2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)

Ensemble Programming for Multipotent Systems

Oliver Kosak, Felix Bohn, Felix Keller, Hella Ponsar, and Wolfgang Reif
Institute for Software & Systems Engineering, University of Augsburg, Germany
E-Mail: {kosak, ponsar, reif}@isse.de, {felix.bohn, felix.keller}@student.uni-augsburg.de

***Abstract*—The orchestration and controlling of groups of robots, i.e., programming ensembles, is a complicated task to do. Deciding and defining *what* the ensemble needs to do in *which order* often requires in-depth problem domain specific knowledge. Further, an ensemble programmer needs to put much effort into the design, implementation, and evaluation of distributed algorithms to coordinate multi-robot executions. Most commonly, expertise for the problem domain and programming knowledge is not united in the same person. To tackle this, we propose *Maple*, an approach for a Multi-Agent Programming Language for Ensembles. Using Maple simplifies an ensemble programmer's typical work-flow by providing generalized solutions for solving technical tasks in ensemble programming. Maple is a graphical ensemble programming language based on the formalism of hierarchical task networks that enables the online generation of ensemble programs. Maple also enables non-technical domain experts to generate ensemble programs for specific use-cases.**

***Index Terms*—ensemble programming, multi-robot systems, hierarchical task networks, self-organization, multipotent systems**

I. Introduction

Mobile robots like unmanned aerial vehicles (UAV) became very popular in industry and research, and the trend finally has also arrived in the consumer trade recently. This progress drives an ever-increasing number of robots available to a potential user, requiring possibilities for commanding and controlling more than one single robot at a time, i.e., for an *ensemble*. Current approaches focus on this problem, e.g., with aggregate programming like Meld [1], and Protelis [2] or swarm programming like Buzz [3]. Unfortunately, they have restrictions when the use case for the ensemble calls for run-time task generation or heterogeneous ensembles. Other approaches aiming at such run-time task-orchestration for ensembles like Dolphin [4] or Swarmanoid [5] lack flexibility (e.g., individual robots need to be directly addressed, or are very specialized for a single task) or proper aggregate/swarm operations. In this paper, we propose our approach *Maple* (Multi-Agent Programming Language for Ensembles) for overcoming this state of the art by supporting an ensemble's programmer with appropriate tools for the run-time commanding of ensembles on the individual and the aggregate level. With Maple, we want even non-experts in programming to be able to program and control their ensemble. Therefore, *(1)* we define what our notion of ensemble programming is by using an analogy to parallel computing, *(2)* we provide an ensemble programming script language on the level of robot capabilities, *(3)* and we hide complexity from the ensemble programmer by providing generalized solutions where possible. We embed our approach in the domain of Search and Rescue [6] where mobile robots and ensembles of such are already used for multiple decades, e.g., [7], [8]. We assume the following, very reduced example scenario: A chemical gas accident happened, the source of a gas leak (gas g) is known, and we want the robot ensemble first to evaluate the height of its dissemination. Further, the ensemble should synchronize at the determined height for finding and informing potentially endangered persons (p). Due to weight constraints, we have three differently equipped robots a_1, a_2, a_3 available. For determining the critical height of the gas's dissemination, one robot a_1 is equipped with a gas sensor. All robots are equipped with cameras for finding persons. To inform endangered persons one robot is equipped with a loudspeaker (a_2). The challenge for an ensemble programmer in this simple scenario, e.g., a firefighter confronted with that situation, is to program each robot in the ensemble appropriately to achieve that the ensemble as a whole accomplishes the defined mission. This job becomes way more complicated when tasks require an increased amount of robots or capabilities [9]. With Maple, we make this job feasible again. While in this paper, we propose Maple on a general level, we are currently deploying our findings to real robots (e.g., UAV, mobile ground robots) which we already successfully achieved for our previous findings [10]–[15]. In Section II, we define the problem of ensemble programming. We describe our approach for programming ensembles in Section III. In Section IV, we briefly review other related work in the problem domain and summarize our findings in Section V.

II. Problem Definition

Programming ensembles typically turns out to be a very complex task [16]. A straight forward approach to generate specific programs for each of the three robots of our example from Section I has many drawbacks. To be executable, each program must be tailored precisely to its robot and capabilities. If we add additional robots or replace one of the robots with another (offering different capabilities) or increase the number of participating robots, this requires new or modified robot programs. Coordination among robots needs to be explicitly defined in the programs. This requires the ensemble programmer to also deal with interactions and data exchange between robots and not only with the robots' capability executions that are of real interests. For coming by these drawbacks, we propose two main requirements for ensemble programming: (1) We require tools for enabling the ensemble programmer to define programs with possibilities to express all necessary ensemble operations, i.e., *how many robots* should execute *which actions* at *which time*. (2) We need controlling and

coordinating structures established within the ensemble to interpret and execute ensemble programs, i.e., managing inter-robot data- and execution-flow. As it is not easy to define what an *ensemble program* is, we analyze the requirements of such programs using an analogy to distributed computing [17].

An *ensemble programming language* requires some basic ingredients: (1) A synchronized storage for variables and their values, (2) an instruction set $\mathcal{I}$ for modifying those variables, and (3) control flow structuring operators for sequential, conditional, repeated, and parallel executions with a proper syntax to express all types of control flow and composition of those. Instructions in an ensemble program need to be referenced by a program counter (pc). Each pc may contain one or multiple instructions, each possibly addressing independent ensemble processing units (*EPU*). This concept can be used to express physical or logical parallelism (instructions executed on different or the same EPU) that should be supported to be used non-exclusively. This calls for an appropriate fork-join concept. As instructions in $\mathcal{I}$ may require access to the variable storage and access to that storage may occur from different EPUs, the storage needs to be shared and synchronized.

To *control* the ensemble program's execution *and schedule program instructions* an instance needs to be aware of the control flow. To determine which path of the control flow graph should be taken for conditional statements and to record the progress during the program execution (e.g., increase counter variables), the controlling instance needs read and write access to the variable storage. When the program requires parallel execution, the controlling instance needs to split up the control flow, schedule it to independently acting EPUs if necessary, and rejoin them after execution and process possible responses.

To allow the ensemble programmer to access the full potential of available operations with the ensemble, the ensemble programming language needs to offer an *instruction set* categorizing the full supported instruction set. From this knowledge base, the programmer should be able to freely associate instructions with EPUs in pc within an ensemble program, no matter which coordination pattern is used.

To assure the execution of the ensemble program's control flow correctly, EPUs addressed within pc's need to be able to *execute* the enlisted *instruction(s)* and return possibly resulting values back to the controlling instance. Due to physical or logical parallelism, EPUs also need to be able to execute multiple instructions at the same time, keep track of their execution and decide the appropriate moment for a response.

III. Approach

With Maple, we intend to hide as much complexity concerning coordination and robot interaction as possible to enable the ensemble programmer to focus on the required executions in the ensemble. Thereby, we rely on the specific characteristics of multipotent systems [15] that fundamentally differs from that of other system classes concerning the association between robots and their capabilities. Other than in 'traditional' systems (characterized as homogeneous [18] or heterogeneous [19] according to their capability to robot allocation), we separate capabilities from robots in multipotent systems. This enables the system to self-adapt the allocation of capabilities to robots at runtime [13]. Robots provide self-awareness abilities to remark changes to their set of available capabilities, so they can decide on their qualification for solving tasks independently [14]. We use this flexibility to enable the ensemble programmer to neglect system internals during the act of programming the ensemble. By that and in contrast to other approaches [20], we do not need to take the system configuration into account when creating programs for robots. Further, we make the following assumptions: Robots in our ensemble can communicate without restriction, i.e., messages do not get lost. Further, we neglect other uncertainties, e.g., robot or other hardware breakdowns (cf. Section V). In the following, we describe how we realize the instruction set $\mathcal{I}$, the behavior of EPUs within a multipotent ensemble, how we can program EPUs with a graphical programming language based on the formalism of hierarchical task networks (HTN) [21], and how we control ensemble programs with multiple EPUs.

In our algorithms describing the behavior of EPUs and the controlling instance, we use the following notation to differentiate between *service calls* (with an AS: prefix) that can be called by other robots (a_x.PROCEDURE calls AS:PROCEDURE of robot a_x) and *internal procedures* that can only be called within the robot (we do not further describe those implementations). To respond to an AS, we can access its *caller* with $\mathbb{C}$. We wait for an AS's to finish with $a.\text{PROCEDURE}(y)^{\downarrow}$, as $x \leftarrow a.\text{PROCEDURE}(y)^{\downarrow}$ if we want to access the result x, and as $a.\text{PROCEDURE}(y)^{\uparrow}$ if we just want to call the service. We write $\left(\text{P}_1 \| \text{P}_2\right)$ to execute P_1 and P_2 in parallel and wait for both to finish before continuing.

A. Capabilities as Program Instruction Set

The instruction set $\mathcal{I}$ is defined by capabilities $c \in C$ that can be allocated by robots at runtime. Obviously, $\mathcal{I}$ is defined very problem domain dependent. Along with possible parameters for each $c \in C$, the content of $\mathcal{I}$ provides a kind of API to the ensemble programmer. Parameters include capability dependent data (p) as well as a parameter to provide an option to define the capability's execution behavior (s). This behavior can either be configured to be *self-finishing* ($s = \top$) or *non-self finishing* ($s = \bot$). If a capability is executed with $s = \top$, the executing robot can determine on its own when the execution of this capability is terminated, e.g., if MOVE($destination$) has finished because the defined $destination$ is reached. If $s = \bot$, the robot can not decide on the termination of the execution itself or is not allowed to do so. Instead, the execution relies on external guidance to be terminated, either by another robot or the ensemble's user, e.g., MOVE($direction$) instructs a robot to move in a direction without defining when to stop doing so. While for some capabilities s is restricted to either be $\top$ or $\bot$, others are not restricted. The capability MEASURE($position$), e.g., may be executed with $s = \top$ to retrieve the current position of an object or with $s = \bot$ to track its position over time. The ensemble programmer can further define the execution

Alg. 1 EPU PROGRAM

```
1: C_pc ← ∅
2: procedure AS:COORD(pc)
3:   s_EPU ← ⊤
4:   e_EPU ← ⊥
5:   R_pc ← [ ]
6:   C'_pc ← C_pc
7:   C_pc ← C_EPU[pc]
8:   STOP(C'_pc \ C_pc)
9:   for c ∈ C_pc do parallel
10:    s_pc ← s_c ∨ c[s]
11:    e_pc ← e_c ∧ c[e]
12:    if c[s] then
13:      p ← c[p]
14:      R_pc[c] ← EX(c, p)
15:    else
16:      EX(c, c[p])↑
17:  join
18:  ℂ.SYNC(s_pc, e_pc, R_pc)
```

behavior of all capabilities by setting an additional parameter e. By setting $e = \top$ (from a default $e = \bot$) for a capability, the ensemble programmer can explicitly request interaction with the ensemble before it can continue execution.

B. Robots as Ensemble Processing Units

Robots adopt the role of EPUs in our system. They allocate a set of capabilities (C_{EPU}), enabling them to execute respective program instructions. During task allocation (performed, e.g., with our approach proposed in [13]), robots get information on which capabilities they should execute at which time (addressed with a pc) and with which parameters (p, s, and e). This guarantees that each robot has the required set of capabilities available when instructed to execute them. For the correct execution of ensemble programs in Maple, EPUs' implement the behavior defined in Alg. 1. When an EPU receives a coordination message (cf. L. 2 in Alg. 1), it needs to execute all capabilities $c \in C_{pc}$ in parallel (cf. L. 9 in Alg. 1). To derive the relevant set of capabilities C_{pc}, the EPU can access the program information with the transmitted program counter pc (cf. L. 7 in Alg. 1). We can address each $c \in C_{EPU}$ with d, s, and e to access the values defined by the ensemble programmer (e.g., $c[s]$). According to the result of $c[s]$ derived for a capability $c \in C_{pc}$, c is executed self-finishing or non-self-finishing parametrized with the respective data p derived by $c[p]$ (cf. L. 12 – L. 16 in Alg. 1). If $c[s] = \top$, the EPU can first store the result locally (e.g., a measurement derived with MEASURE(g), cf. L. 14 in Alg. 1) and second, after all other capability executions are finished, synchronize the results R_{pc} with the coordinating instance (cf. L. 18 in Alg. 1). If $c[s] = \bot$, the EPU can only start the execution of c (cf. L. 16 in Alg. 1). To generate a response to the controlling instance (cf. L. 18 in Alg. 1), the EPU needs to evaluate whether external coordination is required. This can either be because any $c[e]$ was set to $\top$ (cf. L. 11 in Alg. 1) or because $\forall c \in C_{pc}$ the ensemble programmer set $c[s]$ to $\bot$ (cf. L. 10 in Alg. 1). Non-self-finishing capabilities are only stopped again externally when the program information addressed with a follow-up pc sent in a coordination message does not again include the capabilities (cf. L. 8 in Alg. 1). This can be used to further run capabilities even without stopping their execution during synchronization (e.g., MEAS(*temperature*) can continue during multiple MOVE(*position*) executions).

C. Enriched HTN as Syntax for Ensemble Programming

To supply a domain language providing all required controlling operations to the ensemble programmer (i.e., sequential, conditional, repeated, and parallel execution), we enrich the concepts of HTN [21] following the notation of [22] and [23]. By that, we not only gain the possibility to define complex

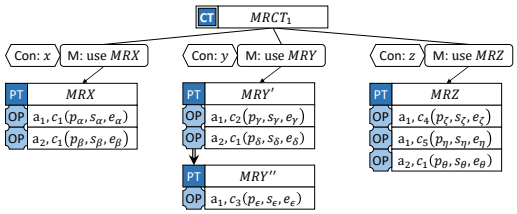

Fig. 1. Physical parallelism in MR-MT [9] involving robot a_1 and a_2.

ensemble programs but also enable the ensemble to generate new situation-aware programs through planning *during runtime* autonomously. Further, designing HTNs can be done graphically which makes them a handy programming tool even for non-technicians [24], e.g., by firefighters (cf. Section I). In our figures, we abbreviate *compound tasks* as *CT*, *primitive tasks* as *PT*, decomposition possibilities through *methods* as *M*, *conditions* for decompositions as *Con*, and *operators* encapsulating actions as *OP*. We indicate sequences of tasks as double line arrows and decompositions of CTs as single line arrows marked with their respective methods and conditions.

1) Associating Instructions to EPUs: To enable ensemble programming using EPUs and instructions from $\mathcal{I}$, we further enrich the operators concept of [23]. In Maple, OPs address a specific robot (i.e., EPU) and its respective capability (i.e., an instruction from $\mathcal{I}$), the ensemble programmer wants the robot to execute. Using OPs, the ensemble programmer can instruct robots what to do under certain circumstances.

a) Logical Parallelism (SR-MT): For a single robot, e.g., robot a_1 in Fig. 1 (where we neglect a_2 for the moment), a pc can contain single instructions (cf. if condition x holds when evaluated in the world state, robot a_1 is instructed to execute capability c_1 with parameters $[p_\alpha, s_\alpha, e_\alpha]$ in PT *MRX*), sequences of such (cf. if y holds, we use *M: use MRY*, causing robot a_1 to first execute capability c_2 with $[p_\gamma, s_\gamma, e_\gamma]$ and then c_3 with $[p_\epsilon, s_\epsilon, e_\epsilon]$), multiple instructions (cf. if z holds, we use *M: use MRZ*, causing robot a_1 to execute capabilities c_4 with $[p_\zeta, s_\zeta, e_\zeta]$ and c_5 with $[p_\eta, s_\eta, e_\eta]$ in parallel), or any combination of the aforementioned. From that information, program information EPU_I can be derived for each robot involved, that is necessary for executing EPU programs (cf. Alg. 1 with Table I for robot a_1 in Fig. 1).

b) Physical Parallelism (MR-MT): a As we want to support physical parallelism for ensemble programming, we can also include multiple agents in OPs to define multi-robot tasks. When we revise Fig. 1 (now including a_2 in our considerations), a_1 and a_2 both need to work on *MRX*, *MRY'*, and *MRZ*. In this case, we derive EPU_I for *two* agents from the plan (cf. Table I) that need to work together (it can be n agents in other programs). If one robot is not instructed in one PT, e.g., a_2 in *MRY"*, this agent should explicitly do nothing while other agents potentially execute instructions (i.e., a_2 should wait while a_1 performs c_3 in *MRY"*). Multiple robots can execute the same instruction with the same or other parameters in the same PT (i.e., physically parallel) or different PTs (e.g., *MRX* in Fig. 1). Thereby, parameters s and e can be independently programmed for all operators. This requires

Alternatives	EPU a:INT	Step pc:INT	capability c:$\mathcal{I}_{ID} \Rightarrow$ [ANY, BOOL, BOOL]
use MRX /	a_1	1	$c_1 \Rightarrow [p_\alpha, s_\alpha, e_\alpha]$
use MRX$_\infty$	a_2	1	$c_1 \Rightarrow [p_\beta, s_\beta, e_\beta]$
use MRY /	a_1	1	$c_2 \Rightarrow [p_\gamma, s_\gamma, e_\gamma]$
use MRY$_\infty$		2	$c_3 \Rightarrow [p_\epsilon, s_\epsilon, e_\epsilon]$
	a_2	1	$c_1 \Rightarrow [p_\delta, s_\delta, e_\delta]$
use MRZ /	a_1	1	$c_4 \Rightarrow [p_\zeta, s_\zeta, e_\zeta]$, $c_5 \Rightarrow [p_\eta, s_\eta, e_\eta]$
use MRZ$_\infty$	1_2	1	$c_1 \Rightarrow [p_\theta, s_\theta, e_\theta]$

TABLE I. EPU_I FOR a_1 AND a_2 DERIVED FROM FIG. 1.

further coordination when physical parallelism is used.

2) Programming Complex Control-flow: To meet the requirements for ensemble programming from Section II, we enrich the HTNs with a *loop* concept, concepts for commanding *world state modifications* (**W**) and *replanning* (**R**).

a) Logical Parallelism in Loops (SR-MT): When we assume the world state variables in Fig. 2 are set to $x = \top$, $y = z = \bot$ initially, planning started at the topmost *MRCT*$_\infty$ results in the following ensemble program (again, we neglect a_2 for the moment): At first, robot a_1 gets instructed to execute c_1 with parameter $[p_\alpha, s_\alpha, e_\alpha]$ repeatedly, until the termination condition t_α holds (cf. Term = t_α in Fig. 2), as second step x should be set to $\bot$, and as step three y should be set to $\top$, both in two *W* steps. As the fourth step, a new program should be generated triggered autonomously with a replanning starting at *MRCT*$_\infty$ given in *R*. Thereby, an ensemble program decomposed from *MRCT*$_\infty$ using the method *use MRY*$_\infty$ is generated. Following the conventions introduced in Section III-C1, for each alternative decomposition a program part can be generated for the respective decomposition method. Instructions contained in a sequence labeled as loop (cf. boxes with dotted lines in Fig. 2) define termination criteria as a Boolean expression on the results of capability executions, e.g., $c_1 =$ MEASURE, $p_\alpha = temperature$, $s_\alpha = \top$, $e_\alpha = \top$ with $t_\alpha : r_{c_1} > 30$ to determine that the result of a temperature measurement is greater than 30 degrees (cf. *MRX*$_\infty$ in Fig. 2). These termination criteria can be defined by the ensemble programmer for each OP that contains an instruction parametrized *to be executed in a self-finishing manner* (i.e., $s_\alpha, s_\gamma, s_\epsilon, s_\zeta$, and s_η need to equal $\top$). Self-finishing is a requirement for instructions used in termination functions because EPUs need to determine the result of the respective capability execution for sending it to the coordination instance which then can evaluate the particular termination criteria (cf. L. 14 and L. 16 in Alg. 1). Because the EPU itself does not need to evaluate the termination criteria, the EPU_I for a_1 in Fig. 2 is equal to that for a_1 in Fig. 1 (cf. Table II). Loop constructs can also be used nested: When *MRCT*$_\infty$ is decomposed using the method *use MRY*$_\infty$, a_1 first needs to repeatedly execute c_2 until t_γ holds (determined by the controlling instance), as a second step execute c_3, and return to the first step if t_ϵ does not hold or otherwise continue. Further, in a termination criteria the programmer can also involve the ensemble's user (cf. Term: $(t_\zeta \vee t_\eta) \wedge t_\theta \wedge u$ in Fig. 2). Consequently, the loop can only be terminated if the user acknowledges the termination.

b) Physical Parallelism in Loops (MR-MT): The loop concept can also be used for multi-robot tasks (cf. Fig. 2 with respective EPU_I in Table I). In this case, termination functions annotated to loops in the ensemble program can contain more than one agent. When we revise Fig. 2 (now

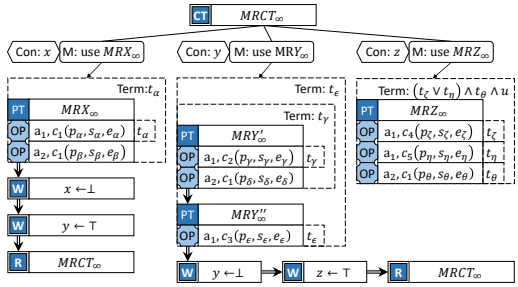

Fig. 2. Using repeated execution for MR-MT [9]. We assume that initially (planning time) variable in the world state are set to $x = \top, y = \bot, z = \bot$.

including a_2 in our considerations), we see that if *MRCT*$_\infty$ is decomposed with method *use MRZ*$_\infty$, the termination of the loop is only achieved, if results determined by capability executions of a_1 and a_2 combined with the boolean operators get evaluated to $\top$ (i.e., $(t_\zeta \vee t_\eta) \wedge t_\theta \wedge u = \top$). Of course, programming sequences with loops involving more than one EPUs where a single instruction for one EPU decides the termination function of the loop causes all other EPUs to be idle for that time. An example therefore is the nested loop included in Fig. 2 when *MRCT*$_\infty$ is decomposed with method *use MRY*$_\infty$, where every execution of instructions from *MRY*$''_\infty$ needs a_2 to wait for the decision determined by the result of a_1. If the ensemble programmer does not intend this behavior, other design decisions for the ensemble program have to be made. Instructions concerning world state modifications and replanning are not contained in EPU_I for both, single-robot tasks and multi-robot tasks. Like the evaluation of termination criteria, they require a superior instance capable of aggregating data-flow and deciding on control-flow (cf. Section II).

3) Control-Flow Coordination: After a successfully performed task allocation including the distribution of respective EPU_I, a controlling instance needs to coordinate the control and data-flow within its ensemble. This is especially urgent for controlling physically parallel executed instructions. If required, also distributed information can get aggregated, e.g., for deciding on the termination of loops. The controlling instance is also needed for world state modifications defined within an ensemble program (cf. Fig. 2). Further, the controlling instance is needed when replanning, i.e., autonomous online program generation, should be performed autonomously. Therefore, one EPU in the ensemble needs to adopt the role of such a controlling instance for every generated plan.

a) Coordinating the Control-Flow: For all controlling functions, we propose to implement the program depicted in Alg. 2. As supplement to generated EPU_I (cf. Section III-B), we also generate control-flow program information CFP_I for each plan derived from an HTN (e.g., Table II for information derived from possible plans in Fig. 2). This CFP_I includes information on participating EPUs as a set of robots A, as well as control-flow and data-flow information referenced by a *pc*. CFP_I are made available to the coordination instance after task allocation (cf. L. 4 in Alg. 2). These *pc*s reference different *types* of functionality the controlling

Alg. 2 CF PROGRAM

```
1: user; u ← ⊥; u_r ← ⊥; t ← NULL
2: plan ← ∅; pc ← 0
3: procedure AS:START(CFP_I)
4:   t ← CFP_I; pc ← 0
5:   EXECUTE()
6: procedure EXECUTE
7:   SELECTPC()
8:   switch t.type[pc] do
9:     case EX
10:      A_w ← ∅; e ← ⊥; s ← ⊤
11:      u ← ⊥; u_r ← ⊥
12:      COORDINATE
13:    case STORE
14:      UPDATESTORE(t.exp[pc])
15:      EXECUTE
16:    case PLAN
17:      plan ← plan ∪ CREATE(t.exp[pc])
18:      EXECUTE
19:    case FINISH
20:      while ∃t_i ∈ plan|t_i.A = ∅ do
21:        INTEGRATEPCS(t_i, t)
22:        plan ← plan \ t_i
23:      if ∃t_i ∈ plan|t_i = t then
24:        plan ← plan \ t_i
25:        ( BROAD(plan) ‖ START(t_i, SELF)↑ )
26:      else
27:        ( BROAD(plan) ‖ COORDINATE )
28: procedure COORDINATE
29:   for a ∈ t.A do
30:     a.COORDINATION(pc, SELF)↑
31: procedure SELECTPC
32:   if t.type[pc] = EX then
33:     if u ∈ T_A then
34:       if u_r = ⊥ then
35:         u_r ← ⊤
36:         user.REQUESTU↑
37:       if EVALUATE(t.exp[pc], R_A) then
38:         pc ← t.trans[pc].FIRST
39:       else
40:         pc ← t.trans[pc].SECOND
41:   else
42:     pc ← t.trans[pc].FIRST
43: procedure AS:SYNC(s_a, e_a, R_a)
44:   A_w ← A_w ∪ ℂ; R_A[a] ← R_a
45:   e ← (e ∧ e_a); s ← (s ∨ s_a);
46:   if t.A ⊂ A_wait then
47:     if e ∨ ¬s then
48:       user.REQUESTCOORDINATION(SELF)↓
49:     EXECUTE
50: procedure AS:RECEIVEU
51:   u ← ⊤
```

instance offers: EX requires the ensemble's EPUs to execute capabilities referenced by the current pc, STORE variable modifications in the storage, PLAN replanning in the HTN, and FINISH the dissolving the ensemble. For each type of pc, the relevant instruction exp differs. In EX, the EXPRESSION in exp encodes the termination criteria over the results determined by $a \in A$ (cf. L. 37 in Alg. 2), STORE encodes instructions on how to modify the storage (cf. L. 14 in Alg. 2), and in PLAN *exp* references a CT in the HTN where replanning should be started (cf. L. 17 in Alg. 2). For FINISH, no instruction is needed as this functionality does not differ in different ensemble programs. Further, for each pc an CFP_I encodes a transition function (*trans*), encoding follow-up pcs for conditional transitions in the control-flow, e.g., loops.

b) Determining Program Counters: When executing Alg. 2, first the next pc is determined (cf. L. 7, L. 31 in Alg. 2). In cases, where the *type* of the pc is not EX, this is straight forward, i.e., the first and second pc in the *trans* are the same (cf. L. 42 in Alg. 2). For pc of type EX, the controlling evaluates the result of EVALUATE$(t.exp[pc], R_A)$ to decide, whether the first or the second entry is relevant. If exp includes the user, EVALUATE$(t.exp[pc], R_A)$ (cf. L. 37 in Alg. 2) can not result in $\top$ until the responded (cf.L. 50 in Alg. 2) to a priorly REQUESTU message (cf. L. 36 in Alg. 2). Boolean flags (cf. L. 2 in Alg. 2) indicate whether this answer is already requested (u_r, a request should only be sent once) and if the answer was received (u). If the user responded or is not involved in exp at all, the coordination instance can decide on the next pc according to the result of EVALUATE$(t.exp[pc], R_A)$) (cf. L. 38 and L. 40 in Alg. 2).

c) Coordinating Parallelism: The program in Alg. 2 interacts with EPU programs (cf. Alg. 1) by sending COORDINATION(pc) messages (cf. L. 30 and L. 2) and receiving SYNC(s_a, e_a, R_a) messages (cf. L. 43 and L. 18 in Alg. 2 to control physically parallel executions. A COORDINATION(pc) message triggers a SYNC(s_a, e_a, R_a) response from the respective EPU, independent of which EPU is addressed and how the referenced instructions are parametrized. An EPU responds to (cf. L. 18 in Alg. 1) either when it has finished to execute self-finishing capabilities (cf. L. 14 in Alg. 1), has started the execution of non-self-finishing capabilities (cf. L. 16 in Alg. 1), or was not instructed to execute instructions at all (i.e., $C_{pc} = \emptyset$ in L. 9 in Alg. 1). This guarantees the synchronization of the ensemble, enabling starting or stopping of physically parallel execution. When any agent's response calls for user coordination ($\exists a \in A : e_a = \top$) or all agents require external coordination for terminating their capability execution ($\forall a \in A : s_a = \top$), the user needs to be involved (cf. L. 48 in Alg. 2 - we wait for the user with $\downarrow$).

Alternatives	**A**: SET	**pc**: INT	**type**: ENUM	**exp**: EXPRESSION	**trans**: TUPLE
use MRX_∞	$\{a_1, a_2\}$	1	EX	$t_\alpha \wedge u$	2, 1
		2	STORE	$x \leftarrow \bot$	3, 3
		3	STORE	$y \leftarrow \top$	4, 4
		4	PLAN	$MRCT_\infty$	5, 5
		5	FINISH	-	-
use MRY_∞	$\{a_1, a_2\}$	1	EX	t_γ	2, 1
		2	EX	t_ϵ	3, 1
		3	STORE	$y \leftarrow \bot$	4, 4
		4	STORE	$z \leftarrow \top$	5, 5
		5	PLAN	$MRCT_\infty$	6, 6
		6	FINISH	-	-
use MRZ_∞	$\{a_1, a_2\}$	1	EX	$(t_\zeta \vee t_\eta) \wedge t_\theta$	2, 1
		2	FINISH	-	-

TABLE II. CFP_I DERIVED FROM FIG. 2.

d) Replanning and Finishing Ensemble Programs: When the program's execution is finished, the controlling instance sends a final coordination message to all EPUs including a pc indicating the termination (cf. L. 27 in Alg. 2). When the *type* of a previous pc was PLAN, the coordinating instance generated new combinations of EPU_I and CFP_I from the result of replanning started from the referenced CT and bundles them as a *plan* (cf. L. 17 in Alg. 2). To avoid overhead caused by ensemble formation, the controlling instance analyzes all newly generated plans before broadcasting them in the system. If there exists an CFP_I where no EPUs are required (i.e., only S or R are contained), the controlling instance can integrate it in its current CFP_I before finishing the program (cf. L. 20 in Alg. 2). The same holds for such plans with CFP_I equal to that the controlling instance has activated currently (cf. L. 23 in Alg. 2). Instead of broadcasting that plan, the controlling instance resets its current pc and restarts the execution of the current program with the same ensemble (cf. L. 25 in Alg. 2).

IV. RELATED WORK

To hide the complexity of the coordination between entities in an ensemble and avoid their individual programming, approaches like Meld [1] and Protelis [2] abstract from the individual in the ensemble and aim at programming on an aggregate level. Protelis [2] provides clear benefits to programmers of ensembles that need to derive and process distributed data. By using the paradigm of spatial computing, focusing on homogeneous devices, aggregate evaluations are easy to do. Its usage for commanding robots, especially for heterogeneous and mobile ones, has not yet been demonstrated, which we think is justified by the different idea of Protelis, how to use the ensemble (that is aggregating information and processing it). Further, as far as we can see in Protelis there is no possibility for autonomous online program generation like in our approach. Meld [1] provides other benefits to an ensemble programmer. With its logic programming approach Meld can generate complex programs from a minimal fact set which needs to be defined by the programmer. On the downside, generated programs are hard to comprehend and retrace,

and there is no existing solution for compositionally reusing partial programs. Up to, Meld also lacks a demonstration of real-world usage. Instead, Meld focuses on other types of ensembles than that we want to control with our approach: It is used for abstractly modeled, simulated, self-shaping modular robots in large scale ensembles (more than one million entities) and thereby is no alternative for commanding mobile-robots in the real world used for catastrophe scenarios. Another approach for programming swarms is Buzz [3] that aims at controlling multi-robot systems. With virtual stigmergy and a neighborhood concept, Buzz provides concepts for common swarm behaviors but supports only reduced controlling of individual action. A domain specific language aiming at orchestrating such individual tasks as well as multi-robot tasks is Dolphin [4]. Like in our approach, in Dolphin the user of the system should be able to define programs without in-depth technical knowledge of the system and in addition, also is intended to actively take part in the ensemble's execution (user in the loop). Dolphin lacks support for ensemble or swarm operations and autonomous plan/task generation.

V. Conclusion

With our ensemble programming language approach Maple, ensembles become easily programmable in a graphical way. Enabled by our system's multipotency, programmers do not have to take into account the current robot configuration during programming, but can focus on the actual problem (*what* to do *when* and *how*). While in the scope of this paper we are only able to demonstrate the general concept of Maple, we already successfully applied it to several scenarios. This enabled the generation of situation-aware programs for our ensembles directly from the programmer's specification. Besides, we can increase ensembles if necessary by adding new operators (we provide an approach to freely combine capabilities from existing capabilities at runtime [14]). Thus, with Maple, also non-experts in technical programming can define ensemble programs, e.g., firefighters in catastrophe scenarios. We further will enable swarm behavior in the future [25], [26]. Further, we will focus on dealing with uncertainties in execution by integrating appropriate self-organization mechanisms to increase the system's autonomy, which is an urgent topic when aiming at real-world applicability.

Acknowledgement

This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 402956354.

References

[1] M. P. Ashley-Rollman, P. Lee, S. C. Goldstein *et al.*, "A language for large ensembles of independently executing nodes," in *Logic Programming*, P. M. Hill and D. S. Warren, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 265–280.

[2] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *Proc. of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. ACM, 2015, pp. 1846–1853. [Online]. Available: http://doi.acm.org/10.1145/2695664.2695913

[3] C. Pinciroli and G. Beltrame, "Buzz: An extensible programming language for heterogeneous swarm robotics," in *2016 IEEE/RSJ Int. Conf. on Intel. Robots and Systems (IROS)*, Oct 2016, pp. 3794–3800.

[4] K. Lima, E. R. Marques, J. Pinto *et al.*, "Dolphin: a task orchestration language for autonomous vehicle networks," in *IEEE/RSJ Int. Conf. on Intel. Robots and Systems (IROS)*. IEEE, 2018, pp. 603–610.

[5] M. Dorigo, D. Floreano, L. M. Gambardella *et al.*, "Swarmanoid: A novel concept for the study of heterogeneous robotic swarms," *IEEE RAM*, vol. 20, no. 4, pp. 60–71, 2013.

[6] R. R. Murphy, S. Tadokoro, D. Nardi *et al.*, *Search and Rescue Robotics*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1151–1173. [Online]. Available: https://doi.org/10.1007/978-3-540-30301-5_51

[7] L. Marconi, S. Leutenegger, S. Lynen *et al.*, "Ground and aerial robots as an aid to alpine search and rescue: Initial sherpa outcomes," in *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE Int. symposium on*. IEEE, 2013, pp. 1–2.

[8] E. F. Flushing, L. M. Gambardella, and G. A. D. Caro, "A mathematical programming approach to collaborative missions with heterogeneous teams," in *2014 IEEE/RSJ Int. Conf. on Intell. Robots and Systems*, 2014, pp. 396–403.

[9] B. P. Gerkey and M. J. Matarić, "A formal analysis and taxonomy of task allocation in multi-robot systems," *The Int. Journ. of Robotics Res.*, vol. 23, no. 9, pp. 939–954, 2004.

[10] O. Kosak, "A decentralised swarm approach for mobile robot-systems," in *Organic Computing: Doctoral Dissertation Colloquium 2015*, vol. 7. kassel university press GmbH, 2015, p. 53.

[11] O. Kosak, C. Wanninger, A. Angerer *et al.*, "Decentralized coordination of heterogeneous ensembles using jadex," in *IEEE 1st Int. Workshops on Found. and Appl. of Self* Systems (FAS*W)*, 2016, pp. 271–272.

[12] B. Wolf, C. Chwala, B. Fersch *et al.*, "The scalex campaign: Scale-crossing land surface and boundary layer processes in the tereno-prealpine observatory," *Bulletin of the American Meteorological Society*, vol. 98, no. 6, pp. 1217–1234, 2017. [Online]. Available: https://doi.org/10.1175/BAMS-D-15-00277.1

[13] J. Hanke, O. Kosak, A. Schiendorfer *et al.*, "Self-organized resource allocation for reconfigurable robot ensembles," in *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, Sep. 2018, pp. 110–119.

[14] C. Wanninger, C. Eymüller, A. Hoffmann *et al.*, "Synthesising Capabilities for Collective Adaptive Systems from Self-Descriptive Hardware Devices - Bridging the Reality Gap," in *8th Int. Symp. On Leveraging Appl. of Formal Methods, Verification and Validation*, Sept 2018.

[15] O. Kosak, C. Wanninger, A. Hoffmann *et al.*, "Multipotent systems: Combining planning, self-organization, and reconfiguration in modular robot ensembles," *Sensors*, vol. 19, no. 1, 2018. [Online]. Available: http://www.mdpi.com/1424-8220/19/1/17

[16] H. Hamann, Y. Khaluf, J. Botev *et al.*, "Hybrid societies: Challenges and perspectives in the design of collective behavior in self-organizing systems," *Frontiers in Robotics and AI*, vol. 3, p. 14, 2016. [Online]. Available: http://journal.frontiersin.org/article/10.3389/frobt.2016.00014

[17] S. Ghosh, *Distributed systems: an algorithmic approach*. Chapman and Hall/CRC, 2014.

[18] M. Brambilla, E. Ferrante, M. Birattari *et al.*, "Swarm robotics: a review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013.

[19] M. B. Dias, R. Zlot, N. Kalra *et al.*, "Market-based multirobot coordination: A survey and analysis," *Proc. of the IEEE*, vol. 94, no. 7, pp. 1257–1270, July 2006.

[20] A. Hussein, M. Adel, M. Bakr *et al.*, "Multi-robot task allocation for search and rescue missions," *Journ. of Physics: Conf. Series*, vol. 570, no. 5, p. 052006, 2014. [Online]. Available: http://stacks.iop.org/1742-6596/570/i=5/a=052006

[21] K. Erol, J. Hendler, and D. S. Nau, "Htn planning: Complexity and expressivity," in *AAAI*, vol. 94, 1994, pp. 1123–1128.

[22] T. Humphreys, "Exploring htn planners through examples," *Game AI pro: Collected wisdom of game AI professionals*, vol. 149, 2013.

[23] D. Nau, "Game applications of htn planning with state variables," in *Planning in Games: Papers from the ICAPS Workshop*, 2013.

[24] D. Bau, J. Gray, C. Kelleher *et al.*, "Learnable programming," *Communications of the ACM*, vol. 60, no. 6, p. 72–80, May 2017. [Online]. Available: http://dx.doi.org/10.1145/3015455

[25] O. Kosak, "Facilitating planning by using self-organization," in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Sept 2017, pp. 371–374.

[26] O. Kosak, "Multipotent systems: A new paradigm for multi-robot applications," in *Organic Computing: Doctoral Dissertation Colloquium 2018*, vol. 10. kassel university press GmbH, 2018, p. 53.