) {
    WaitEvent(ShootEvt);
    ClearEvent(ShootEvt);
    if ( ... ) {
      SetEvent(PoliceCarDriver, SignalHit);
    }
  }
  TerminateTask();
}
```

In OSEK, the arguments passed into these API functions are simply integers. In mbeddr, we have built an extension, `EventMaskType`, which directly acts as a reference to the events declared in the OIL file; pressing `Ctrl-Space` directly shows the available events (`ShootEvent` and `SignalHit` are examples in the code above). This has obvious advantages for program consistency. The code below shows the declaration of these functions.

```
external module kernel resources header : "kernel.h" header : <osek.h>
{
  void TerminateTask();
  void ActivateTask(TaskType task_type);
  void ChainTask(TaskType task_type);
  void ShutdownOS(StatusType status);
  StatusType SignalCounter(CounterType counter);
  StatusType WaitEvent(EventMaskType event);
  StatusType GetEvent(TaskType task, EventMaskType* event);
  StatusType ClearEvent(EventMaskType event);
  StatusType SetEvent(TaskType task, EventMaskType event);
}
```

Note how this external module ”wraps” the header files that define the API provided by OSEK. We redefine the functions using our own `TaskType` and `EventMaskType`. The reason why this works is that these types, when generated to C, are reduced to the same `int` types used by the original API. This way, while we can provide better IDE support and error checking in the IDE, the generated code is still compatible with the original API, without any overhead.

Custom Build Support Lejos-OSEK comes with its own particular flavor of `make` files for building executables. Also, there are some peculiarities about how binaries are configured. For this reason we have built a new kind of platform that plugs into the mbeddr build system:

```
Build Configuration for MutiBot_Ausweicher

Build System:
  lego
    oil file: ATMEL_AT91SAM7S256
    path to ecrobot.mak: /opt/lego/nxtOSEK/ecrobot/

Configuration Items
  << ... >>

Binaries
  rxe executable HelloRXEWorld {
    included modules
      LegoMultiBot (MutiBot_Lego)
      ecrobot_interface (nxtOSEK.api)
      kernel_id (nxtOSEK.api)
      kernel (nxtOSEK.api)
      EcRobotImpl (nxtOSEK.components)
      EcRobotAPI (nxtOSEK.components)
      Messages (multibotAPI.components)
      Gun (multibotAPI.components)
      DriveTrain (multibotAPI.components)
      Orienter (multibotAPI.components)
      Bumper (multibotAPI.components)
      MultibotInstances (MutiBot_Lego)
  }
```

The platform specifies the OIL file to be used for a given system and specifies that path to the build infrastructure provided by Lejos-OSEK. Also, a new generator for `BuildConfigurations` is provided which translates `BuildConfigurations` to valid Lejos-OSEK `make` files.

6 What is your conclusion?

The main purpose of this system is to act as a demo for mbeddr. It serves this purpose well. Developing it helped us uncover and fix all kinds of bugs in the mbeddr system.

However, the development also proved that the idea of mbeddr works: the components and state machines do make the code more readable, more flexible and easier to test. This is achieved without any significant runtime overhead — we were able to run the binaries on the Lego target hardware.

It also showed that project- or platform-specific C extensions are useful and can be built with very limited effort and without changing the C base language itself (the tasks, event types, etc.). OIL files demonstrate the integration

of platform-specific artifacts into the overall system, integrating with type checking, error reporting and IDE support[3].

Finally, the case study shows that custom build infrastructures (`make` file formats, compilers) can be integrated into mbeddr using the hooks provided for just that purpose.

7 Status? Is it in use yet?

We have built several applications for Lego Mindstomrs based on this infrastructure. The code is currently not in the Open Source repository because we do not have the resources to keep it up-to-date as mbeddr changes. However, we expect to open source the code at some point.

[3] This is an example of one of the major ideas of mbeddr: what is often considered *tool* support in classical tools is just simply *language engineering* in mbeddr. The tool itself (MPS) is generic.