Model-Based Testing of Robots with NModel

Juhan Ernits[1], Margus Veanes[2], and Johannes Helander[3]

[1] Inst. of Cybernetics / Dept. of Comp. Sci.,
Tallinn University of Technology, Tallinn, Estonia
juhan@cc.ioc.ee,
[2] Microsoft Research, Redmond, WA, USA
margus@microsoft.com,
[3] Microsoft European Innovation Center, Aachen, Germany
jvh@microsoft.com

Abstract. We present a method of two level modeling for model-based testing of robotics applications. The goal is to perform model-based hardware-in-the-loop testing of a system of robots. The main idea is to use models in two different roles: as control models of individual robots, and as a coordination model on the global level for modeling the behavior of the system. This way it is possible to reduce the total state space visible to the system level model since the local abstractors discard details that are not important from the global perspective. We demonstrate the approach with model programs of the toolkit NModel in the context of LEGO NXT based mobile robots. The system consists of robots sharing a critical resource — an intersection where they must not collide. The local control model distinguishes whether the robot is in the critical section and has the means to pause the robot in an appropriate zone. The coordination model provides the system view and enables to detect system level errors, for example whether it is possible that two robots enter the critical section at the same time. We present the local control models and the coordination models and the full toolchain for model-based testing of the system.

1 Introduction

We present a method for system level model-based testing of mobile robots. The main idea is to use models in two different roles: as control models of individual robots in our case study, and as a system level coordination model for modeling the behavior of the system as a whole. The key to the approach is the need to reduce the total state space visible to the system level model for enabling meaningful tests and the local control models can be viewed as abstractors that discard details that are not important from the system level point of view.

For modeling we use *model programs*. Model programs is a useful formalism for the modeling of software and for design analysis. Model programs are used as the foundation of tools such as Spec Explorer [13] and NModel [4, 8].

We demonstrate the approach with model programs of using the toolkit NModel in the context of LEGO NXT based mobile robots. The system consists of robots sharing a critical resource — an intersection where the robots must not collide. The local control model distinguishes whether the robot is in the critical section, i.e. in the intersection, and has the means to pause the robot in an appropriate waiting zone to force the robots to use their built in program to avoid collision. In addition the local control model can detect if the behavior of a single robot does not conform to the specification.

The coordination model provides the system level view and enables to detect system level errors, for example, whether it is possible that two robots enter the critical section at the same time. We present the local and global coordination models and the full toolchain for model-based testing of the system. The work leads to the need of using various planning techniques in combination with model-based testing technology to make the local control models go into states that are relevant from the perspective of the coordination model.

1.1 Example

Let us first have a closer look at the example that is used for illustrating the proposed testing method. The example is motivated by the well known traffic rule in countries with right hand traffic: ”At an intersection without signs or signals, you should yield the right-of-way to any vehicle approaching from the right.” The environment is made up of an intersection as depicted in Figure 1. The road is divided into four types of zones:

- Safe road zone — the zone where the robot can safely move straight forward.
- Approach zone — the zone which is next to the critical zone.
- Critical zone — the intersection which several robots can enter.
- Turn and calibration zone — special zone where our robots are instructed to turn around and adjust their heading to compensate for odometry errors.

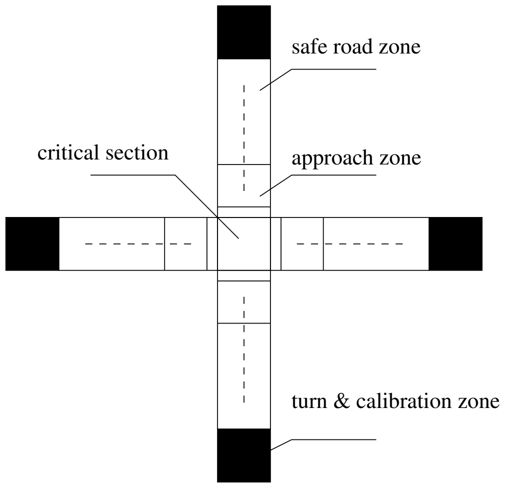
The robots that move in the intersection are in our example differential drive robots built from Lego Mindstorms NXT kits, for example like the robot in Figure 1. Each robot has a *light sensor* to detect the passing of the borders between different zones encoded by lines and solid black areas, and an ultrasonic *range sensor* that is turned $45°$ to the right. When a robot is in the approach zone approaching the intersection then the range sensor is used to detect if there is another robot in the right adjacent approach zone. If the ranger detects an obstacle to the right, it yields, otherwise it drives forward. The robot in Figure 1 also has a compass sensor which can be used as an extra probe for monitoring the heading of the robot.

1.2 Requirements

Unless there are requirements, we cannot test the system. The requirements of the previously described system can be presented in the following way:

Requirements of a single robot

1. When the robot is switched on, it drives forward.
2. When the robot encounters turn and calibration zone, it turns around and continues moving in the opposite direction.



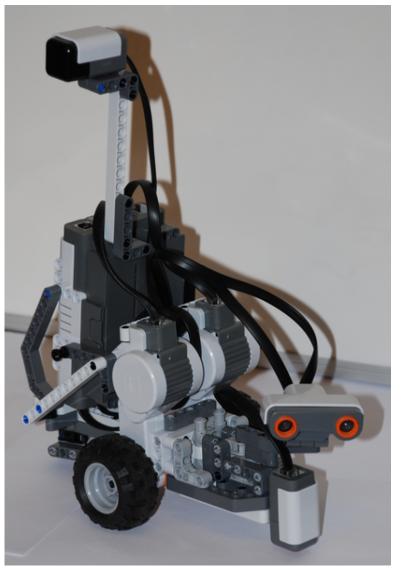


Fig. 1. The intersection that constitutes the environment for the mobile robots *(left)* and an implementation of a mobile robot for the intersection example *(right)*.

3. When the robot moves from the safe road zone to the approach zone, it activates detection of objects to the right to see if there is any robot appoaching from the right.
4. If there is an obstacle detected to the right, the robot pauses. When the obstacle moves away, the robot waits for a safe period of time to allow the robot to cross the intersection and then moves into the critical section unless there is any obstacle to the right.
5. The robot stops detecting obstacles when leaving into the critical section from the approach zone.
6. The robot traverses the zones in the following recurring sequence: safe, approach, critical, approach, safe, turn and calibrate.

Requirements of the system

1. The system must be able to accommodate two robots.
2. Each robot must reach the critical section infinitely often, i.e. the system should not deadlock and none of the robots should get stuck.
3. There should never be more than one robot in the critical section at any point in time.

```
namespace LocalModel
{
public enum Zone {Turning1, Safe1, Approach1, Critical, Approach2, Safe2, Turning2}
public partial class Contract
{
    public static Zone currentZone = Zone.Safe1;
    public static Zone previousZone = Zone.Turning1;
    public static bool stopped = false;
    public static bool obstacleDetected = false;
    #region Actions between the robot and the local model

    [Action]
    public static void ZoneChange(Zone zone)
    { previousZone = currentZone; currentZone = zone; }
    public static bool ZoneChangeEnabled(Zone zone)
    {
        switch (zone)
        {
            case Zone.Safe1:
              return currentZone == Zone.Approach1 && previousZone == Zone.Critical
                 ||currentZone == Zone.Turning1 && previousZone == Zone.Approach1;
            case Zone.Safe2:
              return currentZone == Zone.Approach2 && previousZone == Zone.Critical
                 ||currentZone == Zone.Turning2 && previousZone == Zone.Approach2;
            case Zone.Approach1:
              return currentZone == Zone.Critical && previousZone == Zone.Approach2
                 ||currentZone == Zone.Safe1 && previousZone == Zone.Turning1;
            case Zone.Approach2:
              return currentZone == Zone.Critical && previousZone == Zone.Approach1
                 ||currentZone == Zone.Safe2 && previousZone == Zone.Turning2;
            case Zone.Critical:
              return currentZone == Zone.Approach1 && previousZone == Zone.Safe1
                 ||currentZone == Zone.Approach2 && previousZone == Zone.Safe2;
            case Zone.Turning1:
              return currentZone == Zone.Safe1 && previousZone == Zone.Approach1;
            case Zone.Turning2:
              return currentZone == Zone.Safe2 && previousZone == Zone.Approach2;
            default:
              return false;
        }
    }

    [Action]
    public static void Start()
    { stopped = false; }
    public static bool StartEnabled()
    { return !paused && stopped && obstacleDetected; }

    [Action]
    public static void Stop()
    { stopped = true; }
    public static bool StopEnabled()
    { return !stopped && currentZone != Zone.Critical; }

    [Action]
    public static void ObstacleInSight(bool obstacle)
    { obstacleDetected = obstacle; }
    public static bool ObstacleInSightEnabled()
    {
        return currentZone == Zone.Approach1 && previousZone == Zone.Safe1 ||
               currentZone == Zone.Approach2 && previousZone == Zone.Safe2;
    }
    #endregion
}
}
```

Fig. 2. Local model of a robot.

```
namespace LocalModel
{
public partial class Contract
{
    #region Actions between the coordination and the local model

    public static Zone targetZone = Zone.Safe1;
    public static bool gotoInProgress = false;
    public static bool paused = false;

    [Action]
    public static void GoTo_Start(Zone zone)
    { targetZone = zone; gotoInProgress = true; }
    public static bool GoTo_StartEnabled(Zone zone)
    { return targetZone != Zone.Critical; }

    [Action]
    public static void Goto_Finish(Zone zone)
    { stopped = true; paused = true; gotoInProgress = false; }
    public static bool Goto_FinishEnabled(Zone zone)
    { return gotoInProgress && currentZone == targetZone && zone == targetZone; }

    [Action]
    public static void Continue()
    { paused = false; }
    public static bool ContinueEnabled()
    { return paused; }

    #endregion
}
}
```

Fig. 3. Actions on the port between the local models of robots and the coordination model.

2 Modeling

We model the system on two levels: the behavior of the robot is modeled in the local control model and the global behavior is modeled in the coordination model.

Deriving a formal model from the requirements is one of the most challengeing tasks of model-based testing. Model programs provide a new alternative for writing models using C# where the modeling formalism is disguised into the API of the NModel library. We model the example using such model programs.

Each robot in the system is connected to a local control model via a system adapter. All local control models are connected to the coordination model via a similar mechanism.

2.1 Local Control Model

The local control model of a robot is given in Figures 2 and 3. The actions of the model are split into two parts, the actions between the model and the robot are in Figure 2 and the actions between the local model and the coordination model in Figure 3.

The zones of the intersection are called `Turning1, Safe1, Approach1, Critical, Approach2, Safe2, Turning2` to distinguish between the two distinct but symmetrical areas next to the critical zone. To model the motion of robots, the model uses two state variables `currentZone` and `previousZone`. This ensures that when the robot is for example in the `Critical` zone, it does not go back to the same approach zone it entered the critical zone from. Such details is important only for defining valid behavior of a single robot and will be abstracted on the level of the coordination model.

2.2 Global Coordination Model

In Figure 4 there is a global coordination model of the system. On the level of the global coordination model we do not need to distinguish the different safe and turning states any more. Thus the zones used in the coordination model are calles `AbstractZones`. The mapping from the conrete zones in the local models is made in the system adapter between the local and the coordination model.

`Initialize` is an initialization action that adds robots `A` and `B` to the map which maps robots to zones they are known to be in. Initially both robots are in the `Safe`. It should be noted that such a model is not limited to an intersection of just two roads, more can be added by defining more robots in the enumeration `Robot` and adding them to the `currentState` map.

The coordination model has an action `GoTo_Start` which triggers a command to a particular robot to go to the given zone and pause there. The action `GoTo_Finish` is triggered by the concrete robot when it arrives in the zone defined in `GoTo_Start`.

The action `Continue` is meant to restart robots that have completed the GoTo instruction and are waiting for further instructions.

In Figure 5 there is a state invariant that says that a state of the global coordination model is invalid if there is more than one robot in the critical section at any one time. Such a property is checked at runtime of on-line testing of the system and violations are reported as errors.

3 Implementation

The implementation of the system is built using robots controlled by Lego NXT bricks. The NXT bricks are in our example controlled by software built using the Lego Mindstorms NXT kit software. Communication with the model occurs over a serial port emulated by a Bluetooth connection. The code in the NXT brick sends messages to the model in accordance with the actions taken (start, stop, move) and actions detected (moving from one zone to another). Each local control model of a robot has two ports: the port connecting the model to the robot, and the port connecting the model to the coordination model. Each of the ports supports a different set of actions: the port between the local control model and the robot supports `ZoneChanged()`, `Start()`, `Stop()`, and `ObstacleInSight()`, actions while the port between the local control model and the global model supports `GoTo()`, `Continue`, and `ZoneChanged()` actions. The global model is connected to the local control models via the latter ports. The connections are implemented in the system adapters called *steppers* in the NModel toolkit.

```
namespace GlobalModel
{
    public enum Robot { A, B }
    public enum AbstractZone { Safe, Approach, Critical }
    public enum ControlMode { Initializing, Running }
    public class Contract
    {
        public static Map<Robot, AbstractZone> currentState =
            Map<Robot, AbstractZone>.EmptyMap;

        public static Map<Robot, AbstractZone> targetState =
            Map<Robot, AbstractZone>.EmptyMap;

        public static Set<Robot> ready = Set<Robot>.EmptySet;

        public static ControlMode controlMode = ControlMode.Initializing;

        [Action]
        public static void Initialize()
        {
            currentState = currentState.Add(Robot.A, AbstractZone.Safe);
            currentState = currentState.Add(Robot.B, AbstractZone.Safe);
            controlMode = ControlMode.Running;
        }
        public static bool InitializeEnabled()
        { return controlMode == ControlMode.Initializing; }

        [Action]
        public static void GoTo_Start(Robot r, AbstractZone z)
        { targetState = targetState.Add(r, z); }
        public static bool GoTo_StartEnabled(Robot r)
        { return controlMode > ControlMode.Initializing
            && !targetState.ContainsKey(r); }

        [Action]
        public static void GoTo_Finish(Robot r, AbstractZone z)
        {
            targetState = targetState.RemoveKey(r);
            ready = ready.Add(r);
        }
        public static bool GoTo_FinishEnabled(Robot r, AbstractZone z)
        {
            return controlMode > ControlMode.Initializing &&
              targetState.ContainsKey(r) && targetState[r] == z;
        }

        [Action]
        public static void Continue(Robot r)
        { ready = ready.Remove(r); }
        public static bool ContinueEnabled(Robot r)
        { return controlMode > ControlMode.Initializing && ready.Contains(r); }

        [Action]
        public static void ZoneChanged(Robot r, AbstractZone z)
        {
            currentState = currentState.Override(r, z);
        }
        public static bool ZoneChangedEnabled()
        { return controlMode > ControlMode.Initializing; }
    }
}
```

Fig. 4. The coordination model of a system of two robots.

```
[StateInvariant]
public static bool noCollision()
{
    int robotsInCriticalSection = 0;
    foreach (Pair<Robot,AbstractZone> pair in currentState)
    { if (pair.Second == AbstractZone.Critical)
        robotsInCriticalSection ++;
    }
    return robotsInCriticalSection < 2;
}
```

Fig. 5. A global invariant that should not be violated under valid behavior.

4 Model-Based Testing of the System

We can use the local control model in conjunction with a single robot to test whether the software in the robot is able to correctly detect zone changes and perfrom turns. To perform system level test, we need to incorporate the global model into the system and test if the system of robots conforms to the specification of the global behavior formalized by the global model. The coordination model forces individual robots into positions where all possible reachable combinations of robots and zones have been checked. In particular, it forces one robot to pause in the critical section to test if the other robot correctly detects that there is a robot to the right.

In the case testing does not reveal any errors it is possible to conclude that the software seems to behave correctly under the conditions that were available during testing. The more interesting case is when errors are detected. The errors can be local errors, for example, the robot cannot reliably detect moving into a different zone. It is usually more complicated to test for system level errors that involve interaction between several active components of the system. Such an error is failure to detect an obstacle or failure to stop in the approach zone upon detection of the obstace.

An interesting side effect of such testing is that we detect behaviors which we did not design but which occur due to interaction of different requirements. An example of such behavior in our case study is that the robot stops also when it sees another robot leaving the intersection.

One of the main benefits of two level modeling is that the local control model performs abstractions that reduce the amount of information that needs to be communicated to the global model. For example, communicating odometry data to the global level would use up quite considerable amount of communication bandwidth and reduce the clarity of the global model. In addition such latencies may require to slow the robots down to make them cope with the extra burden.

In addition to random walk or Chinese postman tour in the global model we can introduce scenarios that are of interest from the global perspective. For example, one of the most relevant tests for the obstacle avoidance code is to test the behavior with and without obstacle, i.e. the other robot on the right. NModel allows to specify such scenarios as simple finite state machines.

5 Related Work

In [3] the use of UML collaboration diagrams is proposed for specifying offered and provided protocol contracts. The paper does not present the details how tests are derived based on the specified contracts and how the UML collaboration diagrams translate to proposed JContract specifications.

[10] presents an approach of model based testing of embedded systems. The models are presented as Charon models which are hierarchical hybrid automata. The target systems are Aibo robots but the approach focuses on a single robot.

In distributed test architectures [7] there may be multiple testers active simultaneously. In the general case this gives rise to the problem of controllability (making sure that the test steps received by the separate testers are received in the correct order) and the problem of observability (determining what input caused what output) [1].

Various coordinated test methods [5, 6, 9, 11] have been designed that use special coordination messages to address these problems.

In the current paper we propose a model-based approach for testing distributed services that avoids the controllability and observability problems by using a two-layered modeling approach. A global coordination model is used drive the testing process and to check for conformance violations.

Local models are used to specify the behavior of the individual services and to drive them forward. Typically, the global model is at a higher abstraction level and is concerned with the interaction between the services, whereas the local models provide a more detailed view of the behavior of the individual services and have no or only limited knowledge of the global constraints.

Local models may synchronize through shared actions. Observable actions or events are recorded in separate event logs. There is one event log per service. Collectively, the logs define a partial (causal) order on events. The event logs are multiplexed [2] to provide a valid serial trace that is checked against the global model.

6 Summary

We presented a small example that illustrates how model programs can be used for two level modeling and both system and component level testing of an application involving mobile robots. In this example the local control models are simple in the sense that there is no need to do any planning to reach to the critical section again. In more realistic examples driving the local control model into a state desired from the global perspective may include some planning. To solve this problem, we have two possible solutions in mind: either to present the model as a constraint system and solve using a contstraint solver or to use a reactive planning tester approach presented in [12].

Acknowledgements

Juhan Ernits was partially supported by the ITEA-2 D-MINT project and the Estonian Science Foundation grant No. 7667.

References

1. L. Cacciari and O. Rafiq. Controllability and observability in distributed testing. *Information & Software Technology*, 41(11-12):767–780, 1999.
2. C. Campbell, M. Veanes, J. Huo, and A. Petrenko. Multiplexing of partially ordered events. In F. Khendek and R. Dssouli, editors, *17th IFIP International Conference on Testing of Communicating Systems, TestCom 2005*, volume 3502 of *LNCS*, pages 97–110. Springer, 2005. (Best Paper award).
3. R. Heckel and M. Lohmann. Towards contract-based testing of web services. *Electr. Notes Theor. Comput. Sci.*, 116:145–156, 2005.
4. J. Jacky, C. Campbell, M. Veanes, and W. Schulte. *Model-based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
5. C. Jard, T. Jéron, H. Kahlouche, and C. Viho. Towards automatic distribution of testers for distributed conformance testing. In *FORTE XI / PSTV XVIII '98: Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 353–368, Deventer, The Netherlands, 1998. Kluwer, B.V.
6. A. Khoumsi. A temporal approach for testing distributed systems. *IEEE Trans. Softw. Eng.*, 28(11):1085–1103, 2002.
7. G. Luo, R. Dssouli, G. V. Bochmann, P. Venkataram, and A. Ghedamsi. Test generation with respect to distributed interfaces. *Comput. Stand. Interfaces*, 16(2):119–132, 1994.
8. NModel. NModel web site, 2007. http://www.codeplex.com/NModel.
9. O. Rafiq and L. Cacciari. Coordination algorithm for distributed testing. *J. Supercomput.*, 24(2):203–211, 2003.
10. L. Tan, J. Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In D. Zhang, É. Grégoire, and D. DeGroot, editors, *IRI*, pages 487–492. IEEE Systems, Man, and Cybernetics Society, 2004.
11. H. Ural and D. Whittier. Distributed testing without encountering controllability and observability problems. *Inf. Process. Lett.*, 88(3):133–141, 2003.
12. J. Vain, K. Raiend, A. Kull, and J. P. Ernits. Synthesis of test purpose directed reactive planning tester for nondeterministic systems. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 363–372, New York, NY, USA, 2007. ACM.
13. M. Veanes, C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, and N. Tillmann. Model-based testing of object-oriented reactive systems with Spec Explorer, 2005. Tech. Rep. MSR-TR-2005-59, Microsoft Research. Preliminary version of a book chapter in the forthcoming text book *Formal Methods and Testing*.