Modelling Robot Architectures for Modular Robotics Systems

Gerard T McKee, J. Andrew Fryer
Department of Computer Science
The University of Reading
Reading RG6 6AY
Berkshire, UK
Gerard.McKee@reading.ac.uk

Paul S Schenker
Jet Proplusion Laboratory
California Institute of Technology
4800 Oak Grove Drive/MS 125-224
Pasadena, California 91109-8099, USA
Paul.S.Schenker@jpl.nasa.gov

Abstract

Modularity in software and hardware offers a number of benefits to robotics systems, including high-level design focus and run-time reconfiguration. We have been developing the MARS *model for modelling and reasoning about modular robot systems. In this paper we describe the facilities within the* MARS *model for modelling the data and control flows required to support a range of robot control architectures. We discuss the role of the model in high-level design and we use Brooks' subsumption architecture to illustrate the application of the model to a decentralised, reactive control architecture.*

1 Introduction

Modularity offers significant benefits to robot system designers in the form of unit modules that can be combined in varied patterns to deliver desired functionality. Identifying an appropriate set of modules, ones that can be successfully reused in a range of configurations, is a key challenge of the modular approach. This has been addressed most successfully, recently, in the area of physical modules for reconfigurable modular manipulators and more complex robotics structures [1-3]. The modular approach has also been applied to robotics software, and has included both component-based approaches and object-oriented approaches to software encapsulation [4-7].

It appears obvious that modularity can support top-down design of modular robotics systems and their control architectures. However, can one go beyond the simple transcription of existing architectures to a module based approach? Can one incorporate some level of automatic reasoning into the design process? If so, precisely what is possible and how is it to be achieved? One of the advantages of the modular approach would be the possibility to focus, during design, on modules and their configurations, rather than the detailed implementation of the modules. In order to achieve this flexibility the functionality of the resources must be described in a manner which allows reasoning to be performed about their suitability for a particular task. There must exist a means of expressing the organisation of sets of resources, a means to reason about consequences which will arise when resources interact, and rules for the resolution of these consequences.

The *MARS* model provides mechanisms for modelling robotic resources and methods for reasoning about the functionality which result from their combination. *MARS* models the physical, data and control aspects of modular robot systems. In this paper we describe the second two of these aspects, particularly as they impact on the creation of decentralised control architectures. Previous reports can be found in [8,9]. The remainder of the paper is structured as follows. Section 2 and 3 describe our method for modelling resources and for modelling data and control relationships between resources. Section 4 describes module interaction consequences represented in *MARS* and their resolution. Section 5 describes the application of the model to Brooks' subsumption architecture [10]. Section 6 provides a summary and conclusions.

2 Modelling Resources

Resources, the functional subsystems and components of a robot, may be specified and implemented in various ways. Modularity requires standardised components, described consistently by reference to a general model. For this purpose, *MARS* defines the *module*, a general device with common properties and capabilities. A module is a model of a robotic resource. *MARS* divides modules into the two broad categories, *physical* and *non-physical*. Physical modules have a physical presence and/or action within the robot's environment. They include sensors and effectors such as cameras and manipulators respectively. Non-physical modules exist within a computer environment. They include, for example, image processing and path planning algorithms. *MARS* uses *annotations,* descriptions embedded as comments in the source code implementing the resource, to describe the characteristics of modules. The following two subsections describe how physical (sensor and effectors) and non-physical (algorithm) modules are annotated in the *MARS* model.

2.1 Sensor and Effector Modules

Physical modules possess a number of properties, including *location* and *motion capability*. In *MARS*, module locations are represented using reference frames that are rigidly attached to a point on the physical module. Effectors are devices capable of effecting motion in the environment. Three rotation annotations, Rotn_X, Rotn_Y, and Rotn_Z, describe rotation about the x, y and z axes respectively, and three translations annotations, Trans_X, Trans_Y and Trans_Z, describe translation along the same three axes [9]. Sensors are those elements of the robot which extract data from the environment. Sensors, similar to effectors, have an attached reference frame. A sensor, however, is considered to have no ability to change its position or orientation. It derives this capability from effectors on which it is mounted. Sensors deliver data in a specified format to at most one client. The @Sense annotation is used to describe the sensing functionality of sensors. Its general form is:

```
@Sense <delivery modes> [<frequency>]
```

The delivery modes are specified as a comma delimited list. The delivery modes include NotifyWhenReady (the sensor delivers the data as it becomes available), NotifyAtFrequency (the sensor delivers the data at a specified <frequency>), NotifyOnMax, NotifyOnMin, NotifyOnValue, and NotifyOnChange (the sensor delivers the data only when the specified condition(s) are true), and RequestWhenReady (the client will request the data from the sensor when required). For example, a camera module implemented in C++, and which delivers image data at 25 frames per second, is annotated as follows:

```
// @module_start
// @module_name Camera
// @Sense NotifyAtFrequency 25
ImageFormat  grabimage (void);
// @module_end
```

Annotations are distinguished from general comments with the symbol '@'. The module_start, module_name and module_end directives are used to package the software as *MARS* modules. These will be assumed in what follows.

2.2 Algorithm Modules

Algorithm modules represent the control and data processing algorithms which exist within a robotic system. Algorithm modules that output data are classified as DataOutAlgorithms. Those which output control are classified as ControlOutAlgorithms. Algorithm functionality is characterised by the @DataOut or @ControlOut annotations. The @DataOut annotation takes the following format:

```
@DataOut  <delivery modes> [<frequency>]
```

A SonarMap algorithm module, for example, takes raw sonar data as input and outputs a data representation of the sonar data as a 'map'. It may be annotated as follows:

```
// @DataOut  RequestWhenReady
SonarMap mapgenerator (SonarData);
```

The @ControlOut annotation is used to identify the function which causes a control message to be output. It takes no parameters. An OperatorJoystick module, for example, outputs control signals depending on the position of the joystick. It may be modelled as a ControlOutAlgorithm:

```
// @ControlOut
MotionData polljoystick(void);
```

Control signals currently supported in *MARS* include the six motion functions identified above and grasp functions as per a two-fingered gripper [9].

3 Modelling Robot Systems

Modular robots are organised sets of modules structured according to a physical and control architecture. This organisation is modelled in *MARS* by *relationships*. The *MARS* model identifies three types of relationship. These are illustrated by the simple joystick-controlled mobile camera system shown in Figure 1. Firstly, there are physical connections between the mobile platform and the pan-tilt head. These are modelled by *physical* relationships. Secondly, operator joysticks control the mobile platform and the pan-tilt-head. These are modelled by *control* relationships. Finally, the operator's screen displays image data received from the camera. This is modelled by a *data* relationship.

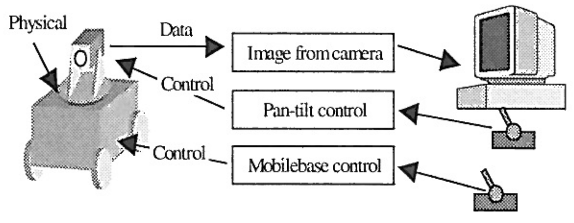


Figure 1. Joystick control of mobile camera

3.1 Physical relationships

In the *MARS* model, when one module is physically connected to another, they are considered to participate in the relationship is_mounted_on. For example, a Camera module mounted on a PanTiltHead module is represented by the relationship:

```
Camera is_mounted_on PanTiltHead
```

The is_mounted_on relationship is parameterised by the geometric transformation which maps the vertex of the mounting module to a vertex of the mounted module. The general format of the relationship is:

```
<module> <vertex> is_mounted_on <module> <vertex>
<transformation>
```

For example, the relationship

```
ModuleA 0 is_mounted_on ModuleB 0 (z, 180, x, -90, 150,_,_)
```

states that vertex 0 of ModuleA is mapped to vertex 0 of ModuleB by first a rotation of 180^o about the z-axis, followed by a rotation of -90^o about the x-axis, and finally a translation of 150mm along the x-axis. Physical relationships are described more fully in [9].

3.2 Control Relationships

In the *MARS* model, the is_controlled_by relationship describes the control structure between two modules: one module is exerting control, the other is accepting control. The general format of the is_controlled_by relationship is:

```
<module> is_controlled_by <module> [<priority>]
```

The optional parameter <priority> can be used to specify that messages from the controlling module have a certain priority relative to messages from other modules. The lower the number, the lower the priority of the control message; zero defines the lowest priority. For example, the following sequence of relationships states that ModuleA accepts control messages from ModuleB, ModuleC and ModuleD with ascending priority:

```
ModuleA is_controlled_by ModuleB 0
ModuleA is_controlled_by ModuleC 1
ModuleA is_controlled_by ModuleD 2
```

An equivalent shorthand notation is:

```
ModuleA is_controlled_by ModuleB, ModuleC, ModuleD
```

MARS identifies two types of control architecture, *flat* and *hierarchical*. Priorities are used only by hierarchical control architectures. If a flat architecture is specified (see below) then priorities will be ignored.

3.3 Data relationships

Data flows are modelled using the uses_data_from relationship. Its general form is:

```
<module> uses_data_from <module> [<delivery mode>]
[<frequency>]
```

where <delivery mode> indicates the delivery mode required and <frequency> is an optional parameter specifying the frequency of the service required. For example, the relationship definition:

```
OperatorScreen uses_data_from Camera NotifyAtFrequency 10
```

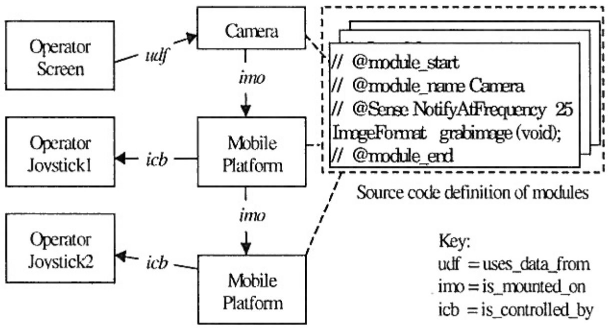
states that the OperatorScreen module requires the NotifyAtFrequency service from the Camera module at a data rate of 10Hz (i.e. image data should be delivered at 10 frames per second).

3.4 Configuration Definitions

A modular robot is described in *MARS* by a *configuration definition*. A configuration definition is identified by the @configuration keyword, which can take an optional parameter specifying the type of the architecture, flat (default) or hierarchical. It is followed by a declaration of the modules involved in the configuration, identified with the @modules keyword, followed by a list of relationship definitions of the form described above. Figure 2, for example, shows the configuration for the joystick controlled mobile camera of Figure 1. Figure 3 shows the same in diagrammatic form and illustrates its relation to source code implementation of the modules.

```
@configuration
@modules Camera, PanTiltHead, MobilePlatform,
OperatorJoystick1, OperatorJoystick2, OperatorScreen
Camera         is_mounted_on  PanTiltHead
PanTiltHead    is_mounted_on  MobilePlatform (_,_,_,_,_,100)
PanTiltHead    is_controlled_by OperatorJoystick1
MobilePlatform is_controlled_by OperatorJoystick2
OperatorScreen uses_data_from Camera
```

Figure 2. Configuration for mobile camera (Figure 1)



```
// @module_start
// @module_name Camera
// @Sense NotifyAtFrequency 25
ImageFormat grabimage (void);
// @module_end
```

Figure 3. Relationships for the mobile camera

4 Reasoning About Configurations

MARS identifies two stages in the realisation of a modular robot, an *analysis* stage which aims to identify the consequences of the particular modular robot configuration, and a *synthesis* stage which resolves these consequences through the introduction of additional modules, delivering in turn a robot *specification*. *MARS* identifies physical, data and control *consequences* in module interactions.

4.1 Physical Consequences

Physical consequences include simple and similar inherited functions, emergent functionality, module conflict and global functionality. These are described in [9]. In this paper we will restrict ourselves to *simple* inheritance. In simple inheritance a sensor or end-effector inherits each of its degrees of freedom from no more than one of the modules on which it is mounted. Physical consequences are resolved using InheritanceNodes [9].

4.2 Data Consequences

The data consequences recognised by the *MARS* model include *divergence* consequences, dependency of *service* consequences, and dependency of *representation* consequences. *Divergence* consequences occur where a data source is required to deliver data to two or more target modules. Divergence consequences are resolved by splitting the data into multiple streams using a DataSplitterNode, which belongs to the class of DataOutAlgorithms. On receipt of a message from the data source, the DataSplitterNode copies the message to each of a set of target modules. The DataSplitterNode is introduced into a specification as a new module. For example, the configuration in Figure 4 has a divergence consequence for the Camera data source. The resultant specification in Figure 5 (illustrated in Figure 6), incorporates a DataSplitterNode between source and destination modules.

```
@configuration
@modules Camera, OperatorScreen, HoughTransform
OperatorScreen     uses_data_from     Camera
HoughTransform     uses_data_from     Camera
```

Figure 4. Data divergence consequence

```
@specification
@modules Camera, OperatorScreen, HoughTransform,
DataSplitter1
DataSplitter1      uses_data_from     Camera
OperatorScreen     uses_data_from     DataSplitter1
HoughTransform     uses_data_from     DataSplitter1
```

Figure 5. Resolution of divergence consequence

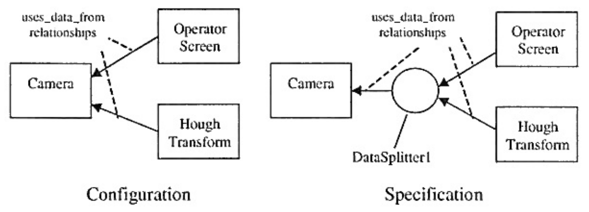


Figure 6. DataSplitterNode: modules & dependencies

Dependency of service consequences occur when one module requires a certain delivery service from the data source. These are also resolved with DataSplitterNodes, since these can take any of the uses_data_from delivery mode parameters. Dependencies of service may also occur with a divergence consequence.

Dependency of representation consequences occur when a module requires data in a format not supported by the data source. These representation consequences are resolved using DataTransformationNodes, belonging to the class of DataOutAlgorithms. If the required data representation is not supplied by a module, a DataTransformationNode is introduced to convert between the representations. Figure 7, for example, contains a dependency of representation consequence. The output representation of Camera is raw image data, whereas the ObjectAnalysis module requires LineDescriptions. These data requirements are identified from the respective module definitions. Figures 8 and 9 demonstrate the consequence's resolution.

```
@configuration
@modules Camera, ObjectAnalysis
ObjectAnalysis      uses_data_from     Camera
```

Figure 7. A data dependency consequence

```
@specification
@modules Camera, ObjectAnalysis, DataTransformer1
DataTransformer1    uses_data_from     Camera
ObjectAnalysis      uses_data_from     DataTransformer1
```

Figure 8. Resolution through data transformation

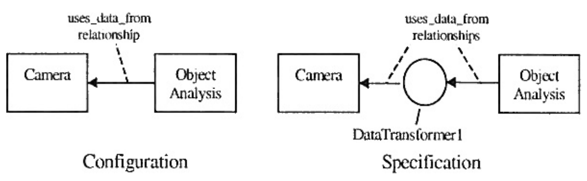


Figure 9. DataTransformationNode introduction

4.3 Control Consequences

Two types of control consequence are identified in the *MARS* model, *convergence* and *divergence*. A *convergence* consequence arises when two or more modules control a single module. Since modules can only accept one control message at a time, some selection for the controlling module must be performed. Convergence of control consequences are resolved using ArbitrationNodes, which belong to the class of ControlOutAlgorithms. The *resolution strategy* they employ to select between the input messages depends on the architecture selected for the configuration. *MARS* currently defines the behaviour of the ArbitrationNode for a *flat* control architecture to involve averaging the received control messages over a time period. The resultant is then sent as a message to the controlled module. Figure 10 shows a configuration for a modular robot containing a convergence of control consequence. Figures 11 and 12 show how an ArbitrationNode resolves the consequence.

```
@configuration
@modulesMobilePlatform, WanderBehaviour,
      AvoidObstacleBehaviour
MobilePlatform  is_controlled_by   WanderBehaviour
MobilePlatform  is_controlled_by   AvoidObstacleBehaviour
```

Figure 10. Convergence control consequence

```
@specification flat
@modules MobilePlatform, WanderBehaviour,
      AvoidObstacleBehaviour, ArbitrationNode
MobilePlatform   is_controlled_by   ArbitrationNode
ArbitrationNode is_controlled_by   WanderBehaviour
ArbitrationNode is_controlled_by   AvoidObstacleBehaviour
```

Figure 11. ArbitrationNode introduction

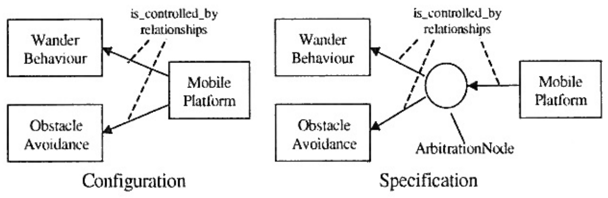


Figure 12. ArbitrationNodes: modules & dependencies

In a *hierarchical* control architecture priorities are assigned to each controlling module. The ArbitrationNode receives messages from all the controlling modules, and selects the one with highest priority. Figure 13 shows an ArbitrationNode introduced to resolve the convergence of control consequence in a hierarchical control architecture. Figure 14 shows the corresponding dependencies and control flows for the architecture.

```
@specification hierarchical
@modules MobilePlatform, WanderBehaviour,
AvoidObstacleBehaviour, ArbitrationNode
MobilePlatform   is_controlled_by   ArbitrationNode
ArbitrationNode is_controlled_by   WanderBehaviour 0
ArbitrationNode is_controlled_by   AvoidObstacleBehaviour 1
```

Figure 13. Convergence control resolution

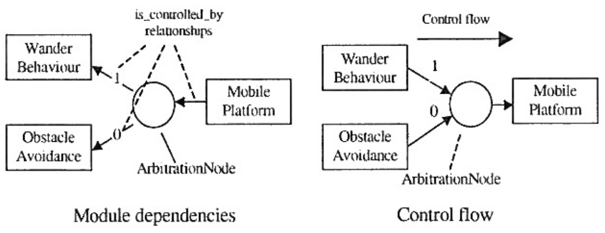


Figure 14. Convergence control resolution

A *divergence of control* consequence arises when two or more modules are controlled by a single module. It is resolved by ControlSplitterNodes, which belong to the class of ControlOutAlgorithms. ControlSplitterNodes duplicate their input messages onto a number of output messages. The configuration in Figure 15, for example, has a divergence of control consequence. Figure 16 shows a ControlSplitterNode used to resolve this consequence.

```
@configuration
@modules Wander, MobilePlatform1, MobilePlatform2
MobilePlatform1     is_controlled_by     Wander
MobilePlatform2     is_controlled_by     Wander
```

Figure 15. Divergence control consequence

```
@specification
@modules Wander, MobilePlatform1, MobilePlatform2,
    ControlSplitter1
ControlSplitter1    is_controlled_by     Wander
MobilePlatform1     is_controlled_by     ControlSplitter1
MobilePlatform2     is_controlled_by     ControlSplitter1
```

Figure 16. ControlSplitterNode introduction

4.4 Discussion

The reasoning model described above involves the incorporation of additional modules, referred to as nodes. We will briefly highlight the benefits of these in the context of high-level design and reconfiguration. The two splitter nodes (DataSplitterNode and ControlSplitterNode) and the data transformation node focus on the glue logic that merges systems into functional units. *MARS* assumes each module delivers data to at most one other module. This may appear restrictive. However, if this were not the case then the key sensor and control modules would need to incorporate sophisticated support for multiple clients and their data requirements, introducing significiant programming and performance overheads. By decoupling the data splitter and transformation functions we remove these issues from the concern of the designer. Indeed, we provide opportunities, for example, to distribute data splitting and transformation functions onto different platforms in order to maintain desired levels of service to the client modules. The arbitration nodes, in contrast, speak to the high-level design process directly, offering support for a number of arbitration strategies. The key choice point for the designer in the current version of *MARS* is in the type of architecture to be realised. However, we see considerable scope for the introduction of a range of arbitration strategies.

5 Case Study Application of the Model

The subsumption architecture (SA) [10] is an interesting control architecture to demonstrate in *MARS* as it makes use of low-level functional modules which interact to realise behaviours. The SA is based around increasingly specialised levels of competence built from 'modules' - finite state machines which send messages over connections between one another. Input to modules can be *suppressed* (and replaced with another signal) by the output of other modules. Similarly, output from modules can be *inhibited*. Suppression and inhibition occur for a given time period. Using the suppression and inhibition mechanisms, levels are capable of *subsuming* the behaviour of the levels below them.

Figure 17 shows Brooks' Level 0 control system, an 'avoid obstacles' behaviour - the robot moves away from approaching obstacles, or halts to avoid collisions. The control system is realised as follows. Data from a sonar sensor is passed to a Sonar module, which generates a map, centred on the robot, containing the location of obstacles. The map is monitored by the Feelforce module which generates a repulsive force for each detected obstacle and outputs the sum of these as a single resultant force. The Runaway module transforms this force into Turn and Forward commands which are passed on to the Turn module. The Turn module executes the turn, and on completion passes the forward heading onto Forward. Turn then enters a wait state. Forward moves the robot forward but halts if it receives an input during the motion. Once the motion is complete, the Turn module is reset. The Collide behaviour detects obstacles immediately ahead of the robot. If a collision is imminent, the robot halts, then turns away from the obstacle.

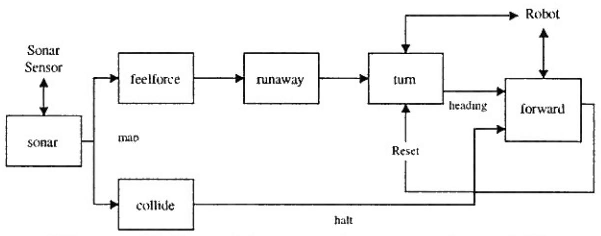


Figure 17. Level 0 control system (from [10])

It is important to note first that a direct translation of this system to *MARS* is not possible. The SA relies on the reset line, which *MARS* lacks, and supports only over-riding of control messages, not of data messages. Subsumption-style control can be demonstrated in *MARS*, however, utilising *hierarchical* control. Figure 18 shows a configuration which implements the level 0 control system in *MARS*. In this architecture, priorities are assigned to resolve control conflicts. The Sonar module is a sensor module which produces sonar data of type SonarData (in NotifyWhenReady mode). SonarMap is a DataOutAlgorithm which produces a sonar map representation – a list of polar coordinates of detected obstacles. Feelforce, a DataOutAlgorithm, produces a potential-field type repulsive force, calculated from the SonarMap data. Runaway is a DataOutAlgorithm which acts as a threshold – data from the SonarMap is output only if it is above a certain value. Turn is a ControlOutAlgorithm which converts an input force into a Rotn_Y motion. Forward is a ControlOutAlgorithm which converts an input force into a Trans_Z control. The Collide module is a ControlOutAlgorithm which monitors for objects immediately in front of the robot. If any are detected, a 'Trans_Z 0' message ('stop') is sent to the Forward module. MobilePlatform represents the robot.

```
@configuration hierarchical
@modules Sonar, SonarMap, MobilePlatform, FeelForce,
   Collide, Runaway, Turn, Forward
Sonar 0           is_mounted_on      MobilePlatform 0
SonarMap          uses_data_from     Sonar
FeelForce         uses_data_from     SonarMap
Collide           uses_data_from     SonarMap
Runaway           uses_data_from     FeelForce
Turn              uses_data_from     Runaway
Forward           uses_data_from     Runaway
MobilePlatform    is_controlled_by   Forward 0
MobilePlatform    is_controlled_by   Turn 1
MobilePlatform    is_controlled_by   Collide 2
```

Figure 18. Configuration for Level 0 control system

The configuration in gives the Forward module a priority lower than that of Turn or Collide so that control from either will override the Forward module, and stop the robot from moving. Therefore, the robot will not move while executing a turn. This architecture therefore gives the same behaviour as the SA level 0 control system. The control system is slightly different from that defined for the subsumption architecture, as the output of the Runaway module is split. This is necessary because *MARS* defines headings in terms of single motions, and a module cannot be sent a combined rotation/translation motion. Therefore, the *MARS* Runaway module generates a heading which is sent to both Turn and Forward modules. These modules extract the turn (rotation) and forward (translation) components of the heading and output the appropriate control message. Analysis and synthesis of this configuration produces the modular robot specification shown schematically in Figure 19.

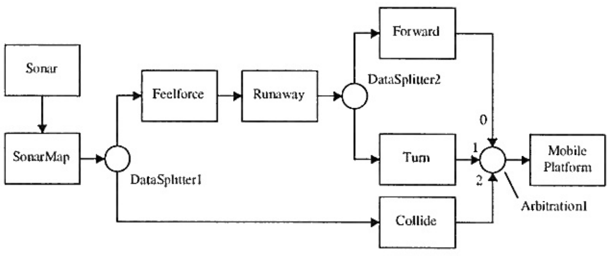


Figure 19. MARS equivalent of level 0 control system

DataSplitter1 manages the divergence of data consequence from the SonarMap to the Feelforce and Collide modules. DataSplitter2 resolves the divergence from the Runaway to the Turn and Forward modules. An ArbitrationNode has been introduced to resolve the convergence of control at the MobilePlatform module.

Brooks [10] also defines a level 1 control system, which adds a 'wander'competence, as shown in Figure 20. The Wander module generates a random heading for the robot to follow. This is then passed to the Avoid module, which also takes in the output of the Feelforce module. The output of the Avoid module is a heading which points the robot in the general direction specified by the Wander module, perturbed to avoid any obstacles. The output of the Avoid module then suppresses the output of the Runaway module and replaces it with its own output. Hence, the Avoid behaviour module subsumes the operation of the Runaway module.

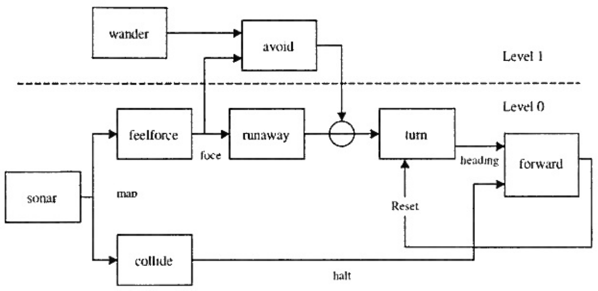


Figure 20. Level 0 and 1 control systems (from [10])

The level 1 control system specified by Brooks raises an important issue. The SA relies on the ability of higher-level outputs to subsume other output lines. In *MARS* this is supported for control through the use of the priority parameters of the is_controlled_by relationship. *MARS* does not currently define the subsumption of data. Therefore, the subsumption architecture as specified by Brooks to Level 1 cannot be directly modelled in *MARS*. Figure 21, however, shows an architecture which realises the same functionality as the level 1 SA. Here, the output of the Wander behaviour feeds directly into an ArbitrationNode which competes with the output of the Level 0 control system for the MobilePlatform.

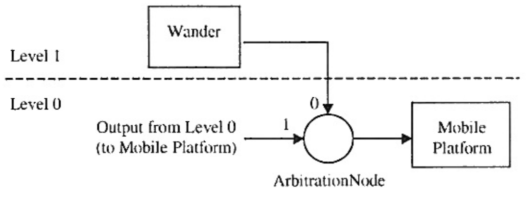


Figure 21. MARS equivalent of Level 1 control system

Brooks has been criticized for the subsumption architecture which Connell [11] claims relies on a holistic approach to a non-intuitive design process. *MARS*, in contrast, offers a more rigorous approach to designing reactive robot architectures.

6 Summary and Conclusions

In summary, we have presented two aspects of the *MARS* model that are important components of robot architectures. The *MARS* model aims at modelling and reasoning about modular robot architectures. We have described the modelling of modules in terms of data and control, and the modelling of the data and control relationships that define interaction between modules in a robot architecture. We have discussed the application of the model to Brooks' subsumption architecture.

The *MARS* model has been implemented and tested on a small set of case studies to evaluate the general approach. More extensive studies are now required to evaluate it both theoretically and practically on complex scenarios incorporating real-time control and data/control synchronisation. Further research is also required to extend the model to incorporate support for high-level task models and runtime reconfiguration. These studies will provide the focus for future research.

References

[1] C. J. J. Paredis and P. K. Khosla, "Kinematic Design of Serial Link Manipulators from Task Specifications," *Int. J. Robotics Research*, Vol. 12, No. 3, pp. 274-287, 1993.

[2] Satoshi Murata, Haruhisa Kurokawa, Eiichi Yoshida, Kohji Tomita and Shigeru Kokaji, "A 3-D Self-Reconfigurable Structure," *Proc. 1998 IEEE Int'l Conf. Robotics and Automation*, pp. 432-439.

[3] Keith Kotay, Daniela Rus, Marsette Vona and Craig McGray, "The Self-reconfiguring Robotic Molecule," *Proc. 1998 IEEE Int'l Conf. Robotics and Automation*, pp. 424-431.

[4] Robert J. Anderson, "SMART: A Modular Control Architecture for Telerobotics," *IEEE Robotics and Automation Magazine*, September 1995, pp. 10-18.

[5] Juan A. Fernandez and Javier Gonzalez, "NEXUS: A Flexible, Efficient and Robust Framework for Integrating Software Components of a Robotic System," *Proc. 1998 IEEE International Conference on Robotics & Automation*, pp. 524-529.

[6] Sara Fleury, Matthieu Herrb, and Raja Chatila, "G^{en}oM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture," *Proc. IROS '97*, pp. 842-848.

[7] Hélène Chochon, "Object-oriented design of mobile robot control systems," *2nd ISER*, Toulouse, France, June 1991, pp. 317-328.

[8] J. A. Fryer and G. T. McKee, "Resource Modelling and Combination in Modular Robotics Systems," *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, Leuven Belgium, May 16-20, 1998, pp. 3167-3172.

[9] J. A. Fryer, G. T. McKee and P. S. Schenker, Physical Configuration Reasoning for Modular Robotics Systems, in *Proceedings of the IASTED International Conference on Robotics and Automation 2000*, Honolulu, Hawaii, pp. 248-254, 2000.

[10] Rodney A. Brooks, "A Robust Layered Control System For A Mobile Robot," *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, March 1986.

[11] Jonathan H. Connell, Minimalist Mobile Robotics: A Colony-style Archi-tecture for an Artificial Creature, Perspectives in Artificial Intelligence, Vol. 5, Academic Press, Inc, San Diego, 1990.