Proceedings of the 2007 IEEE Swarm Intelligence Symposium (SIS 2007)

JaMOS - A $\mathbf{MDL2}\epsilon$ based Operating System for Swarm Micro Robotics

Marc Szymanski and Heinz Wörn

Institute for Process Control and Robotics
Universität Karlsruhe
76128 Karlsruhe, Germany
Email: {szymanski, woern}@ira.uka.de

***Abstract*— Micro robots in large scale swarms often have a very restricted program memory which limits the robot's application range. We present a finite state machine operating system for swarm micro robots, that can overcome such problems and gives the designer of swarm algorithms a tool that is easy to handle. The operating system's flow control or rather the robot's control program is represented in the Motion Description Language Two Extended ($\mathrm{MDL2}\epsilon$). $\mathrm{MDL2}\epsilon$ is based on MDLe but has been extended to a fully functional behaviour description language as shown in this paper. The $\mathrm{MDL2}\epsilon$ based control programs are encoded in a byte code that is interpreted on a micro controller. The byte-code concept significantly reduces the size of the control program which will be shown in this paper.**

I. Introduction

Swarm robotics tries to create complex swarm behaviour with rather simple robots. The complex behaviour should arise from the robot-robot and robot-environment interaction based on comparatively primitive individual behaviour. This paradigm gets more important the smaller the robots and the larger the swarm becomes. Expecially when the robot's size advances to the micro- or nanoscale.

Working with large scale swarms of micro robots one is faced with the problem of limited on-board program memory that crucially constraints the application range. The Alice robot from EPFL for instance has 8 KByte of program memory [1], our Jasmine robot 16 KByte and the I-SWARM robot has only 8 KByte. Therefore designing an operating system for swarm micro robots that is small in program size is an important step in swarm micro robotics. Another problem one is faced with is that usual micro controller architectures do not allow to change the program code during runtime. This makes batch programming of the whole swarm with wireless techniques complicated. Our approach to this problem is an operating system that interprets a byte code and therefore is independent of the size of the program memory. The byte code can be placed in the RAM or in external memory devices. On the other hand the control program of the robot should still be as flexible as possible.

In swarm robotics many mechanisms exist to control the autonomous robots in a swarm. Those mechanisms reach from mainly reactive control [2] by artificial neural networks (ANN) [3], up to more high level control. Combining purely reactive control with other control mechanism like PID controllers, fuzzy logic or ANN leads to a more complex and more usable way of controlling swarm robots. This hybrid control can be covered by the proposed *Jasmine* $\mathbf{MDL2}\epsilon$ *based Operating System* (JaMOS).

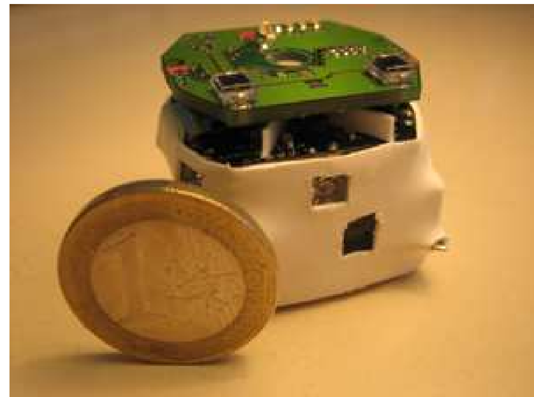


Fig. 1. The micro-robot Jasmine III+ with ODeM board on top.

The operating system combines data acquirement and data processing with robot control in an easily exchangeable manner. The control flow is expressed in $\mathbf{MDL2}\epsilon$ which is transferred as byte code to the robot and than decoded by an interpreter. So the design of JaMOS follows the design of a finite state machine operating system (FSMOS). Expressing a control program as a finite state automaton in an interpretable language has several advantages:

- simulation of the robot in any adaptable simulation, environment (plug-ins for webots, breve and player/stage have been implemented.),
- easy design of the program,
- reusability of code,
- inter robot code exchange,
- minimising code size,
- batch programming of the swarm and
- verifiability and analysability of the $\mathbf{MDL2}\epsilon$-plan with tools from language theory.

Being able to simulate a swarm before running the algorithms on the real swarm saves time and leads to an easy design of

1-4244-0708-7/07/$20.00 ©2007 IEEE

the programs. The way MDL2ϵ works makes it simple to reuse coded behaviours. This reusability can also be utilised by the robots. They can have different codes on board or even construct code for themselfs, that can be exchanged among robots.

The following section introduces the Jasmine robot for which JaMOS was implemented. Afterwards in Section III a brief introduction to MDL2ϵ is given. Section IV describes the JaMOS operating system. Section V shows results on the achieved code size reduction and section VI gives an exemplary MDL2ϵ-plan for a simple swarm tag game.

II. The Jasmine Swarm Micro-Robot

The underlying swarm micro-robot Jasmine was developed especially for swarm robot research and swarm robot games [4]. Despite its small size of about $30 \times 30 \times 30$ mm^3, it has excellent local communication abilities and a far distance scanning and distance measuring sensor. The excellent communication abilities result from six infra-red sensors and emitters arranged around the robot with a displacement of 60 degrees. Those sensors can also be used for short distance measurements. The far distance measuring sensor is hooked to the front of the robot. Two differentially driven wheels, that are controlled via an extra micro controller, give this micro-robot a high maneuverability at a high speed. Optical encoders allow odometry measurements in the mm-range. Different from many other swarm robots Jasmine supports only local communication. Long distance communication via radio frequency is not implemented and does not correspond with the views of the construction team about swarm robot capabilities.

Figure 1 shows the Jasmine swarm micro-robot equipped with the *Optical Data transmission system for swarm Micro robots* (ODeM). The ODeM board allows in conjunction with an external camera, computer, DLP projector and the MDL2ϵ interpreter a batch programming of the robot swarm, data transmission during an experiment, up to four virtual optical pheromones and provides the robot with structured light for calculating its own position in the work space [5].

The robot is equipped with an Atmel Mega168 8 bit μController programmed in AVR-C. The μController has 16 KByte flash memory, 1 KByte RAM and 512 Bytes EEPROM and runs with an internal oscillator with a frequency of 8 MHz.

III. MDL2ϵ in a Nutshell

The basic idea behind MDL2ϵ is to describe the robot's controller as regular language. The words produced by this regular language correspond with the chronology of the actions a robot took. Those actions are the letters of the language and stand for single actions like moving forward, gripping an object, or changing internal states. In the following those letters are called *atoms*. In the original MDLe proposed by Manikonda et al. [6], [7] and also in MDL2ϵ atoms posses an interrupt and a timer. The timer is the upper limit how long such an atom should be executed. The interrupts connect the robot's internal state and sensor inputs with an atom and has to be *true* – also called *active* – that an atom is executed. The regular language describes the possible flow of execution of active atoms and is described by a regular expression. Like in a regular expression MDL2ϵ supports brackets, union, kleen-star operator and concatenation. The corresponding MDL2ϵ-operators are BEHAVIOUR, UNION, RUNION and MULT. For concatenation no special operator is needed. An MDL2ϵ-plan is expressed as a combination of those operators.

TABLE I

List of all boolean operators and their MDL2ϵ counterparts. <interrupt> is an arbitrary combination of those functions and basic interrupts.

$\wedge$	AND(<interrupt>, <interrupt>)
$\vee$	OR(<interrupt>, <interrupt>)
$\neg$	NOT(<interrupt>)

TABLE II

List of all comparison operators implemented in MDL2ϵ. <value> can be a variable or a constant value.

$=$	EQ(<value>, <value>)
$\geq$	GEQ(<value>, <value>)
$>$	GT(<value>m <value>)

Like the atom a behaviour also has an interrupt and a timer. Using behaviours the state space could be divided, e.g. into a behaviour for searching food or for bringing food back to the hive. Therefore MDL2ϵ is not only reactive but could be seen as a behaviour based control paradigm.

Interrupts are boolean expressions that combine basic interrupts which are preprogrammed in MDL2ϵ e.g. IOBSTACLE which evaluates to *true* if an obstacle is sensed on one of the frontal sensors. Therefore, the boolean operators AND, OR and NOT are implemented, see Table I. Additionally to the boolean operators comparison operators were implemented. Those operators are stated in Table II.

An important MDL2ϵ-operator is the RUNION or random union. It allows to randomise the execution flow. This can be used to model time varying behaviours, which execution probability decreases or increases over time, or for instance to model behaviours, which execution probability correlates with the portion of pheromone sensed. Therefore each MDL2ϵ-operator has a probability attribute which could be a constant value or a variable. The probability distribution of a random union is calculated with equation 1:

$$p(i) = \frac{v_p(i)}{\sum_{k=0}^{N-1} v_p(k)}, \tag{1}$$

whereas $p(i)$ is the *probability* and $v_p(i)$ is the *probability value* of the *i*th member of the union and N is the number of members. Other calculations are possible, e.g. ϵ-greedy for Reinforcement Learning.

All MDL2ϵ commands are listed in Table III. Arguments

TABLE III

LIST OF ALL MDL2ϵ OPERATORS.

Type	MDL2ϵ Expression
Plan	<PLAN ** name="<string>" duration="<long>" /> . . . </PLAN>
Atom	<ATOM name="<string>" interrupt="<string>" * arg0="<double>" ... arg19="<double>" duration="<long>" * probability="<int>"/>
Behaviour	<BEHAVIOUR name="<string>" interrupt="<string>" duration="<long>" * probability="<int>"> . . . </BEHAVIOUR>
Union	<UNION ** name="<string>" * probability="<int>"> . . . </UNION>
Random Union	<RUNION ** name="<string>" * probability="<int>"> . . . </RUNION>
Multiplicity	<MULT ** name="<string>" multiplicity="<string>" * probability="<int>"> . . . </MULT>

TABLE IV

SOME BASIC ATOMS FOR THE JASMINE ROBOT.

Atom	Arguments	Explanation
ARS232	int	Send an integer via the rs232 interface.
ASEND	message, send count, sensors	Send a 8 bit message.
ABLUEPHERO	on/off	Start/stop emitting the blue pheromone.
AGREENPHERO	on/off	Start/stop emitting the green pheromone.
AREDPHERO	on/off	Start/stop emitting the red pheromone.
ASETSEM	semaphore number	Set semaphore.
ARELSEM	semaphore number	Release a semaphore.
AROT_R	degree	Rotate left.
AROT_L	degree	Rotate right.
ASTOP		Stop moving.
AMOVE	velocity (0,1,2,3,4), steps	Start moving.

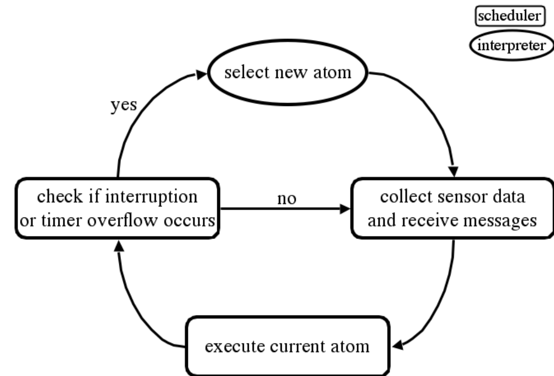


Fig. 2. The general execution cycle in JaMOS.

that are marked with a '*' are optional. Arguments marked with '**' are ignored and are only useful for documentation purpose. An exemplary plan is given in Table XI.

IV. JAMOS

This section describes in detail the *Jasmine* MDL2ϵ *Operating System* (JaMOS). JaMOS is comprised of five layers. Beginning from the bottom layer:

1) the physical layer that represents the robots basic abilities, which are constraint by the underlying hardware,
2) the Basic Input-/Output system (BIOS),
3) the MDL2ϵ layer which connects the atoms, interrupts and variables via the BIOS with the hardware,
4) the interpreter that also schedules the control flow,
5) and the byte code.

The physical layer represents the hardware robot which is abstracted by the BIOS that implements basic abilities for reading sensor data, communication, remote control and motion control.

At the MDL2ϵ layer a library of atoms, interrupts and variables exists that allows the MDL2ϵ plan to execute the desired robot control. The key idea is that this layer is implemented to give a wide range of possibilities to the designer. In Tables IV, V, and VI a list of basic atoms, variables and interrupts is given. However, this layer can be adopted to the needs of the robot programmer.

The fourth layer holds the interpreter/scheduler. The interpreter is responsible for interpreting the byte code whereas the scheduler's task is to continuously poll for the right sensors for the currently active interrupts, to evaluate the interrupts and timers and in case of a state switch order the interpreter to select a new atom from the plan.

TABLE V

SOME BASIC VARIABLES FOR THE JASMINE ROBOT. ALL VARIABLES ARE 8 BIT UNSIGNED INTEGERS.

Variable	Explanation
VREMOTE	Last remote control command.
VMESSAGE1	Received message (low byte).
VMESSAGE2	Received message (high byte).
VENERGY	On-board energy level.
VTOUCH	Last touch sensor value.
VSENSOR0	Last beamer sensor value.
VSENSOR1	Last sensor value front.
VSENSOR2	Last sensor value right front.
VSENSOR3	Last sensor value right back.
VSENSOR4	Last sensor value back.
VSENSOR5	Last sensor value left back.
VSENSOR6	Last sensor value left front.
VDATAL	Last data byte (low byte).
VDATAH	Last data byte (high byte).
VGREYL	Last pheromone value from left sensor.
VGREYR	Last pheromone value from right sensor.
VXPOSH	Last x position (high byte).
VXPOSL	Last x position (low byte).
VYPOSH	Last y position (high byte).
VYPOSL	Last y position (low byte).
VTHETAH	Last angular position (high byte).
VTHETAL	Last angular position (low byte).

Figure 2 shows how the interpreter and the scheduler are working together in the general execution cycle. Always when a state switch occurs due to an interrupt that gets inactive or a timer overflow a new atom is selected by the interpreter from the byte code. Otherwise the scheduler moves on with collecting data and executing the atom.

During the atom selection process the interpreter pushes all interrupts that are part of top level behaviours on an interrupt stack. The last entry of this stack is the interrupt of the newly selected atom. To be selected by the interpreter all interrupts on the stack must be true at the moment of selection. The timers are saved on a second stack. If an active atom was found the scheduler starts its work again and checks all interrupts on the stack in each execution cycle. If an interrupt occurs the next atom will be selected starting from the follower of the behaviour/atom that caused the interruption. The following avoidance behaviour is given as an example for the execution flow:

```
<PLAN name="avoidance" duration="infinite">
  <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" duration="infinite"/>
  <BEHAVIOUR interrupt="ITOUCH" duration="infinite" >
    <ATOM name="AROT_L" interrupt="AND( ISPACEL, NOT(
    TWISYNC ))" duration="50" arg0="25"/>
    <ATOM name="AROT_R" interrupt="NOT( OR( ISPACEL,
    TWISYNC ))" duration="50" arg0="25"/>
  </BEHAVIOUR>
</PLAN>
```

TABLE VI

SOME BASIC INTERRUPTS FOR THE JASMINE ROBOT.

Interrupt	Active
* basic interrupts *	
ITRUE	always
IFALSE	never
* ODeM interrupts *	
IGREYSYNC	Got new pheromone value.
IPOSSYNC	Got position update.
IDATASYNC	Received new data.
* Main board Interrupts *	
ITWISYNC	Step-wise motion finished.
IOBSTACLE	Sensed an obstacle.
IINMOTION	Moving.
IDIRECTION	Moving forwards.
ICRITCOLLISON	Sensed critical collision
* communication interrupts *	
IREC	Received a message.
IRECON1	Received message on front sensor
IRECON2	Received message on front right sensor
IRECON3	Received message on back right sensor
IRECON4	Received message on back sensor
IRECON5	Received message on back left sensor
IRECON6	Received message on front left sensor
* proximity interrupts *	
ITOUCH	Sensed an obstacle.
ISPACEL	There is more space left than right.
* semaphore interrupts *	
ISEMAPHORE	Semaphore 0 is set.
ISEMAPHORE1	Semaphore 1 is set.
...	...
ISEMAPHORE7	Semaphore 7 is set.

TABLE VII

EXAMPLE PULSE DIAGRAM FOR THE EXECUTION OF THE AVOIDANCE PLAN.

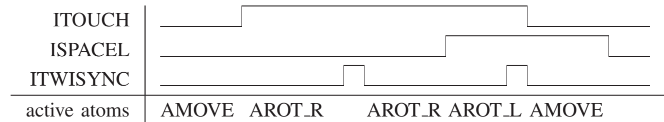
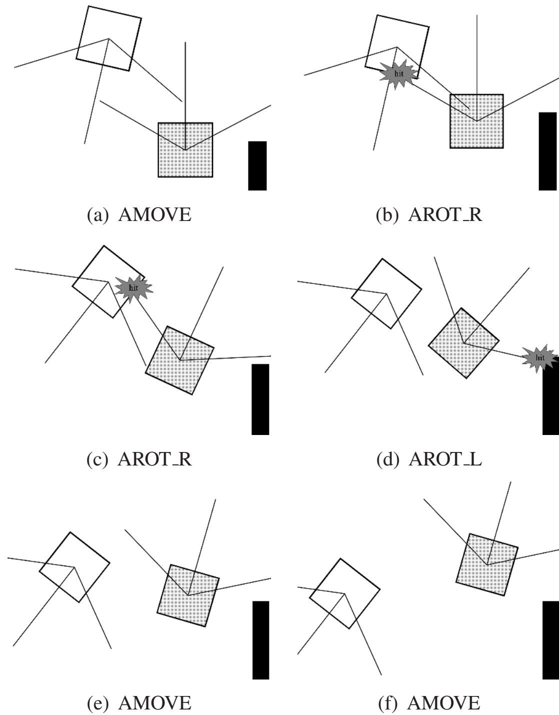


Table VII shows the interrupt pulse diagram for the given example and Fig. 3 shows the robot's motion. In the first step the duration of the PLAN-node will be put on the timer stack. As a placeholder interrupt ITRUE will be pushed on the interrupt stack. If ITOUCH evaluates to false, no object is in front of the robot, NOT(ITOUCH) will be pushed on the interrupt stack and "infinite" on the timer stack. AMOVE is found to be active and will be executed, Fig. 3(a), until ITOUCH switches to true, Fig. 3(b). Then the interpreter starts searching for the next active atom. The old interrupt and timer will be popped from the stack and the timer and the interrupt from the next BEHAVIOUR node will be pushed



(a) AMOVE (b) AROT_R (c) AROT_R (d) AROT_L (e) AMOVE (f) AMOVE

Fig. 3. Exemplary run of the avoidance behaviour. The grey robot is the observed robot. The lines symbolise the distance sensors.

on the stack. The signal TWISYNC gets active when a stepwise motion command was finished by the motor board, see Table VI. By default AROT_L and AROT_R are such stepwise motion commands. At the beginning TWISYNC is false. The robot has more free space on the right hand side, so ISPACEL is false. Therefore AROT_L will not be selected but AROT_R. Again the interrupt and the timer will be pushed on the stack. After the robot rotated 25 degrees the ITWISYNC signal switches to true and the interrupt evaluates to false. Then the plan is interpreted from the beginning, starting with the PLAN-node. There is still an obstacle in front of the robot, Fig. 3(c). The robot starts turning right again. This time the atom is interrupted by ISPACEL before the ITWISYNC signal is received. It seems that the robot rotated too far and there is an obstacle on the right hand side. Now the robot starts rotating to the left, Fig. 3(d). After rotating 25 degrees to the left, Fig. 3(e), the robot continues moving forward again, Fig. 3(f).

The byte code is the most important part of JaMOS. It makes the programming of the robots simple, flexible and exchangeable.

We pursued two ways in generating the byte code. The first way generates the byte code and an interpreter minimal in program size. Therefore the plan is analysed prior to the encoding step according to which $\mathbf{MDL2}\epsilon$ operators, atoms, interrupts and variables are used. The disadvantage of this is that the interpreter can only interpret plans that are coded the

TABLE VIII

EXAMPLE BYTE CODE FOR AN ATOM.

ATOM:
<ATOM name="AMOVE" interrupt="ITOUCH" duration="1024" arg0="255" arg1="1"/>

ATOM	ITOUCH	#arguments	duration
000	10010	10	0000010000000000
arg0	arg1	AMOVE	
11111111	00000001	1011	

same way and do not include other operators, atoms, interrupts, and variables. For changing the control flow or adding new atoms to the plan, the whole interpreter has to be recompiled. This is adequate if very limited memory resources are provided by the underlying hardware. But the exchangeability of code between the robots and the batch programming possibility is lost.

The second way is to allow to use all $\mathbf{MDL2}\epsilon$ operators and a set of atoms, interrupts and variables. This results in a larger basic operating system, but, on the other hand, one can exploit all benefits that are provided by an interpreted control language.

The basic structure of the byte code is to encode all operators, atoms, interrupts and variables as unique bit strings, whereas all of them are treated separately. This reduces the size and can be done due to the fact that the $\mathbf{MDL2}\epsilon$ structure is fixed. The $\mathbf{MDL2}\epsilon$ operators are generally coded with 3 bits. All other $\mathbf{MDL2}\epsilon$ members are coded by a bit string of size n, with $2^n \leq$ *#members in the library*. E. g. the 11 atoms in Table IV can be coded with $\lceil \text{ld } 11 \rceil = 4$ Bits, the 22 variables in Table V with $\lceil \text{ld } 22 \rceil = 5$ Bits and the 27 interrupts in Table V are also coded with $\lceil \text{ld } 27 \rceil = 5$ Bits. Those bit strings are internally mapped directly via an array containing the pointers to the corresponding functions and variables.

Operators like atoms, behaviours and multiplicities have different arguments. The encoding of such operators will be explained in the following. Detailed examples are given for an atom in Table VIII and a behaviour in Table IX.

1) ATOM: An atom has four attributes: *name, interrupt, duration,* and the *arguments*. The atom is coded in the following way. First comes '000' that denominates the ATOM itself. Then the interrupt which can be of variable size, the number of arguments which is also variable, the duration which is always a 16 Bit string, then the arguments which are all 8 bit and finally the coded address of the function that has to be executed.

2) BEHAVIOUR: A behaviour is separated into two parts. The first part is the behaviour declaration that contains the code that should be executed in the behaviour. And the second is the behaviour call. The call includes the behaviour denominator '001' the interrupt and the behaviour's 16 bit start address in the byte code that points to the behaviour declaration. Each behaviour declaration starts with the duration continuous with the code and ends with '111'. Behaviour declarations are copied to the beginning of the byte code. The behaviour in the

TABLE IX

EXAMPLE BYTE CODE FOR A SIMPLE BEHAVIOUR.

```
<BEHAVIOUR interrupt="ITRUE" duration="infinite" >
  <ATOM name="AMOVE" interrupt="ITOUCH"
  duration="1024" arg0="255" arg1="1"/>
</BEHAVIOUR>
```

declaration:

behaviour duration	atom code	behaviour end
0000000000000000	000100101000000 100000000001111 111100000001101 1	111

behaviour call:

BEHAVIOUR	ITRUE	declaration address
001	000	0000000000000000

example in Table IX has the start address 0x0000. Following behaviours will be added at the end of the last behaviour starting at the next free byte.

3) UNION: The UNION starts with ’010’, then all other $\mathbf{MDL2}\epsilon$ entries follow in the described way and ends with ’111’.

4) RUNION: To save code size are the probabilities associated to the RUNION and not to the single operators. The RUNION starts with the string ’011’ followed by the 8 bit number of elements inside of the RUNION. Then all probability values, as described in section III, follow as 8 bit numbers. Afterwards all $\mathbf{MDL2}\epsilon$ entries follow and it finishes with ’111’.

5) MULT: The MULT is addressed with ’100’ which is followed by an 8 bit number expressing the number of loops that should be performed. The following body contains the encapsulated $\mathbf{MDL2}\epsilon$ commands. It also ends with ’111’.

6) PLAN: The plan has only one argument, the duration. For decoding a plan, the plan start address is given to the interpreter. Each plan ends with the string ’111’.

7) Interrupts: Interrupts are a fundamental part of $\mathbf{MDL2}\epsilon$ in the examples in Tables VIII and IX only a single simple interrupt is used. However, interrupts can be constructed by using the boolean and comparison operators from Table I and II. As operands constant 8 bit values and $\mathbf{MDL2}\epsilon$ variables are possible. The interrupt byte code follows the same prefix notation like the operands in the XML-plan. For indicating an operator or an operand a preceding indication bit that is ’0’ for an operand an ’1’ for an operator is introduced. As in the latter cases all operands have their fix indication bit string. To separate between variables and constants another indicator bit has been introduced that is ’0’ for constants and ’1’ for variables. The $\mathbf{MDL2}\epsilon$ interrupt *AND(IRECON3,EQ(VMESSAGE1,42))* that describes that the robot received 42 as message on sensor 3 will be encoded to

O	AND	O	IRECON3	O	EQ	O	V	VMESSAGE1	I	V	42
1	000	0	01101	1	011	0	1	0001	0	0	00101010

where ’O’ stands for the operator/operand and ’V’ for the variable/constant indication flag.

TABLE X

COMPARISON BETWEEN GENERATE BYTE-CODE AND C-CODE ROBOT CONTROLLERS IN BYTE FOR THE SWARM TAG GAME PLAN.

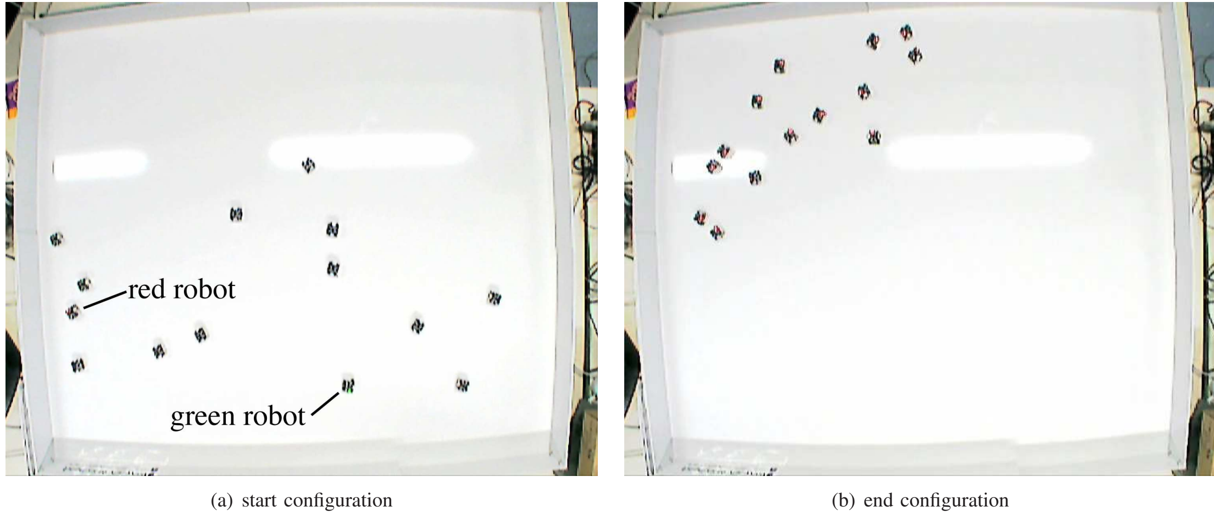
	byte-code	c-code
$\mathbf{MDL2}\epsilon$ plan	120	1024
$\mathbf{MDL2}\epsilon$ structure	1704	728
total	1824	1752

The way of encoding behaviours by their address in the byte code is important for reusability issues. In case of enough memory a wide variety of preprogrammed behaviours can be saved in memory. Those behaviours can then be reused just by pointing to their address. This feature can be used for behaviour based learning, where several behaviours have to be evaluated by the learning system. But one can also save several plans on board that are constructed from those behaviours. A 64 KByte TWI memory device is integrated on the ODeM board. Using preprogrammed behaviours also saves device programming time which is crucial for the ODeM that operates with an estimated transfer rate of 6 Byte per second without redundancy and 3 Byte per second with redundancy for error correction.

V. RESULTS

One of the main goals of JaMOS is to reduce the size of a swarm robot control program. Several measurements have been performed with plans of different sizes and different structural elements. We compared JaMOS with an earlier version of JaMOS which generates C-code that represents the $\mathbf{MDL2}\epsilon$-plan using mostly if-then-else statements and additional code for unions, random unions and multiplicities. Both versions of JaMOS are equal in their functionality regarding the robot behaviour and they share the same code for atoms, interrupts and the BIOS.

We distinguished in this comparison between a structural part and the $\mathbf{MDL2}\epsilon$-plan that was created from the XML-file. The structural part covers the data acquirement, timer management and interrupt handling. This structural part is for the byte-code version four times larger as it also covers the $\mathbf{MDL2}\epsilon$ interpreter, which has a size of less than 3 KByte. However, the advantage in the structure part is lost when it comes to the representation of the $\mathbf{MDL2}\epsilon$ plan. Here the strength of the interpreter approach is visible. The interpreted plan needs only a ninth up to a thirtieth of the space of the C-code version. This advantage grows dramatically if the plan gets larger. However, those advantages decrease with a smaller plan. Table X shows the code sizes for the plan listed in Table XI. Here the C-code version has a slight advantage. The structural part of the byte-code in this example is much smaller than 3 KByte what results from the possibility of adapting the interpreter to the used $\mathbf{MDL2}\epsilon$ operators. As the tag game plan has no random union, union or multiplicity the interpreter code for this operators has been switched off and has not been compiled.



(a) start configuration (b) end configuration

Fig. 4. Figure (a) and (b) show the start and end configuration of the Jasmine tag game. During this run the red robots won the game and aggregated in the upper left corner of the arena, see Fig. (b). The full video to this experiment can be downloaded from www.swarmrobot.org. The video shows the swarm game in simulation and in real.

TABLE XI

MDL2ϵ-CODE FOR THE SWARM TAG GAME.

```
 1 <MDLeScript>
 2    <PLAN name="JasmineGame" duration="infinite">
 3
 4      <BEHAVIOUR name="random-walk" interrupt="NOT(IREC)" duration="infinite">
 5         <ATOM name="AMOVE" interrupt="NOT(ITOUCH)" duration="infinite"/>
 6         <BEHAVIOUR name="avoidance" interrupt="ITOUCH" duration="infinite">
 7            <ATOM name="AROT_L" interrupt="AND(ISPACEL,NOT(ITWISYNC))" duration="20" arg0="120"/>
 8            <ATOM name="AROT_R" interrupt="AND(NOT(ISPACEL),NOT(ITWISYNC))" duration="20" arg0="120"/>
 9         </BEHAVIOUR>
10      </BEHAVIOUR>
11
12      <BEHAVIOUR name="red" interrupt="EQ(VMESSAGE1,1)" duration="infinite">
13         <ATOM name="AREDON" interrupt="ITRUE" duration="1" />
14         <ATOM name="ASEND"  interrupt="ITRUE" duration="1" arg0="1" arg1="3"/>
15         <ATOM name="ASTOP"  interrupt="ITRUE" duration="infinite" />
16      </BEHAVIOUR>
17
18      <BEHAVIOUR name="green" interrupt="EQ(VMESSAGE1,2)" duration="infinite">
19         <ATOM name="AGREENON" interrupt="ITRUE" duration="1" />
20         <ATOM name="ASEND"    interrupt="ITRUE" duration="1" arg0="2" arg1="2"/>
21         <BEHAVIOUR name="random-walk" interrupt="ITRUE"/>
22      </BEHAVIOUR>
23
24    </PLAN>
25 </MDLeScript>
```

VI. SWARM EXPERIMENTS

This section describes an exemplary swarm experiment implemented in MDL2ϵ utilising JaMOS. Other swarm experiments using MDL2ϵ have been performed in [8] and [9]. In the following experiment the robots play tag. This simple game includes aggregation, collision avoidance, inter robot communication and random walk based dispersion. At the beginning of the game all robots perform a random walk with collision avoidance. Two of those robots try to infect the other robots with two different kinds of behaviours, *red* and *green*, via infra-red communication. Behaviour *red* makes the robots stop, turn on their red light and allows each of those robots to infect three other robots. The other behaviour keeps the robot in the random walk mode and they turn on their green light. Those robots are allowed to infect two other robots. All robots could be infected by any of those two behaviours regardless of the behaviours they executed before. The game stops when there are only red or only green robots in the arena.

The $\mathbf{MDL2}\epsilon$-plan for the tag game is listed in table XI. It is structured in three main behaviours: in lines 4-10 the random walk behaviour, which includes the avoidance strategy; lines 12-16 behaviour *red*; and lines 18-22 behaviour *green*, which reuses the random walk behaviour in line 21.

The plan can be read the following way. As long as the robots did not receive any message (line 4) they go on with a random walk including collision avoidance, which means they move forward until they sense an obstacle with their frontal sensors (line 5). If they sensed any obstacle (line 6) they rotate anti-clockwise, if there is more space at the left hand side (line 7) otherwise rotate clockwise (line 8). If they received a message they check if the message is equal 1 or 2 (line 12 and 18). If it is 1, behaviour *red* is executed, the robot turns on the red light (line 13), starts sending message 1 three times (line 14) and stops (line 15). If the received message was 2 the robot turns the green lamp on (line 19), starts sending message 2 two times (line 20) and performs the random walk with collision avoidance (line 21). It is to say that the ASEND atom starts a communication process with acknowledge, which runs concurrently to the execution of other active atoms even if the ASEND atom is not active any more. It stops sending when the robot sent all allowed messages to other robots.

Figure 4 shows two snapshots of the experiment, the start configuration and the end configuration with a cluster of red robots. The experiment shows a highly dynamic behaviour that leads to clusters of red robots that move through the arena permanently dispersing to green robots and aggregating as red. The same experiment has also been performed with a second programming method that implemented the behaviours in a finite automaton in C-code using the autonomy cycle approach [10]. No significant difference in terms of the robots' behaviour can be seen between both methods. The continuous interpretation of the $\mathbf{MDL2}\epsilon$-program and especially interrupts do not lead to a worse response time of the robot.

VII. Summary and Outlook

In this paper a new finite state machine operating system for swarm micro robots has been introduced. The task control is based on regular Motion Description Language Two Extended which has been described in detail. The operating system uses an interpreted byte code version of $\mathbf{MDL2}\epsilon$ as control program. The byte code and the interpreter can be automatically adjusted to the used control program but can also be based on a fixed description.

Subsequently the interpreter is evaluated in terms of memory efficiency which shows a significant reduction in terms of a ninth up to a thirtieth of the former C-coded size. Then an example is given that describes a swarm tag game utilising some canonical behaviours like collision avoidance, aggregation and dispersion through communication.

In future work this operating system will be enhanced with learning abilities, e.g. Reinforcement Learning. Currently work is going on that utilises $\mathbf{MDL2}\epsilon$ for Genetic Programming. Several other extensions like atoms that are based on Artificial Neural Networks or fuzzy controllers can be implemented.

In general the strength of this operating system lies in utilising the possibility of exchanging code fragments among robots in a swarm. Using a well designed BIOS and well thought atoms, interrupts and variables even the code exchange between inhomogeneous robots is possible. Combining code exchange with swarm memory effects can lead to a more efficient and more adaptable swarm. Several programs can latently exist in the swarm and during a recruitment phase communicated from one recruited robot to another.

Acknowledgement

This work supported by the European Union within the "Beyond Robotics" Proactive Initiative – 6th Framework Programme: 2003-2007 (I-SWARM project, Project Reference: 507006).

References

[1] G. Caprari and R. Siegwart, "Design and control of the mobile micro robot alice," 2003.

[2] R. Beckers, O. E. Holland, and J.-L. Deneubourg, "From local actions to global tasks: Stigmergy in collective robotics," in *4th Int. Workshop on the Sythesis and Simulation of Living Systems (Artificial Life IV)*, R. Brooks and P. Maes, Eds. July 1994, pp. 181–189, MIT Press.

[3] M. Dorigo, V. Trianni, E. Sahin, R. Gro, T. Labella, G. Baldassarre, S. Nolfi, J. Deneubourg, F. Mondada, D. Floreano, and L. Gambardella, "Evolving self-organizing behaviors for a swarm-bot," 2004.

[4] Sergey Kornienko and Marc Szymanski, *Exploring Robotic Swarms with Micro Robot Jasmine*, Cornell University Library, to appear.

[5] A. Bolettis, A. Brunete, W. Driesen, and J-M Breguet, "Solar cell powering with integrated global positioning system for mm^3 size robots," in *IROS 2006*, Beijing, China, October 9-15 2006, p. pp204.

[6] V. Manikonda, P. Krishnaprasad, and J. Hendler, "A motion description language and hybrid architecure for motion planning with nonholonomic robots," 1995.

[7] Manikonda, Krishnaprasad, and Hendler, "Languages, Behaviors, Hybrid Architectures, and Motion Control," *Mathematical Control Theory*, 1998.

[8] M. Szymanski, T. Breitling, J. Seyfried, and Heinz Wörn, "Distributed shortest-path finding by a micro-robot swarm.," in *ANTS Workshop*, Marco Dorigo, Luca Maria Gambardella, Mauro Birattari, Alcherio Martinoli, Riccardo Poli, and Thomas Stützle, Eds. 2006, vol. 4150 of *Lecture Notes in Computer Science*, pp. 404–411, Springer.

[9] H. Hamannn, M. Szymanski, and H. Wörn, "Orientation in a trail network by exploiting its geometry for swarm robotics," in *IEEE Swarm Intelligence Symposium*, April 2007, in this proceeding.

[10] S. Kornienko, O. Kornienko, and P. Levi, "*Flexible Manufacturing Process Planning based on multi-Agent Technology*," in *Proceedings of the 21st IASTED International Conference on Artificial Intelligence and Applications (AIA 2003)*. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, January 2003, pp. 156–161, Innsbruck, Austria: n. a.