KMARF: A Framework for Knowledge Management and Automated Reasoning*

Aneta Vulgarakis Feljan, Athanasios Karapantelakis, Leonid Mokrushin, Rafia Inam, Elena Fersman, Carlos R. B. Azevedo, Klaus Raizer and Ricardo S. Souza
Ericsson Research
Email:fistname.lastname@ericsson.com

ABSTRACT

In this paper, we present a generic framework for knowledge management and automated reasoning (KMARF) as an enabler for intelligent adaptive systems. KMARF targets multiple reasoning problem classes (such as planning, verification and optimization) that can share the same underlying system state representation. The idea behind KMARF is to automatically select an appropriate problem solver based on a formalized reasoning expertise in the knowledge base, and convert a problem definition to a problem solver-readable format. Automation of the reasoning process reduces operational costs and enables the system to operate in dynamic environment conditions. We demonstrate our approach using a transportation planning use case.

KEYWORDS

Knowledge model, Adaptive systems, Reasoning, Model transformation

1 INTRODUCTION

Internet of Things (IoT)-based systems in general use a vast amount of heterogeneous data streams and information (e.g., behavioral models) that need to be analyzed, combined and actioned upon. This creates a complexity that is unsustainable by only human management and therefore, automation is a requirement. One of the first steps towards automation is formalization of knowledge extracted from the different sources such as sensor networks, documents, tools and from system experts. Since all the states of the system and the environment cannot be predicted at design time, there is a need for developing intelligent , self-adaptive systems which learn to adjust their behavior in response to the environment. To do this, one needs to leverage the learnings from artificial intelligence and cognitive technologies research that aim at building intelligent systems that know about their world and are able to automatically draw conclusions and act upon them, as humans do. A fundamental assumption in this research is that knowledge is represented in a tangible form (usually via ontologies), suitable for processing by dedicated reasoning engines [3]. Even though there are a number of frameworks for general intelligence attempting to solve several classes of reasoning problems, such as planning, verification and optimization [10, 15, 16, 19], it is still not clear to the intelligent software community how to effectively cope with the integration of both declarative and procedural knowledge and some authors advocate for keeping the behavior descriptions separated from the semantic, static domain knowledge [9].

In this paper, we present a Knowledge Management and Automated Reasoning Framework (KMARF), which targets multiple reasoning problems. The purpose of KMARF is to (a) reduce system development and deployment time, by reusing as much knowledge as possible, such as domain models, behaviors and reasoning mechanisms, and (b) reduce operational costs by enabling systems during run-time to automatically decide how to adapt to changes in their contexts and environments, with minimal or no human interaction.

The strong point of KMARF is the particular way in which it relies on a knowledge model that combines both declarative and procedural knowledge. This combination allows the system to automatically do extensive analysis and provide answers to different reasoning questions, such as "what is the (optimal) strategy for reaching a given state?", "can the system end-up in an unsafe state?", "how much profit can the system generate after a given amount of time?". On top of this KMARF supports system self-adaption by selecting appropriate problem solvers and their parameters upon encountering any performance degradation. For example, the system may use KMARF in order to switch from preplanned to reactive operations due to insufficient time to replan and changed environment conditions. In the first case the system will use an offline planner as a problem solver, and in the latter case it might use a local decision maker based on the current environment inputs. In order to use problem solvers specialized in solving specific classes of reasoning problems, KMARF can be extended with model transformation rules that translate from our knowledge model to the targeted problem solver model.

KMARF is suitable for large scale IoT based systems, and so far we have applied it on the Intelligent Transport Systems (ITS) domain. As an illustration example, we consider a transportation planning problem i.e., how to transport passengers or goods with a minimal cost. A cost can be e.g., the traveled distance, the time needed for transportation of each of the passengers or goods, or the number of buses or trucks required for transportation. The answer to the task may be a plan i.e., a sequence of steps for the system to perform in order to reach the goal state. In case the task cannot be performed the answer from KMARF could be a reason why the task cannot be performed, as well as a possible solution, which could be to increase the number of vehicles.

In the literature, there exist several reference architectures for development of self-adaptive systems, such as MAPE-K [7], PELEA [11] and SOA-PE [18]. Common to these architectures is their identification of four processes for adaptivity: monitoring, analyzing, planning, and executing. Additionally, KMARF extends these architectures with a meta-reasoner that by using meta-reasoning

expertise draws a conclusion about an appropriate problem solver or a method, and a relevant prior knowledge needed for solving a given problem. Once the meta-reasoner decides on an appropriate problem solver to be used KMARF provides model transformation rules to translate the problem definition from KMARF's knowledge model into a format understandable by the selected problem solver, such as Prolog, Timed Automata [1] or PDDL [17].

In brief, our contribution is threefold:

- An architecture of a generic framework for knowledge management and automated reasoning (Section 2.1)
- A knowledge model for representing both declarative and procedural system knowledge in a machine-readable form (Section 2.2).
- A prototype implementation of the KMARF architecture (Section 3).

2 KMARF – FRAMEWORK FOR KNOWLEDGE MANAGEMENT AND AUTOMATED REASONING

In this section, we introduce our Knowledge Management and Automated Reasoning Framework by describing its architecture and the knowledge model that it relies on. KMARF is targeting multiple reasoning problem classes (such as planning, verification and optimization) that can share the same underlying state representation. This enables reusability of knowledge and reasoning methods across different domains, thus KMARF reduces operational costs and supports systems operation in changing environment conditions.

2.1 Architecture

A high-level conceptual view of the architecture of the framework is depicted in Figure 1. The main components of KMARF are the *Perception Engine*, the *Knowledge Base*, the *Reasoner*, the *Interpreter* and the *Actuation Engine*.

The *Knowledge Base* is responsible for representing aspects of the domain under consideration (such as objects or concepts, instances and states) and their relations, in well defined, machine processable syntax and unambiguous semantics. The format of the knowledge stored in the knowledge base complies with the knowledge model introduced in Section 2.2, and one of the possible formats is RDF/OWL [12]. In addition, the knowledge base contains *meta-reasoning expertise*, as well as *model transformation rules* that are described below.

The *Reasoner* is used for solving reasoning problems. Reasoning in context of this paper is defined as the process of solving problems related to planning, verification, optimization, etc. By relating a *goal query* to a *meta-reasoning expertise*[1] stored in the knowledge base the *Inference Engine* draws a conclusion about an appropriate method or a problem solver, and relevant prior knowledge for solving a given problem. The goal query can be initiated by the user or the *Inference Engine* itself based on comparison between expected and actual system states. For example, if the goal query is to reach a certain goal, the inference engine will look up the knowledge base and deduce that a *Planner* should be used to generate a strategy to reach that goal. Additionally, given that most of the planners accept planning problems in Planning Domain Definition Language (PDDL) [17] as their input, the inference engine looks up corresponding *model transformation rules* that should be applied to formulate the problem in a format understandable by the selected planner. The *Interpreter* takes the generated strategy and maps it to state changes that it gives to the *Actuation Engine*, so that it can perform actuation in the real world.

Since the physical world is not entirely predictable KMARF needs to take into consideration that there might be changes in the information stored in the knowledge base. The *Perception Engine* is responsible, when needed, to push new knowledge from the environment (i.e., predicates) in the knowledge base. Additionally, when executing the strategy the *Interpreter* works tightly with the *Reasoner*. In case there are any changes in the expected state of the system the *Reasoner* sends a replanning request to the *Planner*.

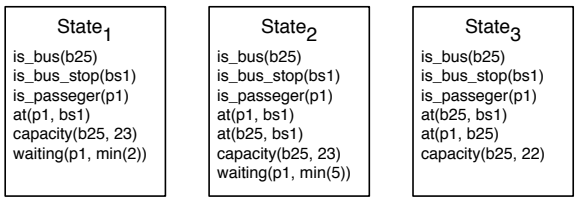

Figure 2: Specification of three example states.

2.2 The Knowledge Model

After introducing the architecture of KMARF, we can move on presenting its knowledge model that combines both declarative and procedural knowledge.

Syntax. We model declarative knowledge by describing discrete states of a system. One such state may represent the current state, and the others may describe either previous system states or hypothetical states that the system may end up in the future. In this context we do not strictly apply the notion of time, i.e., the system may change its state instantly. However, the order of states is important as it describes how the system evolves and may explain the reasons behind its progress.

A *state* is represented by an (implicitly conjunctive) set of predicates $\{P_1, P_2, ...\}$ expressing the facts known about the state. Each predicate is a compound term that has a form of $P_i(a_1, a_2, ..., a_n)$, where P_i is a predicate's functor specified as a literal, i.e., a sequence of characters, and $a_j|j \in [1..n]$ are the arguments. The number of arguments n is called arity of the predicate. If $n = 0$, then P_i denotes a simple atomic fact. If $n > 0$, then P_i denotes a factual relation between its n arguments. The arguments of predicates may include:

- numbers denoting literal quantity;
- literals denoted by sequence of alphanumeric characters that start from a lower case character $\{a, b, c, ...\}$ that represent objects or concepts in the domain (e.g., *car* may represent "a car");

[1] Meta-reasoning is reasoning about reasoning, i.e., it is comprised of computational processes concerned with the operation and regulation of other computational processes within the same entity [20].

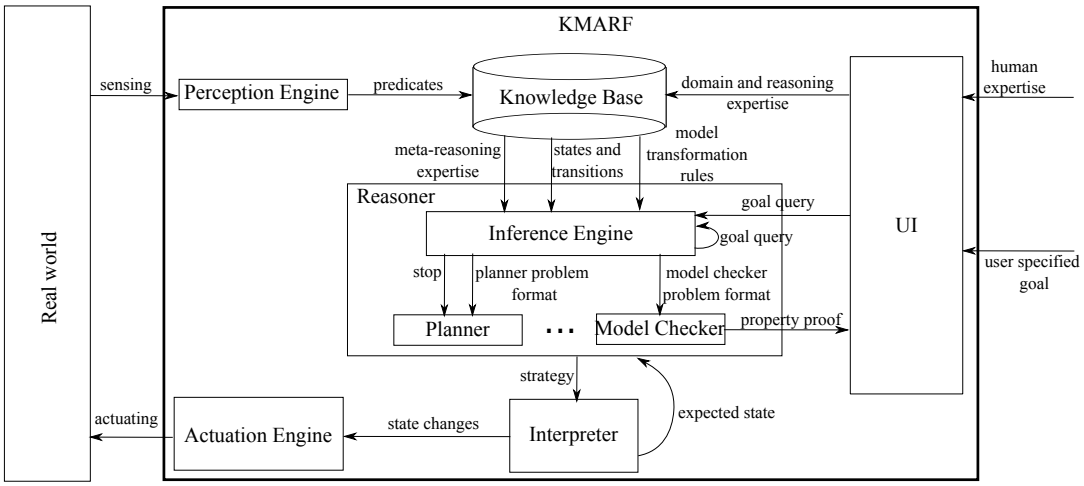

Figure 1: A high level, conceptual view of KMARF's architecture.

- compound terms of the form $T_i(a_1, a_2, ..., a_n)$ that may have one or many arguments, which can be numbers, literals or compound terms (e.g., $velocity(car, kmh(50))$ may stand for "the velocity of the car car is 50 km/h").

An example shown in Figure 2 contains a state $State_1$ defined using a set of six predicates $\{is\_bus, is\_bus\_stop, is\_passeger, at, capacity, waiting\}$. The arguments of the first three predicates declare existence of bus $b25$, bus stop $bs1$, and passenger $p1$ accordingly. The fourth predicate states that passenger $p1$ is at bus stop $bs1$. The fifth predicate indicates that bus $b25$ has 23 available places. The last predicate declares a fact that passenger $p1$ has been waiting at the bus stop from last two minutes.

The procedural knowledge is modeled as a collection of specifications of potential transitions between states. A *transition* specification consists of a *precondition*, a *computation*, and an *action*. Precondition and action both have the same syntax as the state, i.e., they are represented as a set of predicates, except the following two differences. Variables denoted by sequences of alphanumeric characters starting from a capital letter $\{X, Y, Z, ...\}$ are allowed in arguments of predicates and compound terms in both precondition and action. The action predicates are restricted to the set $\{add, delete\}$. Intuitively, action predicates denote the procedures performed with a state when a transition is performed. Computation is an ordered list of effect free function calls, i.e. they do not modify the state and are only used during the processing of the transition. The arguments of a function call may be numbers, literals, variables, and functions. If a function returns a value, the last argument of a function call is a variable that holds it. The result of a function call may be used as an argument in subsequent function calls of the computation or in the action.

Example in Figure 3 demonstrates specification of a transition $Transition_1$ that allows a system to evolve from state $State_2$ to state $State_3$ defined in Figure 2. After matching the precondition, the computation checks if a passenger has been waiting for less than 20 minutes and if there is enough capacity to onboard a passenger, it decreases the bus capacity by 1. The action removes the passenger

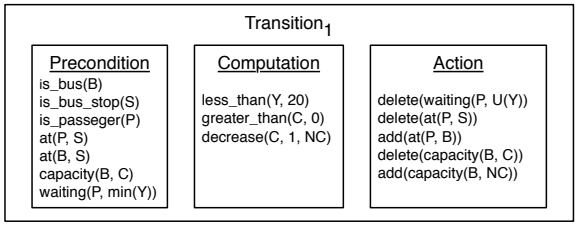

Figure 3: An example of a transition specification.

waiting predicate, updates passenger location and available bus capacity value.

By defining the rule consequents as add and delete operations on the knowledge base itself, we allow for a self-referential rule-based system [6] that can update the knowledge base using meta-rules triggered by events detected in the data streams coming from real-time system execution. This feature also enables self-adaptation capabilities based on real-time changes in the knowledge base.

Semantics. We formalize semantics of literals, compound terms and predicates by associating meanings to their symbols. For example, we use the predicate $at(p1, bs1)$ to model the fact that a passenger $p1$ is at the bus stop $bs1$. We say that the meaning of at is to represent a close spacial relationship between its two arguments.

We define semantics of our knowledge model in terms of a transition system. Formally, a transition system is a tuple $(S, T, \rightarrow)$, where S is a set of states, T is a set of transition specification names, and $\rightarrow$ is set of state transitions (i.e., a subset of $S \times T \times S$). The fact that $(State_1, Transition_1, State_2) \in \rightarrow$ is written as $State_1 \xrightarrow{Transition_1} State_2$, and represents a transition between a source state $State_1$ and a destination state $State_2$ by applying transition specification $Transition_1$.

In order for a transition specification to be applied its precondition must match the source state and its computation must succeed. The semantics of matching the precondition with a state are formalized by defining a logical unification between predicates of the

precondition and the predicates of the state, as follows. We unify every predicate in the precondition with all the predicates from the state, and use variables substitutions in subsequent unification of the remaining precondition predicates. If the unification of all precondition predicates with the state predicates succeeds, the computed variable substitution is used in the computation and the action, as explained below. Obviously, there can be multiple matches of a precondition with a source state. In this case, every match will produce a transition in $\rightarrow$ given that the corresponding computation succeeds.

The meaning of the computation is evaluation of its functions in the order of specification. We use denotational semantics to define the meaning of a function F as a set of ordered tuples

$$\{< a_1^1, ..., a_{n-1}^1, a_n^1 >, ..., < a_1^m, ..., a_{n-1}^m, a_n^m >\},$$

where $a_1^i, ..., a_{n-1}^i$ are function arguments, and a_n^i is the value returned by F given those arguments. A function call $F(a_1, ..., a_{n-1}, a_n)$ is the process of finding such a_n for given $a_1, ..., a_{n-1}$ that there is a tuple $< a_1^j, ..., a_{n-1}^j, a_n^j >$ in the definition of F for some j. If there is no such tuple found, the function call fails. Otherwise, the function call succeeds and the value of a_n^j is assigned to a corresponding variable. If the returned value a_n is of boolean type, i.e. it belongs to the set $\{true, false\}$, then the function succeeds if $a_n = true$ and fails if $a_n = false$. A computation succeeds if all the function calls in it succeed. Otherwise, the computation fails.

The semantics of the transition action execution are defined by two operations. The first operation instantiates predicates in the action by applying computed variable substitution to them. This means that all the variables in the action predicates are replaced with the corresponding values from the variable substitution. The second operation copies all the predicates from the source state to the destination state, and for every predicate in the action we perform the following procedures on the destination state:

- if the instantiated predicate symbol is *add*, then its argument is treated as a predicate, and it is added to the destination state;
- if the instantiated predicate symbol is *delete*, then its argument is treated as a predicate, and it is removed from the destination state.

3 IMPLEMENTATION

This section describes current progress towards a prototype implementation of the KMARF architecture illustrated in Figure 1. The implementation targets a large problem area in ITS known as "transportation planning", which we define as *the schedules generated for a set of vehicles to pickup and alight people or cargo along one or more routes, within a given amount of time* (see also Section 1). The transport planning problem includes a set of connected vehicles, for example buses or trucks, and a central coordinating function that computes the schedule and transmits it to these vehicles[2]. In this implementation we assume that the *Inference Engine* component has already deduced that a *Planner* should be used to solve the transportation planning problem, using *meta-reasoning expertise* and *user query* data supplied from the *Knowledge Base* and the user interface components respectively.

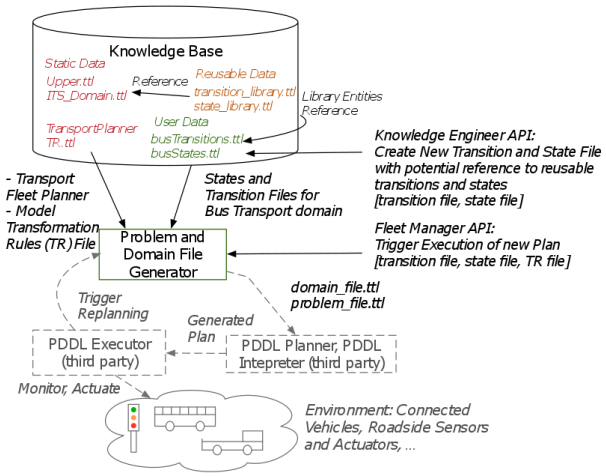

Figure 4: Overview of the implemented system. The ".ttl" extension denotes model files in Turtle format. Components in dotted rectangles are third-party.

Figure 4 shows the components of the implemented system. One of the components implemented is the *Knowledge Base*, which contains *model transformation rules* for PDDL language as well as *states and transition* models that are based on a set of ontologies that the authors have defined in [14] and contain information for the particular transport planning problem. The ontologies are organized in multiple layers of abstraction, a common one, called "Upper", and an "ITS" specific one used for reasoning about ITS related problems. Since the ontologies are out of the scope of this paper, for more details we refer the reader to [14]. The above models are described using semantic web technologies and are based on the W3C Web Ontology Language (OWL) [12] and stored in Turtle format [4]. The other component is a PDDL Generator, which, given the transformation rules, states and transition files as input, generates problem and domain files in PDDL language. PDDL Generator is implemented in Java [13] and uses Apache Jena [2] for parsing the Turtle-formatted input from the knowledge base. Additionally, Eclipse Jetty [8] provides a Representational State Transfer (REST) API for triggering PDDL file generation, and defining custom states and transition models. More specifically:

- The API allows human experts (for example knowledge engineers) to specify a transport logistics problem, by adding a new initial and goal state in the knowledge base, in the form of a *state file*, and a set of transitions with precondition, computation and action parts in the form of a *transition file*. These two files are jointly used by the PDDL Generator software component in order to generate a new schedule. The state file defines the agents, vehicles and routes, contains information about the initial state of the system (e.g., the location of agents in the route, the route and its waypoints, the time required for vehicles to travel a route, etc.) and defines goal conditions (e.g., all agents

[2]Correct interpretation of the schedule rests on the vehicles, which can be partially or completely autonomous or they may also have human drivers.

are serviced). The transitions file describes intermediate transitions that are used by the planner to reach the specified goal state from the initial state. An example of such plan can be found at [14].

- The API also provides means for triggering generation of a new PDDL problem and domain file given the above input on request of a human or other system. Typically this request is created from a customer (e.g., human operator, or an automated fleet management system). Once generated, the files are assigned Universal Resource Identifiers (URIs). An external system can subsequently perform Hypertext Transfer Protocol (HTTP) GET requests using the URI references to retrieve the files. An example of such a system can be a PDDL solver[3] For this implementation, we use a third-party solver named "OPTIC", originally developed by Benton et al [5].

The authors have released the current implementation of the Knowledge Base and PDDL Generator as open-source, available for the community to use [14]. One exemplary use of our implementation can be to initiate replanning activity based on detection of a blocked route (e.g., due to road accident or roadwork) or runtime input on larger passenger demand. Currently, our implementation does not cover such use cases since there is no component to support interaction with the real world, both for triggering the planning process, but also for actuating real-world connected devices (e.g. buses or sensors) upon execution of the generated plan, however this is planned work. In its current state the implementation can be used for rapid prototyping of transportation planning functions. In addition to the software itself, the "Upper", "ITS", "PDDL model transformation rules" and a set of common reusable transitions and state ontologies are provided in Turtle format.

4 CONCLUSION AND FUTURE WORK

In this paper, we introduce a generic framework for knowledge management and automated reasoning (KMARF) as an enabler for intelligent adaptive systems. Given a stimulus as a problem statement, KMARF automatically performs reasoning, selects appropriate problem solvers, and allows the system to adapt to changes in its context and environment. To do this, KMARF relies on a meta-reasoner that by using meta-reasoning expertise draws a conclusion about an appropriate problem solver or a method to perform reasoning. Once a particular problem solver has been selected, KMARF uses a combination of a declarative and procedural knowledge model and a transformation rules model to generate an input for the problem solver. Subsequently, the result of problem solving is used to progress towards completing the original problem.

In the implementation of KMARF we have so far progressed into the development of an ontology for the knowledge base using OWL Web Ontology Language, and we have used the framework to automatically generate PDDL files, feed them into a planner, and create plans. Since, KMARF has a much bigger vision than solving planning problems, in the future we plan on studying how our knowledge model can be correlated to other formalisms e.g., Timed Automata.

As future work, we also plan to study how meta-reasoning can help KMARF to determine which prior knowledge and algorithms are relevant when a new, problem or unforeseen instance arrives. Such instance corresponds to the current state of the world, along with all current available sensory information. Initially, the system only knows how to solve problems it has seen before and had previously found reductions that could be solved separately using known procedures. If it can not find such a reduction for a new instance, it must recur to space state exploration for generating a sequence of state transitions that either lead to the specified goal state or to a better state in which either the system knows how to proceed with further reductions or declares the problem as intractable under its current knowledge.

[3] In its current form, the API does not support adding of new model transformation rules, which means that only PDDL language is supported. In the future however, we plan to expand the functionality by adding support for "pluggable" problem solving expertise files.

REFERENCES

[1] Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183 – 235.

[2] Apache Foundation. 2016. Apache Jena: A free and open source Java framework for building Semantic Web and Linked Data applications. (2016). https://jena.apache.org

[3] Chitta Baral. 2003. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA.

[4] David Beckett, Berners-Lee Tim, Prud'hommeaux Eric, and Carothers Gavin. 2014. RDF 1.1 Turtle: Terse RDF Triple Language. *W3C Recommendation* (February 2014). http://www.w3.org/TR/turtle/

[5] J. Benton, Amanda Coles, and Andrew Coles. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *International Conference on Automated Planning and Scheduling*. https://www.aaai.org/ocs/index.php/ICAPS/ICAPS12/paper/view/4699

[6] William J. Clancey. 1983. The epistemology of a rule-based expert system – a framework for explanation. *Artificial Intelligence* 20, 3 (1983), 215 – 251.

[7] Autonomic Computing. 2003. An architectural blueprint for autonomic computing. *IBM Publication* (2003).

[8] Eclipse Foundation. 2016. Jetty: Open-Source Servlet Engine and HTTP Server. (2016). http://www.eclipse.org/jetty/

[9] Malik Ghallab, Dana Nau, and Paolo Traverso. 2014. The actor's view of automated planning and acting: A position paper. *Artificial Intelligence* 208 (2014), 1 – 17. DOI: http://dx.doi.org/10.1016/j.artint.2013.11.002

[10] Ben Goertzel, Cassio Pennachin, and Nil Geisweiller. *Engineering General Intelligence, Part 1*. Atlantis Thinking Machines, Vol. 5. Atlantis Press. http://link.springer.com/10.2991/978-94-6239-027-0

[11] César Guzmán, Vidal Alcázar, David Prior, Eva Onaindia, Daniel Borrajo, Juan Fdez-Olivares, and Ezequiel Quintero. 2012. PELEA: a domain-independent architecture for planning, execution and learning. In *Proceedings of the Scheduling and Planning Applications Workshop*, Vol. 12. 38–45.

[12] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F Patel-Schneider, and Sebastian Rudolph. 2012. OWL 2 Web Ontology Language Primer (Second Edition). *W3C Recommendation* (December 2012). https://www.w3.org/TR/owl2-primer/

[13] Gosling James, Joy Bill, Steele Guy, Bracha Gilad, and Buckley Alex. 2015. The Java Language Specification: Java SE 8 Edition. *Oracle* (February 2015). https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[14] KMARF authors 2016. Prototype PDDL Generator Public Repository. (2016). https://github.com/SSCIPaperSubmitter/ssciPDDLPlanner

[15] John Laird. 2012. *The SOAR cognitive architecture*. MIT Press, Cambridge, Mass. ; London, England.

[16] J. Licato, R. Sun, and S. Bringsjord. 2014. Structural representation and reasoning in a hybrid cognitive architecture. In *2014 International Joint Conference on Neural Networks (IJCNN)*. 891–898. DOI: http://dx.doi.org/10.1109/IJCNN.2014.6889895

[17] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. 1998. *PDDL - The Planning Domain Definition Language*. Technical Report. CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212

[18] Swarup Kumar Mohalik, Mahesh Babu Jayaraman, Badrinath Ramamurthy, and Aneta Vulgarakis Feljan. 2015. SOA-PE: A service-oriented architecture for planning and execution in cyber-physical systems. In *Proceedings of IEEE IC-SSS*.

[19] Alexei V. Samsonovich. 2010. Toward a Unified Catalog of Implemented Cognitive Architectures. In *Proceedings of the 2010 Conference on Biologically Inspired Cognitive Architectures 2010: Proceedings of the First Annual Meeting of the BICA*

Society. IOS Press, Amsterdam, The Netherlands, The Netherlands, 195–244.

[20] Robert Andrew Wilson and Frank C Keil. 2001. *The MIT encyclopedia of the cognitive sciences*. MIT press.