Proceedings of the 2016 IEEE International Conference on
Simulation, Modeling, and Programming for Autonomous Robots
San Francisco, USA, Dec 13-16, 2016

Graph-based Software Knowledge: Storage and Semantic Querying of Domain Models for Run-Time Adaptation

Nico Hochgeschwender, Sven Schneider, Holger Voos, Herman Bruyninckx and Gerhard K. Kraetzschmar

***Abstract*— Software development for robots is a knowledge-intensive exercise. To capture this knowledge explicitly and formally in the form of various domain models, roboticists have recently employed model-driven engineering (MDE) approaches. However, these models are merely seen as a way to support humans during the robot's software design process. We argue that the robots themselves should be first-class consumers of this knowledge to autonomously adapt their software to the various and changing run-time requirements induced, for instance, by the robot's tasks or environment.**

Motivated by knowledge-enabled approaches, we address this problem by employing a graph-based knowledge representation that allows us not only to persistently store domain models, but also to formulate powerful queries for the sake of run time adaptation. We have evaluated our approach in an integrated, real-world system using the `neo4j` graph database and we report some lessons learned. Further, we show that the graph database imposes only little overhead on the system's overall performance.

I. Introduction

The realization of modern robotic systems is a knowledge-intensive process that reflects, involves and builds upon decisions from complex, heterogeneous fields of research and engineering – reaching from hardware design over domains such as control, perception or planning to software engineering. Especially in robotic's software engineering, the challenging and interdisciplinary integration of these fields is all too often solved in an ad-hoc manner for very specific problems. Consequently, knowledge and assumptions about the robot's software frequently remain *implicit*, as discussed in [1]. Formal methods and explicit representations of software knowledge are promising approaches to cope with implicit knowledge representations. However, these approaches are usually regarded as a tool for human robot designers – the robots themselves are denied access to this knowledge.

Recently, the robotics community has developed a growing interest in MDE [2] as a means to capture domain knowledge of the various robotics fields *explicitly* in the form of *domain models* [3]. Such domain models are described by *domain-specific languages* (DSL). In contrast to general purpose programming language, DSLs allow domain experts to specify their knowledge in familiar terminology and representations such as textual, graphical or mathematical notations. DSLs and their associated domain models fulfill varying requirements such as documentation of taken approaches, visualization of concepts or the generation of code. The latter point, i.e. code generation, is most frequently associated with MDE and a major focus in nowadays robotics software engineering [3]. In such an approach a part of or a complete robot system is modeled at *design time*, then the software is generated and often the modeling effort is forgotten.

We argue that the consequent next step is to provide robots with these explicit knowledge representations and let them reason about their software at *run time*. This is one of many key ingredients for autonomous and intelligent robots that are able to adapt to their tasks and dynamic environments [4]. Thus, the core problem investigated in this paper is: *How to grant robots access to the software-related models at run time?* This involves *a*) persistently storing the different notations and formats of DSLs; *b*) composing the various domain models; and *c*) querying over multiple domains at run time.

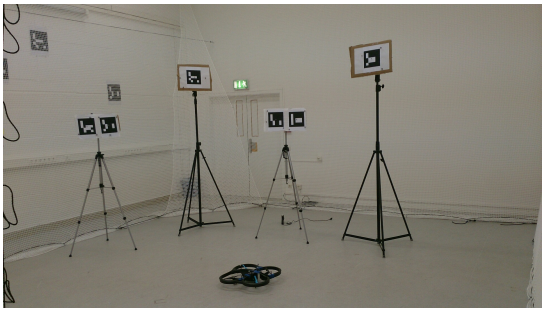

Fig. 1. The quadrocopter in an indoor environment with time-varying illumination conditions.

The AI community has already realized this requirement, as evidenced by knowledge-enabled approaches like KnowRob [5], RoboBrain [6] or the OpenRobot Ontology (ORO) [7]. At the core of these approaches, graph-based knowledge representations such as ontologies provide common representations and query interfaces to the robot's run-time environment. In this connection, we make the following contributions for software-related knowledge:

- We employ *labeled property graphs* as simple, yet powerful means to persistently store and compose domain models originating from different functional domains and software development phases.

Nico Hochgeschwender, Sven Schneider and Gerhard Kraetzschmar are with the Department of Computer Science, Bonn-Rhein-Sieg University of Applied Sciences, Germany. Email: `forename.surname@h-brs.de` Nico Hochgeschwender and Holger Voos are with the Research Unit in Engineering Sciences, University of Luxembourg, Luxembourg. Email: `holger.voos@uni.lu` Sven Schneider and Herman Bruyninckx are with the Department of Mechanical Engineering, KU Leuven, Belgium. Email: `Herman.Bruyninckx@mech.kuleuven.be`

978-1-5090-4616-4/16/$31.00 ©2016 IEEE

- We demonstrate how *semantic querying* can be applied at run time to derive implicitly defined information based on composed domain models.
- We evaluate our approach in an integrated, real-world system based on the `neo4j` graph database by creating, updating and querying domain models for the sake of run-time adaptation.

II. Motivation: Model-based Development

We exemplify the structured, model-based development of a real-world robotic application based on the BRICS Robot Application Development Process (BRICS RAP or short RAP) [8]. The RAP is a holistic, tailorable, iterative process model for developing robotic applications both in academic and industrial settings. The process model forsees eight different phases, therefrom four will be employed in the following as those are sufficient to motivate the scenario.

Application Scenario

The application (system) under study constitutes a quadrocopter instructed to fly in a GPS-denied indoor environment with time-varying illumination conditions (see Fig. 1). The environment is equipped with fiducial markers [9] used by the quadrocopter to localize itself. As we experimentally analyzed in [10], the recognition performance in the presence of time-varying illumination conditions significantly depends on adapting the modifiable parameters of the marker recognition algorithm at run time. Therefore, the quadrocopter needs to continously monitor the illumination condition and eventually adapt its software architecture to continue to properly estimate its own pose. However, as the computational hardware of the quadrocopter is limited it is not possible to execute all functionalities (e.g. marker detection, flight control etc.) on the same platform. In that case a remote computer with time-varying memory resources is available to where functionalities can be swapped out.

Model-based Development

To apply the RAP or any other process model in combination with a model-based development approach, one applies textual and graphical DSLs in certain development phases to represent *domain models*. Those models make domain knowledge explicit, which is on the one hand relevant for a certain functional or architectural concern of the application under study, and on the other hand is important to be represented during a particular development phase. An example of a domain model is shown in Fig. 3. Those domain models are either created by humans supported by development tools (e.g. language workbenches and editors) or by run-time environments in an (semi-)automated manner. In both cases, domain models can be of various nature such as source code, configuration files, drawings or technical documentation to name a few, all of which are usually represented in heterogenous formats. Therefore, it remains challenging to compose those domain models technically and conceptually in order to infer answers about the system as a whole. Such a situation is exemplified in Fig. 2. Here, in the *platform building* phase the quadrocopter's computational hardware is modeled with AADL [11] yielding a textual model which makes connections and properties such as the number and size of physical memory explicit. However, some models can only be partially instantiated or not instantiated at all at design time as binding information is not (yet) available. For example, the concrete memory usage of an application is not known before deployment time and depends on the execution context. Therefore, several authors [12] [13] [14] argue that domain models need to be created, modified and eventually executed at run time. Subsequently, in the *functional design* and *capability building* phase top-level functionalities are identified, decomposed and modeled in terms of feature diagrams [15], basic and composite components. For example, the quadrocopter's perceptual ability to recognize fiducial markers could be modeled with a plethora of component models developed in robotics [3]. During the *system deployment* phase the mapping of components to computational units is modeled with DSLs such as [16] [17] [18]. Further, domain models do not necessarily remain isolated. In fact, as shown in Fig. 2 domain models do have implicit links refining some information, e.g. the link from the capability building phase to the functional design phase refines the information how a certain feature is resolved in terms of software components. However, all too often those links are not made explicit which prevents the systematic composition of domain models during design time and run time.

```
rpsl.sensor_component do
   name "camera"
   add_port :out, "out_port", "rgb_image"
end

rpsl.processing_component do
   name "aruco"
   add_port :in, "in_port", "rgb_image"
   add_port :out, "out_port", "marker"
end

rpsl.perception_graph do
   name "marker_detection"
   connect "camera", "out_port", "aruco", "in_port"
end
```

Fig. 3. A domain model of a marker detection system created by the DSL proposed in [19].

In summary, applying a model-based development approach throughout a complete development process is seldom done and it remains challenging

- to persistently store and compose heterogenous domain models in a unified, systematic manner,
- to query composed domain models originating from different functional domains and development phases, and
- to systematically modify and employ domain models also at run time.

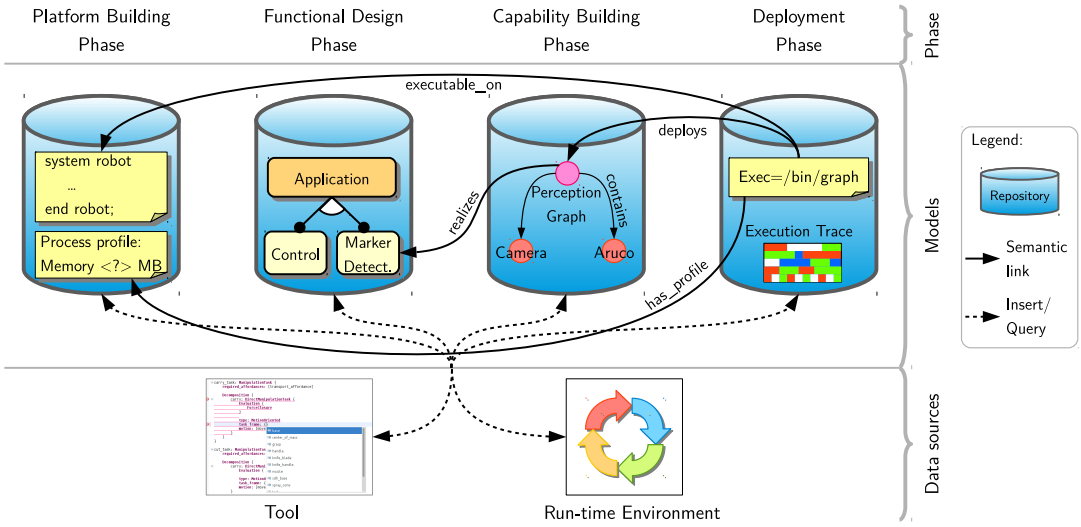

Fig. 2. A schematic representation of a labeled property graph containing heterogenous domain models, originating from different functional domains and development phases. Some domain models are semantically connected through links (e.g. `realizes` which denotes that the perception graphs realizes a feature). Human developers or run-time environments either insert new elements into the graph or update existing ones.

III. Graph-based Storage of Robot Domain Models

To store and compose heterogenous domain models we need to define some sort of common representation or *lingua franca* that describes those models. From a DSL developer's perspective this appears to be a somehow paradoxal situation as DSL developers usually aim to define very specific abstractions and representations. Nevertheless, we argue that such a *lingua franca* is crucial in order to persistently store and eventually compose domain models. Therefore, we propose to employ labeled property graphs. On the one hand graphs are well-studied, naturally preserve structure and on the other hand can be easily implemented. We define a labeled property graph $\mathcal{G}$ formally as a quadruple

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P}, \mathcal{L}) \tag{1}$$

where $\mathcal{V}$ are the nodes and $\mathcal{E}$ are the edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ of the graph. Additionally, the graph contains properties represented as key-value pairs ($\mathcal{P}$) and labels ($\mathcal{L}$). Arbitrary many properties $p \in \mathcal{P}$ can be attached to either nodes or edges. Similarly, arbitrary many labels can be attached to nodes ($l_v \in \mathcal{L}$) and edges ($l_e \in \mathcal{L}$).

This generic graph structure is not sufficient to enable semantic queries (see Sec. IV) or to enrich the graph with meaning. Therefore, we impose further constraints on the aforementioned labeled property graph. There must be one or more $l_e \in \mathcal{L}$ attached to any edge in order to give meaning to relations among nodes. Here the meaning is expressed by domain-specific labels which are either pre-specified ($\mathcal{X}$) or coming from the domain expert ($\mathcal{D}$), please note that $(\mathcal{X} \cup \mathcal{D}) = \mathcal{L}$.

To demonstrate the formal description we provide some examples based on Fig. 2 and Sec. II. For instance, in the *capability building phase* software components are modeled which we represent as *nodes* in the graph. As they represent components we further label them as *Component*. Similarly, atomic features, represented in the *functional design phase*, are labeled as *Feature*. Further, to link the *Aruco component* node with the *Marker detection feature* node one introduces two edges. The first one from *PerceptionGraph* to *Aruco* is labeled *contains* and encodes that a particular component is part of a larger architecture (here perception graph). The second one from the *PerceptionGraph* to the *Marker detection feature* with the label *realizes* encodes that a particular *Perception Graph* realizes a higher-level description of a functionality. In addition, properties of the nodes could be attached in the form of key-value pairs such as names of nodes, e.g. $\langle Name, Aruco \rangle$. The labeled-property graph model also supports the late binding of domain information in the form of blanked properties and uninitialized nodes. For example, in the *deployment phase* a description of the location and name of some executable is provided which we represent as a node in the graph. Depending on the deployment infrastructure or robot software framework this node is labeled as *DeploymentFile*, *RosLaunch* or *SystemDServiceFile* with a link to some *ProcessProfile* node which encodes execution properties of the deployed process. As the properties of the latter are not known before deployment, the values of the key-value pairs remain blank, e.g. $\langle MemoryUsage, - \rangle$ and $\langle StartTime, - \rangle$.

From Domain Models to Graph DB Models

To store the domain models in a graph database we need to both assess the domain models and translate them to the graph concepts discussed above. The assessment involves the question of which entities of the domain models should be represented as nodes and which should be represented as edges (relationships). For software-related domain models in robotics, such as component models, coordination models

(e.g. state-charts) or deployment descriptions the translation step is obvious. The core entities of interest should be represented as nodes, whereas edges are used to represent connections between entities. For example, states in a state-chart are represented as nodes and transitions between states are represented as edges. Following such an approach also paves the way for (semi-)automatic translations of domain models to graph database representations. Depending on the application scenario a developer also needs to decide which and how much information is translated from a domain model to a graph database model. Fortunately, the labeled property graph model does not impose any constraints here. Developers can either decide to completely translate domain models to graph concepts or to partially translate them, where a node simply contains a property which points to the location of the more detailed domain model, for example, on the disk.

Features of the Neo4j Graph Database

To implement the concepts described above we utilize the open-source graph database management system `neo4j` [20]. The `neo4j` graph database exposes the property graph data model with nodes and relationships as first-class citizens. By composing nodes and relationships into arbitrarily connected structures we do not depend on join operations (relational databases) or other customized operations (document-oriented databases) to infer connections between entitites. This allows us, both, to preserve the structure of domain models and to compose domain models through labeled edges. Similarly to other professional database management systems, `neo4j` supports full ACID (Atomicity, Consistency, Isolation, Durability) transaction rules. Further, database drivers for several programming languages (e.g. Java, Python, Ruby) are available which eases the integration on a robot system. A core feature of `neo4j` is the Cypher graph query language. Having Cypher at their disposal developers can declaratively query and update the graph database. More details are provided in the following section.

IV. Semantic Querying of Robot Domain Models

Up to now we showed how to store and compose domain models in a graph database. To retrieve and eventually update those models we need to employ some means to query the database. We propose to use the Cypher query language provided by `neo4j`. Although, Cypher is well integrated with `neo4j` other means to query the graph database exist (e.g. SPARQL [21]).

The general principle of a Cypher query on the graph is that of matching a graph pattern of the following form: $(A) \rightarrow [R] \rightarrow (B)$. Here A and B are nodes whereas R is an edge. By using such a statement in a `MATCH` clause the graph database retrieves those nodes and edges where there is an outgoing relationship (edge) between A and B of type R. This pattern is the general principle of querys which can be arbitrarily extended and combined with directed, undirected, optional and multi-step relationships among nodes and more advanced clauses such as `RETURN` for node/edge retrieval and `CREATE` for node/edge creation to name a few.

```
MATCH (f:Feature)-[*]-(c:Component)
WHERE f.name = 'MarkerDetect'
MATCH (p:Platform)<-[:EXECUTABLE_ON]-(d:Deployment)
WHERE p.name = 'Remote'
RETURN d;
```

Fig. 4. A query where several domain models are involved.

In Fig. 4 a simple, yet realistic example is given. Here, we query the domain model of Fig. 2 where features, components, platforms and deployment descriptions are represented (stored) as nodes and connected through labeled edges encoding the relations among these nodes. The query then retrieves deployment descriptions for those components having a relation to the marker detection feature and are executable on the remote platform. As in the context of semantic web technologies those queries are called *semantic queries* as they derive implicitly defined information based on the structural information expressed by the graph. This is achieved by making relations between domain models explicit. It is important to note that the *executable_on* relation links the platform domain with the deployment domain which subsequently allows to filter the results in both domains through additional `WHERE` clauses.

V. Case Study

As the core contribution of this work is not a single, monolithic system, but a general approach to store and query domain models, it is hard to quantitatively assess the contribution. Therefore, we report and discuss in the following some lessons learned which we gained by applying our approach in realizing a real system. We followed a model-based development approach to realize the application described in Sec. II and [10]. More precisely, we employed two DSLs [16], [19] to create domain models representing *a*) different marker detection configurations suitable for varying illumination conditions (see Fig. 3); *b*) their associated deployment descriptions encoding name and location of the executable; and *c*) the computational hardware of the quadrocoptor and remote platform used in the application. Those domain models are then stored in a graph database and queryied for the sake of adapting the system at run time.

Storage of Domain Models

To store the domain models in the graph database one needs to define a conceptual and technical transformation from domain model elements (e.g. `sensor_component` in Fig. 3) to elements of the labeled property graph. This requires an assessment of which domain model elements shall be represented as nodes, edges or properties. In order to perform such an assessment knowledge about both the domain models and the potential graph database application is required. In the context of this case study we translated the core first-class citizens of the DSLs (e.g.

```
MATCH(f:Feature {name: 'Marker'})<-[*]->(c:Component)
WITH DISTINCT c
WITH SUM(c.memory_demand) as MEM
MATCH (p:Platform)<-[:EXECUTABLE_ON]-(d:Deployment)
WHERE p.memory_available >= MEM
RETURN p;
```

Fig. 5. A query retrieving the platforms meeting the memory requirements.

`processing_component` in Fig. 3) to nodes and employed edges whenever we intended to describe relations between them (e.g. the *contains* relation in Fig. 2). Simultaneously, we investigated which queries are required for the case study and tested whether they are realizable and feasible with the proposed transformation. As the names of edges and nodes appear directly in the queries they need to be meaningful and consistent. For example, naming the relation *executable_on* (see Fig. 2) makes only sense when the edge is directed from the deployment description to a platform description and not vice versa. The technical transformation is achieved by using a templating approach which yields an automatic transformation from domain models to graph database operations such as node creation and others.

Integration of the Graph Database

We integrated the graph database containing the domain models in our adaptive marker detection architecture proposed in [10]. The architecture (see Fig. 6) follows the MAPE-K (Monitor, Analyse, Plan, Execute and Knowledge) reference architecture [22] known from the self-adaptive software engineering domain and constitutes the building blocks described in the following. The *context monitor* observes both the memory which is available on the platforms and the current illumination condition. The *marker detector selector* decides which marker detector shall be executed on which platform. The decision is driven by the available memory on the platforms ($\{p_1, .., p_n\}$) and by the current illumination condition. First, the selector employs the query shown in Fig. 5 to derive the platforms where all the components required for the marker detector feature can be deployed without violating the memory demands. Second, the selector employs the algorithm described in [10] to further find out which of the marker detectors ($\{m_1, .., m_n\}$) is suitable for the current illumination condition. Once a marker detector m_i and a platform p_i is selected the *deployer* stops the current marker detector and starts the new one if it is not already being executed. Further, to also enable some retrospective analysis of the executed marker detectors we are interested in storing some process meta information about the deployed marker detector. Similarly, as described in Sec. III we aim to store information about the *memory usage*, the process *start time*, and the *process ID*. This information is inserted and linked to the corresponding marker detector in the graph database by the *deployer*. In summary, two graph database operations are performed: *insertion* and *querying*.

Semantic Querying of Domain Models

The case study demonstrated the need to query domain models originating from different functional domains and development phases. By employing relatively simple queries (see Fig. 4) a developer can incrementally extend them (see Fig. 5) in order to derive the information required for the task at hand. In the same way a developer can cope with growing graph databases by concatenating several `MATCH` clauses. Also additional constraints can be easily included through more and advanced `WHERE` clauses. In the context of the case study we also developed queries

- to retrieve those components required to realize the marker detector feature, but which are deployable also on the remote computer,
- to check whether the marker detector feature can be deployed with different camera resolutions. That is, whether or not camera components (see Fig. 2) with different resolution properties are part of a perception graph realizing the marker detector feature,
- to retrieve those components required to realize the marker detector feature, but which have been deployed in the past and their average *memory usage* was below a certain treshold, and
- to check whether the CPU workload would exceed an application-defined limit when the marker detector and the flight control were both deployed on the same platform. CPU workload profiles of software components are either acquired at run time or have been annotated at design time.

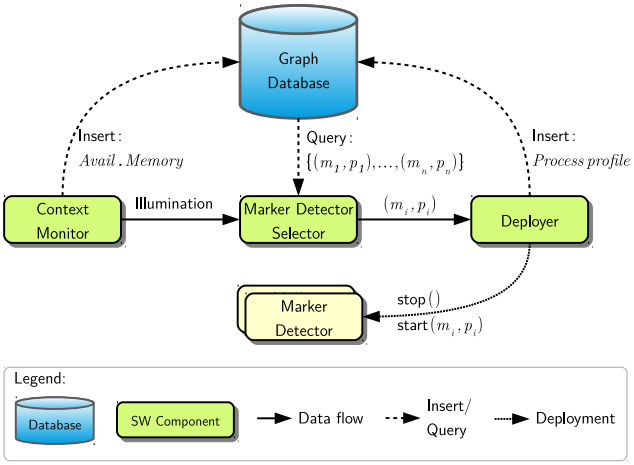

Fig. 6. The architecture integrating the graph database and realizing the adaptive marker detection application.

Run-time Overhead

One might argue that the application of a graph database system introduces a significant run-time overhead on the overall system. For this reason we designed the following experiment to investigate this overhead. We populated the graph database with N domain models (see Table I) of

potential marker detectors, all of them varying in different configuration properties. Please note, N does not denote the number of nodes in the graph. In fact, the number of nodes is approximately three times N as additional nodes are integrated e.g. encoding the deployment and platform. On a computer[1] with Linux Ubuntu 14.04 with Version 2.2.5 of the `neo4j` graph database we replayed different illumination scenarios described in [10], selected and deployed different marker detectors based on the domain models stored in the graph database. By doing so, we measured the time of the graph database operations (`query` and `insert`) and related them to the timing of the adaptation operations, namely starting and stopping an executable. Here, the executable is a C program implementing the marker detector. For each N, we repeated the experiment 100 times and report mean (μ) and standard deviation (σ) of those timings. As seen in Table I, the insertion operation seems to be independent of N. This can be explained with the fact that no graph traversal is required as the exact location of insertion is known, namely next to the selected marker detector. Interestingly, the impact of N on the timing of the query operation is rather limited as there is only one major increase from $N = 10$ with $4ms$ to $N = 100$ with $7ms$. To which extend the queries can be optimized through caching or other mechanism remains to be investigated and also depends on application-specific graph structures. In summary, the graph operations are more costly than the adaptation operations. However, this depends also on how the adaptation operations are implemented. For example, not preserving the state of a component as done in our experiments is faster than saving the state before stopping the component.

VI. Related Work and Discussion

The application of databases in robotics is not a new concept. In fact, Niemueller *et al.* [23] have shown that it is feasible to apply a document-oriented database like MongoDB, even for logging raw sensor data and analyzing robots' behavior in retrospect. Also knowledge-enabled and ontology-based approaches such as KnowRob [5], Robo Brain [6] or the OpenRobot Ontology (ORO) [7] rely on knowledge bases to store and query specifications of robots, their capabilities, tasks and environments. For the matter, this is complementary to the graph databases which we propose in this paper. Our efforts bridge the gap between *a*) the knowledge descriptions about robot's capabilities, tasks *etc.*; and *b*) the knowledge specifications about the *implementation* of the software that solves the tasks.

In the context of software models, we can build on the various model-enabled approaches and DSLs that already support software designers and domain experts. Some examples of MDE in robotics are the work of Schlegel *et al.* [24] for the specification of component-based software architectures, the work of Gherardi and Brugali [15] for the definition of functional reference architectures, the coordination of robot behaviors by state charts [25] [26] or building of system models via SysML [27]. In [18] the RobotML language is introduced which enables a domain expert to specify robot system architectures, communication mechanisms and the behavior of components. Interestingly, the development of RobotML was based on an ontology supporting the DSL designer by providing concepts specific to the robotic domain. The process of defining an ontology is somehow related to our process of defining a labeled property graph. However, in RobotML the ontology solely supports the DSL domain analysis, whereas in our approach the domain models carry their meaning into the robot's runtime.

Working with labeled property graphs revealed some analogy with the four-layered metamodeling hierarchy as described by the Object Management Group's (OMG) Meta-Object Facility (MOF) [28] which we would like to discuss. In the MOF the **M0** layer are real-world instances which are represented by models on the **M1** layer. The **M1** layer conforms to a metamodel on the **M2** layer which in turn conforms to a meta-metamodel on layer **M3**. We argue that a graph such as the one depicted in Fig. 2 is just a different representation of one or more **M1** models. Therefore, we derive the set of available properties $\mathcal{P}$ and labels $\mathcal{L}$ from our metamodels on the **M2** layer. Consequently, the graph in Equation 1 aligns with the meta-metamodel of the **M3** layer. Obviously the graph structure itself does not constrain the attachment of properties and labels to the nodes and edges. Thus, we first check the well-formedness and validity of specifications by relying on modelling tools and frameworks. Only then the valid models are transformed into the graph representation. To which extend **M1**, **M2** and **M3** models shall be stored in the graph database remains to be investigated and depends also on application requirements.

In robotics [12] [29] and software engineering [30] authors already investigated the application of software-related models for robots at run time. They demonstrated how software-related models can be employed to resolve dynamic variability faced at run time such as changing environment conditions and decreasing resources. To derive adaptation actions different adaptation principles are employed *e.g* constraint optimization methods [29]. Irrespective of the underlying adaptation principle, all run-time adaptation approaches in robotics need to access and query software-models originating from different domains and process phases. Therefore, we argue that our approach is a complementary building block for developing adaptive robot software architectures. More precisely, our work relates to the *knowledge* building block where domain models are placed in the well-known MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) [22] reference architecture.

By storing such models in a graph database we can on the one hand let the domain experts use their familiar, domain-specific tools and notations, while on the other hand providing a common interface to these models via semantic queries. Due to the inherent graph-based nature of most models, the storage in a graph database provides an integration point for further model-based approaches, for instance, semantic

[1]Intel Core i7-3632QM CPU 2.2GHz x 8 with 8GB RAM

	Graph DB Operations				Adaptation Operations			
	`insert`		`query`		`start`		`stop`	
N	μ	σ	μ	σ	μ	σ	μ	σ
10	26.861ms	5.936ms	4.083ms	1.274ms	1.491ms	2.299ms	0.066ms	0.023ms
100	27.766ms	6.385ms	7.323ms	3.198ms	1.602ms	1.906ms	0.072ms	0.025ms
1000	26.599ms	5.942ms	8.443ms	2.937ms	1.670ms	2.731ms	0.065ms	0.021ms
10000	26.308ms	6.208ms	7.901ms	2.476ms	1.651ms	3.345ms	0.074ms	0.082ms

TABLE I

TIMING RESULTS OF THE GRAPH DATABASE OPERATIONS VERSUS ADAPTATION OPERATIONS.

annotations to robot kinematics by Kunze *et al.* [31].

The growing interest in software engineering for robotics has already resulted in models and DSLs like the ones above. Thus, a consistent next step is to apply these models at run time. We have demonstrated that graph databases are a powerful ingredient to achieve this step. However, they are not the only one. Additionally, we need a better understanding of the implications of models on the run-time architectures as seen in our application. For example, which aspects of the run-time system should be represented and how to implement the required monitoring facilities such as probes for performance measurements.

Up to now, only very few systems (see Biggs *et al.* [27] for an example) have been realized by following a completely model-based development process. One reason for this could be the *DSL cacophony*, meaning that a vast number of (very relevant) DSLs exists, but their integration into an overall system remains challenging. We argue that not only harmonized meta-models, advanced language workbenches and tooling is required, but also a common interface of graph databases could offer a means of supporting this integration effort.

VII. CONCLUSION

In this paper we tackled the problem of granting robots access to software-related domain models at run time. We employed property graphs as a means of storing, composing and querying domain models. We demonstrated and discussed the feasibility of the overall approach in a real-world application and showed that the graph database imposes little overhead on the system's overall performance. In future work we aim to investigate to which extend graph databases can be also employed to share software knowledge among robots.

ACKNOWLEDGEMENT

Nico Hochgeschwender and Sven Schneider received a PhD scholarship from the Graduate Institute of the Bonn-Rhein-Sieg University which is gratefully acknowledged. Furthermore, the authors gratefully acknowledge the on-going support of the Bonn-Aachen International Center for Information Technology.

REFERENCES

[1] D. Brugali, "Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics," *Robotics Automation Magazine, IEEE*, vol. 22, no. 3, pp. 155–166, Sept 2015.

[2] A. R. da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139 – 155, 2015.

[3] A. Nordmann, N. Hochgeschwender, D. Wigand, and S. Wrede, "A survey on domain-specific modeling and languages in robotics," *Journal of Software Engineering for Robotics (JOSER)*, vol. 7, no. 1, pp. 75–99, 2016.

[4] "Robotics 2020: Multi-annual roadmap for robotics in europe," http://sparc-robotics.eu/wp-content/uploads/2014/05/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf, 2015, accessed: 2016-08-25.

[5] M. Tenorth and M. Beetz, "KnowRob – A Knowledge Processing Infrastructure for Cognition-enabled Robots," *International Journal of Robotics Research (IJRR)*, vol. 32, pp. 566–590, 2013.

[6] A. Saxena, A. Jain, O. Sener, A. Jami, D. K. Misra, and H. S. Koppula, "Robo Brain: Large-Scale Knowledge Engine for Robots," *International Symposium on Robotics Research (ISRR)*, 2015.

[7] S. Lemaignan and R. Alam, "Explicit knowledge and the deliberative layer: Lessons learned," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[8] G. K. Kraetzschmar, A. Shakhimardanov, J. Paulus, N. Hochgeschwender, and M. Reckhaus, "Specifications of architectures, modules, modularity, and interfaces for the brocre software platform and robot control architecture workbench," 2010, BRICS Project Deliverable D2.2.

[9] S. Garrido-Jurado, R. Muñoz Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recogn.*, vol. 47, no. 6, pp. 2280–2292, June 2014.

[10] N. Hochgeschwender, M. Olivares-Mendez, H. Voos, and G. Kraetzschmar, "Context-based Selection and Execution of Robot Perception Graphs," in *Emerging Technologies Factory Automation (ETFA), 2015 IEEE 20th Conference on*, 2015.

[11] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[12] A. Steck, A. Lotz, and C. Schlegel, "Model-driven Engineering and Run-time Model-usage in Service Robotics," in *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2011, pp. 73–82.

[13] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, pp. 22–27, 2009.

[14] L. Gherardi and N. Hochgeschwender, "RRA: Models and Tools for Robotics Run-time Adaptation," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, Sept 2015, pp. 1777–1784.

[15] L. Gherardi and D. Brugali, "Modeling and reusing robotic software architectures: The hyperflex toolchain," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, May 2014, pp. 6414–6420.

[16] N. Hochgeschwender, L. Gherardi, A. Shakhimardanov, G. Kraetzschmar, D. Brugali, and H. Bruyninckx, "A Model-Based Approach to Software Deployment in Robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).*, 2013.

[17] D. Alonso, C. Vicente-Chicote, F. Ortiz, J. Pastor, and B. Alvarez, "V3cmm: a 3-view component meta-model for model-driven robotic software development," *Journal of Software Engineering for Robotics*, vol. 1, no. 1, pp. 3–17, 2010.

[18] S. Dhouib, S. Kchir, S. Stinckwich, T. Ziadi, and M. Ziane, "Robotml, a domain-specific language to design, simulate and deploy robotic applications," in *Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, ser. SIMPAR'12. Springer, 2012, pp. 149–160.

[19] N. Hochgeschwender, S. Schneider, H. Voos, and G. K. Kraetzschmar, "Declarative Specification of Robot Perception Architectures," in *Proceedings of the 4th International Conference on Simulation, Modeling and Programming for Autonomous Robots*, D. Brugali, J. F. Broenink, T. Kroeger, and B. A. MacDonald, Eds. Springer International Publishing, 2014, pp. 291–302.

[20] "neo4j," http://neo4j.com/, accessed: 2016-02-25.

[21] "SPARQL," https://www.w3.org/TR/rdf-sparql-query/, accessed: 2016-02-25.

[22] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[23] T. Niemueller, G. Lakemeyer, and S. Srinivasa, "A generic robot database and its application in fault analysis and performance evaluation," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, Oct 2012, pp. 364–369.

[24] C. Schlegel, A. Steck, D. Brugali, and A. Knoll, "Design Abstraction and Processes in Robotics: From Code-Driven to Model-Driven Engineering," in *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, 2010.

[25] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using uml/p statecharts," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, May 2013, pp. 461–466.

[26] T. J. de Haas, T. Laue, and T. Röfer, "A scripting-based approach to robot behavior engineering using hierarchical generators," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, May 2012, pp. 4736–4741.

[27] G. Biggs, T. Sakamoto, K. Fujiwara, and K. Anada, "Experiences with model-centred design methods and tools in safe robotics," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, Nov 2013, pp. 3915–3922.

[28] omg, *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006, accessed: 2016-02-25. [Online]. Available: http://www.omg.org/cgi-bin/doc?formal/2006-01-01

[29] J. F. Inglés-Romero, A. Lotz, C. Vicente-Chicote, and C. Schlegel, "Dealing with run-time variability in service robotics: Towards a DSL for non-functional properties," *CoRR*, vol. abs/1303.4296, 2013. [Online]. Available: http://arxiv.org/abs/1303.4296

[30] F. Fleurey and A. Solberg, "A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, ser. MODELS '09. Springer, 2009, pp. 606–621.

[31] L. Kunze, T. Roehm, and M. Beetz, "Towards semantic robot description languages," in *IEEE International Conference on Robotics and Automation (ICRA)*, Shanghai, China, May, 9–13 2011, pp. 5589–5595.