Machine-learned Behaviour Models for a Distributed Behaviour Repository

Alexander Jahl, Harun Baraki, Stefan Jakob, Malte Fax and Kurt Geihs
Distributed Systems Department, University of Kassel, Wilhelmshöher Allee, Kassel, Germany

Keywords: Multi-agent Systems, Autonomous Systems, Self Organizing Systems, Agent Models and Architectures, Task Planning and Execution.

Abstract: Dynamically organised multi-agent systems that consist of heterogeneous participants require cooperation to fulfil complex tasks. Such tasks are commonly subdivided into subtasks that have to be executed by individual agents. The necessary teamwork demands coordination of the involved team members. In contrast to typical approaches like agent-centric and organisation-centric views, our solution is based on the task-centric view and thus contains active task components which select agents focusing on their Skills. It enables an encapsulated description of the task flow and its requirements including team cooperation, organisation, and location-independent allocation processes. Besides agent properties that represent syntactical and semantic information, agent behaviours are considered as well. The main contributions of this paper are hyperplane-based machine-learned Behaviour Models that are generated to capture the behaviour and consider the Behaviour Implementations as black boxes. These Behaviour Models are provided by a distributed behaviour repository that enables tasks to actively select fitting Behaviour Implementations. We evaluated our approach based on agents playing chessboard-like games autonomously.

1 INTRODUCTION

Multi-agent systems may consist of heterogeneous agents that act autonomously in their environment. They are equipped with various actuators, sensors, and executable routines which all can be represented by software or hardware components. Such routines are software programs that are performed at runtime, thus termed as runtime behaviours. While simple tasks are executed by single agents with appropriate hardware and runtime behaviours, complex tasks can be thought as workflows of subtasks. Thus, they ask for cooperation and coordination of several agents. Typical concepts which address this issue apply either the agent-centric or the organisation-centric viewpoint (Picard et al., 2009). In the case of the agent-centric view, an explicit definition of the workflow is neither given to the agents nor a central entity. Instead, the coordination mechanisms are embedded into subtasks which are performed by assigned agents. In contrast, the organisation-centric view applies predefined roles and coordination structures that are either part of the knowledge of each agent or managed by a central coordinator. In (Jahl et al., 2021), we introduce the task-centric view, which is employed in this work. In contrast to the previous views, this view considers a task as a separate active component that proactively searches for an executing unit that is able to fulfil its requirements. A unit may represent pure software as well as physical agents. Anyhow a unit can be any interactive software component with well-defined interfaces such as a microservice or a shell to command devices. Since the management of tasks and the executing units are separated and not predefined, neither any structural specification nor a set of tasks has to be provided at start time. Tasks may emerge at run time and executing units are not tailored to specific tasks only. In the case of a given structural specification, our approach maps to the organisation-centric view. If the set of tasks is predefined and the allocation of executing units is unique, our approach resembles the agent-centric view.

In our task-centric view, a complex task is described by a plan that is not active. Instead, it provides restrictions for the active tasks, which include minimum and maximum cardinalities of executed instances and the required capabilities to perform its contained tasks. Furthermore, tasks in a plan can depend on further plans, such that a hierarchical management is achieved. In Figure 1, $\mathcal{T}ask_1$ is linked

Jahl, A., Baraki, H., Jakob, S., Fax, M. and Geihs, K.
Machine-learned Behaviour Models for a Distributed Behaviour Repository.
DOI: 10.5220/0010804000003116
In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022) - Volume 1*, pages 188-199
ISBN: 978-989-758-547-0; ISSN: 2184-433X

to the additional $\mathcal{P}lan_f$ whose tasks have to be executed first. Initially, a plan is submitted to an arbi-

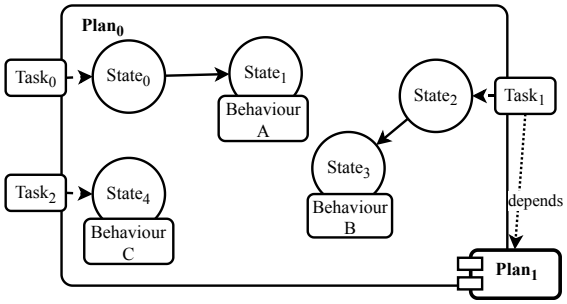


Figure 1: Hierarchical plan structure.

trary unit of the multi-agent system, which extracts the tasks and annotates them with plan information. Subsequently, the extracted tasks start their search for a fitting unit. If a unit is selected, the task enters the initial state. A state change occurs in case a transition condition is met. States can be equipped with runtime behaviours. Behaviour implementations are encapsulated domain-specific software routines. We consider them as black boxes that provide their functionalities through interfaces described in YAML syntax and denoted as *Behaviour*. Depending on the task and the fitting unit, different behaviour implementations may perform better or worse. To ensure the best possible execution, a mechanism to select an appropriate implementation is required.

The main contribution of this work are self-optimising tasks that may iteratively improve their performance by capturing, sharing, searching, and applying different behaviour implementations. A straightforward solution would apply a registry that maps Behaviours to available behaviour implementations. However, relying only on interface information may lead to several issues. Interfaces comprise structural information that restricts data types for input and output and function names. Choosing alternative behaviour implementations by only constraining interface structures could result in misinterpretations of data types and operations. For example, a returned floating-point number might hold a temperature in Fahrenheit, Celsius, or Kelvin. In this situation, developers may provide semantic annotations. However, semantics-based approaches rely on annotations by developers and a common top-level ontology. This shifts the burden towards the developers. Moreover, it requires their willingness to agree on and adhere to a common ontology. Furthermore, these annotations may be incomplete and outdated. Our solution contains machine-learned behaviour models based on a black-box view. We apply machine learning algorithms to gain abstraction of the transformations of input and output data. The resulting machine-learned models represent the behaviour implementation. This enables tasks to detect the degree of similarity between behaviour implementations.

The remainder of this paper is structured as follows. Section 2 introduces the formal definition of our Skill concept. Section 3 describes the applied classification mechanisms and the architecture of our Distributed Behaviour Repository. In Section 4 we outline firstly an application scenario and our multi-agent framework that is used for the implementation and subsequently in the evaluation. Related work is presented in Section 5. Finally, the main contribution of this paper is summarised and concluded in Section 6.

2 UNIT-SKILL-TASK CONCEPT

In our *task-centric* view (Jahl et al., 2021), a *Skill Unit* is defined as an abstract representation of an agent including its specific skills. Skill Units are allocated to *Task*s. The definition of a Skill Unit is formed by combining one or more *Primitive* and *Complex Skill*s. These can be added, removed or altered at runtime. Thus, the set of Skills can be dynamically adapted to changing environments. Furthermore, subdividing the Skills into different types supports the maintainability and enables a more precise mapping of the different properties of a unit as well as the separation of meta information from concrete callable software routines.

To formalise this, we define a Skill Unit u as a tuple of an agent a and a set of Skills:

$$u = (a, \mathcal{S}_u) \text{ where}$$
$$\mathcal{S}_u = \{s_f, ..., s_i\} \text{ with } s \in \mathcal{S}_p \cup \mathcal{S}_c \quad (1)$$

The set of Skills encompasses two distinct Skill types. Figure 2 illustrates the Skill concept of both Skill types.

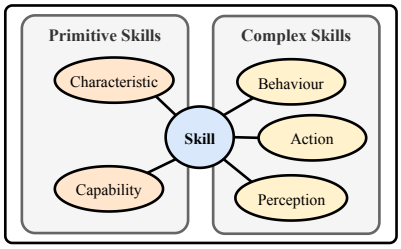


Figure 2: Skill classification.

Primitive Skills. The first type are Primitive Skills $\mathcal{S}_p$ which contain set of characteristics s_{ch} and capabilities s_{cap}:

$$\mathcal{S}_p = \{s_{p_f}, ..., s_{p_n}\} \quad (2)$$

Characteristics $s_{ch} = (key, value)$ comprise inherent properties like number of CPU cores, capacity of storage, memory size and additionally environmental information like the location.

Capabilities $s_{cap} = (string)$ represent semantic information about Complex Skills of an agent, such as *canMove*, *canSend*, *canReceive*.

Complex Skills. The second type are Complex Skills $\mathcal{S}_c$ which include set of runtime behaviours s_b, actions s_a, and perceptions s_e.

$$\mathcal{S}_c = \{s_{c_1}, ..., s_{c_n}\} \tag{3}$$

Actions $s_a = act(\mathcal{V})$ with $\mathcal{V} = \{v_1, ..., v_n\}$, describe simple functions passing a set $\mathcal{V}$ of values v_x as parameters.

Perceptions $s_e = perc(\mathcal{E})$ with $\mathcal{E} = \{e_1, ..., e_n\}$ that is a set of environmental perceptions provided by the sensors of the agent.

Behaviours s_b are linked to Capabilities and contain the definition of an interface, which comprises the key commands start, stop, terminate, and Actions and Perceptions calls. Furthermore, a State Graph $\mathcal{G}_s$ and a model representation $\mathcal{M}$ can be included.

$$\begin{aligned} s_b &= (\mathcal{S}_x, \mathcal{G}_s, \mathcal{M}) \\ \mathcal{S}_x &= \{s_1, ..., s_n\} \text{ with } s \in \mathcal{S}_{cap} \cup \mathcal{S}_e \cup \mathcal{S}_a \end{aligned} \tag{4}$$

Active Tasks. Finally, a Plan p includes a set of Tasks $\mathcal{T}$. Each *Task* t is defined as a tuple of a State Graph $\mathcal{G}_s$ and a set of required Skills $\mathcal{S}_r$. A State Graph represents a tree of finite state machines.

$$\begin{aligned} t &= (\mathcal{G}_s, \mathcal{S}_r) \\ \mathcal{S}_r &= \{s_r \mid s_r \in \mathcal{S}_p \cup \mathcal{S}_c\} \end{aligned} \tag{5}$$

In order to execute the Task, the preconditions formed by the required Skill set $\mathcal{S}_r$ have to be fulfilled.

3 RUNTIME BEHAVIOURS

Runtime behaviours are combinations of Behaviours and Behaviour Implementations. While Behaviours represent syntactic and semantical descriptions of the interface, Behaviour Implementations encapsulate domain-specific software routines.

3.1 Behaviours

As mentioned in the introduction, the tasks with their annotated plan information are submitted to an arbitrary Skill Unit in the system. Subsequently, the tasks are extracted and start their search for a fitting Skill Unit. In order to find a suitable unit, the task translates their descriptions of the Behaviour, which encapsulate Primitive and Complex Skills into a logic program. The descriptions are defined in YAML syntax by default. An example is shown in Listing 1.

```yaml
requirements:
  action:
    name: "storeData"
    parameter: list
    returnType: string
  ...
  characteristic:
    name: "storeCapacity"
    value: 20
    valueType: integer
  ...
```

Listing 1: Skill Unit YAML description.

The Listing contains an action and a characteristic as requirements for the task. For example the action `storeData` has a `list` as parameter and a `string` as return value. The characteristic is modelled in an analogous way. YAML is a standard solution for an efficient description of knowledge in a readable form. However, it does not provide any reasoning support to match the required Skills of the tasks with the provided Skills of the current Skill Unit. A suitable knowledge representation and reasoning formalism is *Answer Set Programming* (ASP) (Gebser et al., 2012). ASP provides non-monotonic reasoning and supports the definition of defaults. An ASP program consists of *rules* that are divided into two parts which are the *head* and the *body* and are separated by the deduction symbol `:-`. In general, the head of a rule is derived if all *literals* of the body holds. A literal is a statement that can be true or false. Furthermore, a rule without a body is considered as a fact since it is unconditionally true.

```
action("storeData").
parameter("storeData", list).
returnType("storeData", string).
...
characteristic("storeCapacity").
value("storeCapacity", 20).
valueType("storeCapacity", integer).
...
```

Listing 2: Skill Unit ASP description.

Listing 2 shows the translation of the requirements in Listing 1. All requirements are translated into

facts. The Skill Unit provides its Skills as ASP facts marked by a prefix `unit_` . To detect a missing requirement that is demanded by the task but not provided by the Skill Unit, a rule is added to the ASP program for each requirement of the task. Listing 3 illustrates an excerpt of these rules.

```
1  missing(REQ) :- action(REQ),
       not unit_action(REQ).
2  wrong_param(REQ, PARAM) :-
       parameter(REQ, PARAM),
       not unit_parameter(REQ, PARAM).
3  ...
```

Listing 3: Matching rules.

Informally speaking, the head of the rule in Line 1 is derived if an action is required but not provided by the unit. The results of the ASP program are provided by an Answer Set. It contains the minimal set of literals which includes all facts and the derived rules. If all requirements are met, the Answer Set only contains the initial facts.

3.2 Behaviour Implementation

In order to generate models of the Behaviour Implementation, the Skill Unit monitors the corresponding input and output data. Afterwards, the Skill Unit applies machine learning algorithms on the observed input and output data to create a detailed description that captures the behaviour at runtime. Thus, a classification and comparability of the runtime behaviours in different tasks is feasible and a potential future optimisation of the task flow is supported.

For example, One-Class Support Vector Machines (OC SVM) learn a hyperplane that can be used to classify the correct inputs and outputs. The application of hyperplane-based classifiers ensures interpretable and transferable models which can be easily compared. Skill Units apply One-class (OC) classification since it is particularly suited for a fast generation of a compact representation model of the Behaviour Implementation. In order to distinguish several Behaviour Implementations, Multi-class (MC) classification can be employed. Several strategies exist to reduce the complexity of MC-classification. For example, multiple binary classifiers can be trained, and their results can be combined to provide a final classification. In this case, One-vs-All (OvA) and One-vs-One (OvO) classifiers can be distinguished. Figure 3 shows an overview of the considered classifiers. Both, OvA classifiers and OvO classifiers create a model for each class. In contrast to the OvO classifier, the OvA classifier enables the subsequent separation of data according to their class.

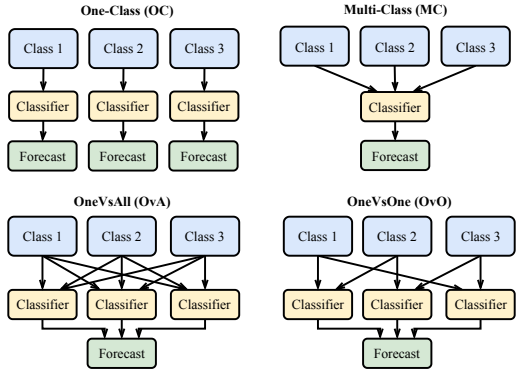


Figure 3: Multi vs. binary vs. unary classification.

OC classification approaches employ unary classifiers that do not assign data points to multiple classes. Instead, they decide whether a data point belongs to a class or not. They apply a decision function that assigns a higher weight to regions with high density and thus tries to classify the area. This results in the creation of decision boundaries that can be used to classify data. A single class is established, which is a small, consistent subset of the data. Density estimation (Tarassenko et al., 1995), volume optimisation, and model reconstruction (Bishop et al., 1995) provide the underlying techniques for common OC classifiers.

Individual classification algorithms can be prone to under-fitting or over-fitting and therefore, often lead to ambiguity when assigning a classification method. In order to prevent this, models are classified by applying ensemble learning with bootstrap aggregation (bagging) and a voting classifier (Xing and Liu, 2020). The ensemble learning approach utilises a collection of classifiers that is taken into account when determining the results of each classifier. In order to produce a majority decision, the results of all classifiers are subsequently aggregated. In addition to higher accuracy, this results in a more comprehensive mapping of the classes under investigation. While all classifiers calculate the same prediction in the best case, they produce a tie in the worst case. The validity and significance of the predictions can be evaluated in this manner. Ensemble learning provides on average better results compared to the weakest classifier in the collection. However, it may calculate worse results than the best classifier.

3.3 Distributed Behaviour Repository

Each Skill Unit (*Skill Unit*) is equipped with a *Skill Manager* that is responsible for managing the learned *Behaviour Models* provided by the ensemble learn-

ing. The network of all Skill Managers forms the *Distributed Behaviour Repository*. Figure 4 depicts the Skill Manager of a Skill Unit. The aforementioned categories of classifiers take effect in each Skill Manager. The Skill Unit collects the input and output data of the current Behaviour Implementation to generate and maintain a corresponding Behaviour Model through an OC classifier. The structure of this Behaviour Model depends on the applied machine learning mechanism. For example, the usage of a Support Vector Machine (SVM) results in a hyperplane-based model. To apply MC-classification, each Skill Unit would have to collect the input and output data from all existing Skill Units, which, however, does not scale and would violate privacy concerns. After the data collection, the Skill Unit combines the Behaviour Model with the extracted *Skill Set*. The obtained Skill Set is forwarded to the Skill Manager. In the Skill Manager, already classified Behaviour Models are grouped according to their Skill Sets. The Skill Manager decomposes the incoming Skill Set and uses the extracted Skills, for example, the interface description or the location, to select all associated Behaviour Model groups. The latter ones are compared with the newly received model by applying ensemble learning.

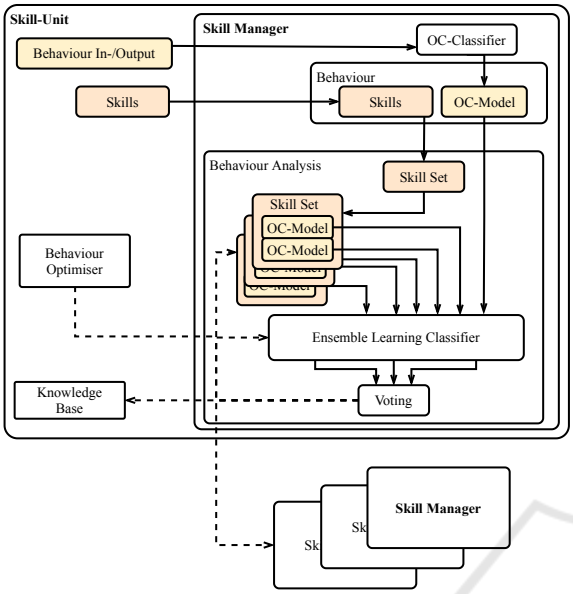


Figure 4: Skill Unit architecture.

The resulting prediction is used to narrow down the selection to at least one Behaviour Model group. If there is a high probability that the selected group will satisfy the query, an OC-classifier representing the group is trained. Afterwards, the Behaviour Model of the Skill Unit is classified with this OC-classifier. In the case of a positive result, the Behaviour Model of the Skill Unit is added to the group. The model of the classifier is returned to the Skill Unit, which in turn verifies the model with its own input and output data observations. Otherwise, the query is forwarded to the Skill Manager of semantically fitting Skill Units. Therefore, we apply our adaptive semantic routing mechanism that we developed in a previous work (Jakob et al., 2021). This mechanism is tailored for multi-agent systems in dynamic environments. In the worst case, no fitting group is found, and a new group is created on the last visited Skill Unit, and a new entry is established in the *Knowledge Base*. Any future Skill Unit that is classified into the same Behaviour Model group is added to this entry and replicates it to the Knowledge Base.

Since the Skill Sets and the input and output data of a Skill Unit are subject to changes during runtime, the Skill Managers need appropriate information to perform updates on them. The Skill Unit continuously monitors the Skill Set of its Behaviour Implementation and forwards changes to the Skill Manager. In the case of a change of a Skill, the update is propagated by means of our routing mechanism. If a Behaviour Implementation changes, which the OC-classifier of the Skill Unit detects, the change is propagated to the appropriate Skill Managers that verify the change. If the verification fails, the initial process of announcing a Behaviour Model is started again.

Over time, Behaviour Model groups are growing, and the assignment of Behaviour Models to a group is imprecise and has to be renewed. Therefore, the Skill Managers have to analyse and restructure the groups. In this process, single Behaviour Model groups are disbanded and are reintroduced to the system separately. Thus, Behaviour Models are assigned to the best fitting group, or new groups are created.

3.4 Behaviour Optimiser

Each Skill Unit is equipped with a Behaviour Optimiser that has access to the Knowledge Base and the Distributed Behaviour Repository, which is represented by the network of Skill Managers. Using the Knowledge Base, the Behaviour Optimiser can query the Skill Managers for suitable Behaviour Implementations and their features. These features are collected during the execution of the respective Behaviour Implementation and serve for calculating the performance metric. The metric is determined by the utility function of the task, which denotes the relevant features. By default, the rate of successful executions is captured. This feature is applied by a default function if no utility function is given by the task. The

default function scores the successful against the unsuccessful executions of a Behaviour Implementation. An example of a task-specific utility function is explained in the evaluation. In case a utility function is provided by the task, its set of features is compared to the captured features of the Behaviour Implementation. If they do not match, the utility function is ignored. Instead, the rate of successful executions is used. If the performance metric of the received Behaviour Implementation is higher, the current one is replaced.

4 EVALUATION

To demonstrate the power of our black-box approach to the extraction of Behaviour Models, we selected complex software routines with similar Skill Sets but different Behaviour Implementations. In our evaluation scenario, we use software agents for Skill Units that play chessboard-like games against each other. The Behaviour Implementations may distinguish themselves in the game rules they adhere to, such as Standard Chess, Chinese Chess, and Checkers, and the game strategies they apply.

The implementation of our Skill Units is based on an extended version of the ALICA framework (Skubch, 2012). ALICA stands for *A Language for Interactive Cooperative Agents* and enables modelling of agent and team behaviour. To ease the development of the agents, ALICA provides a graphical modelling tool. By means of the tool, plans with their containing tasks and the linked finite state machines can be designed. Figure 5 depicts the plan for the

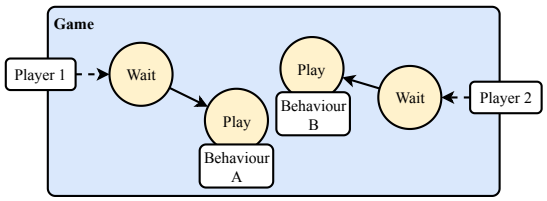


Figure 5: Game plan.

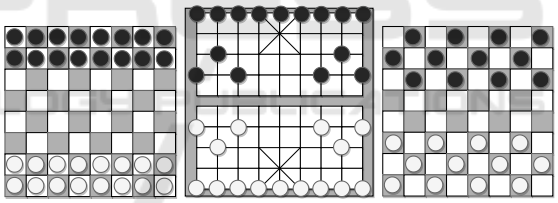
evaluation scenario. It includes two tasks where each player first waits for the other player and then starts to play. Since the Behaviour Implementations for the corresponding Behaviours is domain-specific, it has to be provided by the developer or a third party. In our case, the role of the third party is fulfilled by the Skill Unit that may overwrite the Behaviour Implementation deployed by the developer.

As mentioned before, the Skill Manager of the receiving Skill Unit extracts the tasks of the incoming plan. After approving the compliance of the syntactic and semantical descriptions of the interface provided by the task with the Skills of the Skill Unit, the Behaviour Optimiser extracts the utility function of the task, if existing, and the Behaviour Implementations with their Skill Set. The Behaviour descriptions in the Skill Set are applied to query the Distributed Behaviour Repository for alternative Behaviour Implementations and their captured features.

Considering our chessboard-like game behaviour evaluation, the utility function m calculates a score taken wins x, draws y, and losses z into account.

$$m(x, y, z) = 2 \cdot x + y - z \tag{6}$$

Furthermore, we could normalise the metric by dividing it by the number of played games. However, this could favour newly created Behaviour Implementations since they could achieve similar results to frequently executed Behaviour Implementations with a few consecutive wins. Without normalisation, the robustness of the Behaviour Implementations is considered. Hence, we do not normalise the metric. The utility function is attached to the tasks *Player 1* and *Player 2*, illustrated in Figure 5. As Behaviour Implementations, we utilise different chessboard-like game strategies. These are Standard Chess (Figure 6a), Chinese Chess (Figure 6b), and Checkers (Figure 6c). The gameboards of the considered games have similar dimensions. While Standard Chess and Checkers boards have 8x8 positions, Chinese Chess uses 10x9 positions. Furthermore, Standard Chess has 32, Chinese Chess has 30, and Checkers has 24 pawns. The course of each game is turn-based, and during each turn, a single pawn can be moved. While Standard and Chinese Chess allow moving a pawn only once in one turn, a pawn in Checkers can be moved several times if an opponent pawn is removed from the gameboard. Each game has different rules and, thus a different Behaviour.



(a) Standard (b) Chinese (c) Checkers

Figure 6: Chessboard-like games.

For the evaluation, we consider different rule implementations. We utilise four rule implementations for Standard Chess, *Laser*[1], *Stockfish*[2], *Pulse*[3], and

[1]https://github.com/jeffreyan11/uci-chess-engine
[2]https://github.com/mcostalba/Stockfish
[3]https://github.com/fluxroot/pulse

Fruit-Reloaded[4]. For Chinese Chess, two rule implementations are include, *Mars*[5] and *Elephant Eye*[6]. In the case of Checker, one rule implementation is used *Ponder*[7], since it is the only available open-source implementation. To utilise these rule implementations, we provide corresponding Behaviour Implementations. Furthermore, for simple expandability, we provide a common protocol that enables the interaction between the different Behaviour Implementations since they are encapsulated in separate processes and have to communicate via messages. These messages adhere to the following standard protocols:

UCI. Universal-Chess-Interface is a communication protocol first published by Huber and Meyer-Kahlen in (Meyer-Kahlen and Huber, 2015). The protocol allows different Standard Chess implementations to interact with a graphical user interface or with other implementations. All used Standard Chess Engines support the UCI specification.

UCCI. Universal-Chinese-Chess-Interface is implemented by the Chinese Chess Engines. The protocol adapts the UCI specification, and all commands relevant for the evaluation are directly transferable.

Table 1 presents the interaction between *Player 1* and *Player 2* during a game of Standard Chess. Player 1 utilises the Stockfish rule implementation and starts the game by moving a pawn from field `e2` to field `e4`. In every turn, the current player gets a list of all previously performed moves as input and provides an answer, including the next move as output. This results in input and output pairs that are hardly distinguishable syntactically but differ semantically.

Table 1: Game history of moves at the beginning.

Impl.	Col	Input	Output
Stockfish	w		e2e4
Laser	b	e2e4	e7e5
Stockfish	w	e2e4 e7e5	g1f3
Laser	b	e2e4 e7e5 g1f3	b8c6
Stockfish	w	e2e4 e7e5 g1f3 b8c6	f1b5
...	...	...	...

[4]https://www.chessprogramming.net/fruit-reloaded
[5]https://github.com/yytdfc/ChineseChess-engines
[6]https://github.com/xqbase/eleeye
[7]https://github.com/neo954/checkers

4.1 Classification Methods

We mainly differentiate two types of classification. The first type is context-based which means that it comprises the syntactical information about the message structure. In contrast, the second type is context-free and thus, does not have access to such kind of information. Both consider input and output data to analyse the Behaviour Implementation. Figure 7 illustrates the two types and the derived classifiers.

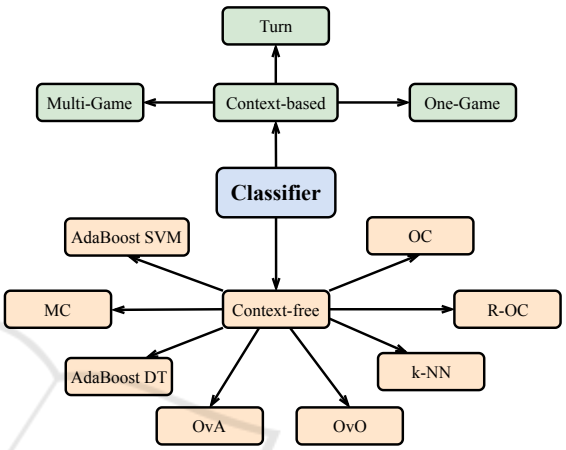


Figure 7: Relations between applied classifiers.

Context-based Classification. Context-based classifiers utilise knowledge about the structure of the input and output data. The structure-based knowledge comprises information about game turns and game rules, as well as information about the message structure. This, for instance, enables the extraction of all turns from the input and output data and a subsequent conversion of each turn from a string representation `a1a3` to an equivalent sequence of numbers `0103`. The following classifier methods apply this structural knowledge about the communication. *Turn-Classifiers* use the distribution of moves for the

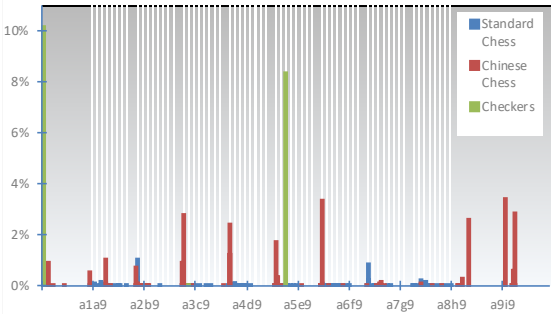


Figure 8: Move distribution grouped by game type.

classification. Therefore, each game history is divided into segments consisting of four characters that de-

note a single turn, e. g., `a0b2`. The first two characters `a0` are the start, and the second two characters `b2` are the end position of the turn. This classifier uses the course of complete matches as input. However, a single match does not provide sufficient information to learn the actual distribution of moves. Therefore, all game histories are combined, and the percentage distribution of all moves is determined. Figure 8 presents the move distribution of the utilised training data. The grey areas indicate moves that can only occur in Chinese Chess since its gameboard is bigger than Standard Chess and Checkers. Additionally to the distinction of each game type, the Turn-Classifier enables differentiation between Behaviour Implementations of a single game type. Figure 9 illustrates the resulting move distributions. Behaviour Implementations with similar strategies have similar move distributions. Again, the grey areas mark moves only available in Chinese Chess. After the classifier has

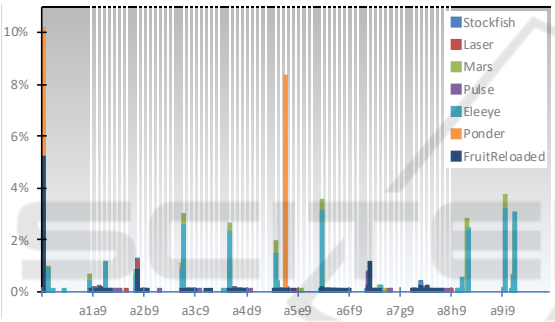


Figure 9: Move distribution of different implementations.

been trained, it compares the move distribution to a new game history. The similarity is determined by the differences to the learned frequencies. Finally, the classifier selects the Behaviour Implementation with the lowest divergence as the result. *Turn-Multi-SVM-Classifiers* employ the same training data set used to learn the Turn-Classifiers. Instead of focussing on the percentage distribution, it trains two distinct Multi-SVMs. Therefore, it focuses on white and black turns separately and determines the frequency of each move in the input and output data. By using the resulting pairs of move and percentage, two Multi-SVMs are trained. Finally, the game types and the Behaviour Implementations are used as the class labels independently for the corresponding Multi-SVM resulting in two Multi-SVMs, one for all game types and one for all Behaviour Implementations.

Turn-OC-SVM-Classifiers follow a similar strategy. Instead of training separate Multi-SVMs for game types and Behaviour Implementations, Turn-OC-SVM-Classifiers train single OC-SVMs for each game type and each Behaviour Implementation. Thus, ten OC-SVMs are created. Each OC-SVM classifies a new game history, and the best result is finally selected.

Context-free Classification. In contrast to context-based classifiers, context-free classifiers only consider input and output data streams without considering any context of the data structure. During our evaluation, we have selected several string metrics for this type of classifiers. Table 3 provides an overview of the selected string metrics. The necessary comparison string is learned by training an OC classifier model.

4.2 Experimental Results

We implemented the chessboard-like game scenario by utilising the adapted version of the ALICA framework. The experiments run on an Ubuntu server with an Intel Core i7@2.8 GHz with 4 CPU cores and 12 GB of RAM. Some of the Behaviour Implementations require Java 8 for execution. The other ones are written in C++ and compiled on the server. This evaluation focuses on the viability of our approach by combining different classification techniques and implementations. These include support-vector-based techniques, decision-tree-, and k-NN-based implementations.

During the experiment executions, 5670 matches were played, 810 for each Behaviour Implementation. Additionally, we created further Behaviour Implementations for Standard and Chinese Chess, which simulate matches against the real implementations. They create random turns which still adhere to the dimensions of the gameboard. By using these Behaviour Implementations, we produced 200 additional game histories, 100 for Standard and Chinese Chess each.

Context-based Classifier. The first experiment examines context-based behaviour classification. The evaluation framework creates Behaviour Models by utilising the context-based classifiers. The framework used the first 90 % of 810 game histories to train the classifiers and subsequently the remaining 10 % for the evaluation.

Table 2 summarises the results of the context-based classifiers. In the case of the *Turn* and the *Turn-Multi-SVM* classifier, the classification of the game type achieves high accuracy with 97.52 % and 99.9 %. However, the OC-SVM is not suited for this approach since it achieves similar accuracy, such as a random guess. The results are influenced by the specific characteristics of the game types. Chinese Chess

Table 2: Accuracy of context-based classifiers.

	Classifier		
Class. type	Turn	Multi-SVM	OC-SVM
Game Type	97.52 %	99.9 %	30.45 %
Beh. Impl.	63.86 %	78.71 %	17.08 %

allows moves that are not impossible or forbidden in the remaining game types (grey areas in Figure 8). In the case of Checkers, their matches start with similar moves and thus are distinguishable from both chess variants. Furthermore, Checkers consists of diagonal movements, such that vertical or horizontal moves are prohibited.

In general, the classifiers for the Behaviour Implementations achieve a lower accuracy since they have to consider more classes that are more similar than the classes of game types. The *Turn* classifier have an accuracy of 63.86 % and the *Turn-Multi-SVM* classifier 78.71 %. again, the OC-SVM can be compared to a random guess.

In summary, game types, as well as Behaviour Implementations, can be distinguished by applying context information. However, the classification of different Behaviour Implementations is less accurate than the classification of the game types.

Context-free Classifier. The second experiment considers context-free behaviour classification. For this experiment, we generated 10000 strings and afterwards identified the 100 best-suited strings by training an OC classifier model. The selected string are used as comparison strings for the string distance metrics.

The results are shown in Table 3. Each percentage denotes the average classification accuracy of all selected strings applied on corresponding machine learning algorithms. Again, we use 90 % of the 5670 matches to train the classifiers and the remaining 10 % for the evaluation. During this evaluation, we utilise two kinds of one-class (OC) classifiers. The first one is the Enhancing-Eta OC-SVM (Amer et al., 2013) which requires less memory and CPU. Therefore, the SVMs are enhanced by reducing the number of considered support vectors (Amer et al., 2013). Based on its support vectors, the R(econstruction)-OC-SVM is trained. The R-OC-SVM is based on the Enhancing-Eta OC-SVM, too. R-OC-SVM is an OC-SVM that learns a model using support vectors extracted from previously trained SVM models. Both achieve very similar results with a slight advantage for the R-OC-SVM. However, their accuracy is not high enough to be applicable.

In contrast to OC-classifiers, MC-classifiers achieve suitable performance consistently. Thus, the ensemble learning implementation of all Skill Managers is equipped with the evaluated MC-classifiers. Besides the Decision Tree approach, all multi-class classifiers achieve similar results and high accuracy. As expected, the accuracy is higher for game types than for Behaviour Implementations.

Evaluation Results. After several passes through the evolution process, it is obvious that our approach is able to determine the game type reliably. Standard Chess and Checkers are always detected. Checkers has turns that do not occur in both chess variants. Considering both chess variants, Chinese Chess is classified as Standard Chess in roughly 2.4 % of the cases. This is caused by the similarity of the turns of both game types.

Considering the Behaviour Optimiser, the performance metric shown in Equation (6) provides a ranking between the Behaviour Implementations. Table 4 presents the results for each game type. The Behaviour Implementation of FruitReloaded achieves the best result for Standard Chess with a metric of 419 after playing 566 games. Normalising this result would lead to a metric value of 0.79. On the other hand, Stockfish achieves a significantly higher normalised metric value of 0.84. However, it has played only 472 games. Applying the performance metric, the results consider the number of played games and thus consider the reliability of the Behaviour Implementations. This leads to the ranking of the Behaviour Implementations in Table 4.

5 RELATED WORK

In the area of multi-agent systems, several papers deal with learning of behaviours. However, only a few focus on evaluating the behaviours of agents to extract a corresponding behaviour model.

Dia et al. determine in (Dia, 2002) the behaviour of human drivers by specifying it via questionnaires. These questionnaires are used to generate representative models of human behaviour for specific routes. The applied multi-agent system integrates the created models to reduce congestions and to enhance the performance of road networks. In contrast, our approach provides an automated model generation for arbitrary agent behaviour. Furthermore, we utilise these generated models to categorise different behaviour implementations. The resulting behaviour repository is used to optimise the performance of agents during specific tasks.

Table 3: Results of Context-free Classifiers.

	MC-SVM		OC-SVM		R-OC-SVM		AdaBoost SVM	
Dist. Metric	Type	Impl.	Type	Impl.	Type	Impl.	Type	Impl.
Weighted-Levenshtein	86.13 %	53.59 %	25.57 %	10.86 %	28.41 %	9.73 %	89.38 %	55.46 %
Normal-Levenshtein	78.83 %	44.23 %	21.16 %	8.48 %	26.76 %	7.21 %	86.56 %	35.05 %
Damerau-Levenshtein	78.84 %	44.24 %	21.24 %	8.58 %	26.66 %	7.20 %	86.81 %	26.81 %
Sorensen-Dice	63.79 %	16.34 %	3.73 %	0.73 %	3.17 %	0.60 %	69.83 %	16.16 %
Jaccard	63.52 %	16.16 %	1.27 %	0.16 %	1.27 %	0.16 %	68.74 %	16.16 %
Jaro-Winkler	66.21 %	22.42 %	6.92 %	1.50 %	5.15 %	1.03 %	70.83 %	22.05 %
	k-NN		AdaBoost DT		OvO		OvA	
Dist. Metric	Type	Impl.	Type	Impl.	Type	Impl.	Type	Impl.
Weighted-Levenshtein	89.06 %	55.62 %	64.26 %	29.41 %	90.43 %	58.92 %	90.32 %	53.24 %
Normal-Levenshtein	84.63 %	46.56 %	63.98 %	25.43 %	86.20 %	51.74 %	86.16 %	43.64 %
Damerau-Levenshtein	83.03 %	46.67 %	64.77 %	26.22 %	86.20 %	51.59 %	86.16 %	44.22 %
Sorensen-Dice	26.15 %	13.11 %	63.52 %	16.16 %	63.52 %	16.16 %	63.52 %	20.06 %
Jaccard	26.15 %	13.11 %	63.52 %	16.16 %	63.52 %	16.16 %	63.52 %	20.06 %
Jaro-Winkler	90.68 %	60.68 %	73.62 %	27.39 %	67.52 %	27.05 %	67.91 %	20.74 %

Table 4: Ranking achieved by the behaviour optimiser.

Rank	Metric	Behaviour Implementation
1. St. Chess	419	FruitReloaded
2. St. Chess	396	Stockfish
3. St. Chess	356	Laser
4. St. Chess	-4	Pulse
1. Ch. Chess	463	Mars
2. Ch. Chess	250	Eleeye
1. Checkers	454	Ponder

Suryadi et al. describe in (Suryadi and Gmytrasiewicz, 1999) a system of agents where each agent predicts the behaviour of all other agents by utilising a trained behaviour model. Thus, they know all possible actions. However, the internal state is unknown. To generate a behaviour model, the agents observe each other and their environment. The resulting model is represented by an influence diagram which is used to optimise the own behaviour. In contrast, we consider an agent behaviour as a black box. The generated behaviour models are used to rate the own behaviour and find the most suitable behaviour implementation.

However, the approaches considered in multi-agent systems typically do not classify or replace their behaviour to optimise their performance. Related work in the area of service-oriented architectures typically focuses on these two aspects. Service change management provides automated tools, including service replacement during the application life-cycle, which is essential for the robustness and dependability of a dynamically changing system. In this area, many papers deal with this topic in the areas of interface specification, service compatibility, service discovery, service matching, and service replacement.

Several tools, frameworks, and strategies were proposed to detect different kinds of service changes using syntactical (Fokaefs and Stroulia, 2013) and semantic information (Stavropoulos et al., 2019). In addition, ontologies are used to represent semantic service information (Groh et al., 2019).

Because of the large number of works and limited space, we discuss related work that focuses on service analysis utilising data mining and machine learning techniques to categorise services and find possible replacements in the following paragraphs.

In (Shen and Liu, 2019), web service discovery is divided into two parts. First, the web service clustering represents the service descriptions as vectors and maps them to the semantic information contained

in the descriptions. The authors provide four different unsupervised sentence representations. Second, a Latent Dirichlet Allocation method detects semantic topic information of web services after a service request and stores it into a specific cluster according to its web service text-description vector.

The authors in (Yang et al., 2019) present a deep neural network to abstract service descriptions to high-level features. The additional service classification process utilises 50 service categories.

In (Li et al., 2018), the authors propose an automatic approach to tag web services by extracting WSDL (Web Services Description Language) information and provide tag recommendations for service discovery using the weighted textual matrix factorisation. In contrast to our solution, these works focus on analysing, extending and categorising interface descriptions and do not consider the behaviours of services.

In (Yahyaoui et al., 2015) the authors propose an approach for modelling and classifying service behaviours by capturing the service performance through predefined behavioural patterns. Each pattern is a typical sequence of observations. An observation denotes the quality of a service for one interaction. They also consider services as black boxes but attempt to match their performance on predefined patterns.

However, none of the works uses the black box behaviour observation of services exclusively for classification in combination with support for additional data such as service descriptions and semantic information for classification improvement.

6 CONCLUSIONS

The main contributions of this paper are self-optimising tasks that learn representations of Behaviour Implementations. Thus, the tasks are empowered to evaluate their given Behaviour Implementations and, if necessary, exchange them for better evaluated implementations of the same behaviour. The implementations are considered as black boxes, and thus only the input and output data is considered. Furthermore, a distributed behaviour repository organises the learned Behaviour Models and supports storing, searching, and sharing of the corresponding Behaviour Implementations. Our evaluation shows the feasibility of our approach by comparing a set of suitable machine learning algorithms. In general, they achieve reliable results during the classification of different behaviours.

The machine-learned behaviour model represents the transformation of an input stream to its output stream. These models can be compared and, based on the results, equivalent Behaviour Implementations can be determined. We apply hyperplane classifiers to learn the Behaviour Model. Due to their specific characteristics, especially the simple model transferability and comparability, Support Vector Machines are well suited. In (Jahl et al., 2018), it is proven that unsupervised Support Vector Machines are appropriate for the application in this approach if restricted to one-dimensional inputs and outputs. This work overcomes the restriction and enables the analysis of complex data structures.

Further research and experiments are necessary to achieve detailed results about the accuracy of the utilised classifiers in additional application domains. In our future work, we want to improve our prototype by selecting machine learning techniques tailored for data streams for Ensemble Learning. This leads to a distributed behaviour repository where individual Skill Managers are specialised on a specific type of Behaviour Implementations and thus provides a better classification for the corresponding behaviour type. Additionally, higher-level management will provide a tree-like structure to improve the organisation and selection of Behaviour Implementation replacements.

REFERENCES

Amer, M., Goldstein, M., and Abdennadher, S. (2013). Enhancing one-class support vector machines for unsupervised anomaly detection. In *Proceedings of the ACM SIGKDD workshop on outlier detection and description*, pages 8–15.

Bishop, C. M. et al. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Dia, H. (2002). An agent-based approach to modelling driver route choice behaviour under the influence of real-time information. *Transportation Research Part C: Emerging Technologies*, 10(5):331–349.

Fokaefs, M. and Stroulia, E. (2013). WSDarwin: Studying the Evolution of Web Service Systems. In *Advanced Web Services*, pages 199–223. Springer New York.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012). Answer set solving in practice. *Synthesis lectures on artificial intelligence and machine learning*, 6(3):1–238.

Groh, O., Baraki, H., Jahl, A., and Geihs, K. (2019). COOP - automatiC validatiOn of evOlving microservice comPositions. In *Seminar Series on Advanced Techniques & Tools for Software Evolution*. SATToSE2019, CEUR-WS.

Jahl, A., Jakob, S., Baraki, H., Alhamwy, Y., and Geihs, K. (2021). Blockchain-based Task-centric Team Build-

ing. In *Proceedings of the ICAART 2021*, volume 1, pages 250–257. SCITEPRESS.

Jahl, A., Tran, H. T., Baraki, H., and Geihs, K. (2018). WiP: Behavior-Based Service Change Detection. In *International Conference on Smart Computing*. IEEE.

Jakob, S., Baraki, H., Jahl, A., Nyakam Chiadjeu, E. D., Alhamwy, Y., and Geihs, K. (2021). Adaptive Semantic Routing in Dynamic Environments. In *Proceedings of the ICAART 2021*, volume 2, pages 997–1004. ICAART2021, SCITEPRESS.

Li, G., Cui, Y., Wang, H., Hu, S., and Liu, L. (2018). Web Services Tagging Method Based on Weighted Textual Matrix Factorization. In *International Conference on Computer Engineering and Networks*, pages 385–390. Springer.

Meyer-Kahlen, S. and Huber, R. (2015). UCI Schach Engine Protocol. *Shredder Chess*.

Picard, G., Hübner, J. F., Boissier, O., and Gleizes, M.-P. (2009). Reorganisation and Self-Organisation in Multi-Agent Systems. In *1st International Workshop on Organizational Modeling, ORGMOD*, pages 66–80.

Shen, Y. and Liu, F. (2019). An Approach for Semantic Web Discovery Using Unsupervised Learning Algorithms. In *Cyberspace Data and Intelligence, and Cyber-Living, Syndrome, and Health*, pages 56–72. Springer.

Skubch, H. (2012). *Modelling and Controlling of Behaviour for Autonomous Mobile Robots*. Springer Science & Business Media.

Stavropoulos, T. G., Andreadis, S., Kontopoulos, E., and Kompatsiaris, I. (2019). SemaDrift: A Hybrid Method and Visual Tools to Measure Semantic Drift in Ontologies. *Journal of Web Semantics*, 54:87–106.

Suryadi, D. and Gmytrasiewicz, P. J. (1999). Learning models of other agents using influence diagrams. In *UM99 User Modeling*, pages 223–232. Springer.

Tarassenko, L., Hayton, P., Cerneaz, N., and Brady, M. (1995). Novelty Detection for the Identification of Masses in Mammograms. In *Fourth International Conference on Artificial Neural Networks*, pages 442–447.

Xing, H.-J. and Liu, W.-T. (2020). Robust AdaBoost Based Ensemble of One-class Support Vector Machines. *Information Fusion*, 55:45–58.

Yahyaoui, H., Own, H. S., and Malik, Z. (2015). Modeling and Classification of Service Behaviors. *Expert Systems with Applications*, 42(21):7610–7619.

Yang, Y., Ke, W., Wang, W., and Zhao, Y. (2019). Deep Learning for Web Services Classification. In *International Conference on Web Services (ICWS)*, pages 440–442. IEEE.