The Role of Symbolic AI with Machine Learning in Robotics

London, May 2022

Tomás Sabat Stöfsel, James Fletcher, Wejdan Ismail, Vaticle
tomas@vaticle.com

Abstract

This article explores how to combine the approaches of symbolic AI with machine learning to achieve greater autonomy in robots. It discusses why a TypeDB knowledge base can provide the right framework for these types of knowledge-enabled robots. It then offers worked examples with code snippets in TypeDB demonstrating how to create robotics modules for environment models and task planning. Finally, this article offers a full fledged robotics schema that anyone can use in their own development environment.

1 What is Robotics?

Robotics is a multi-disciplinary field in computer science dedicated to the design and manufacture of robots, with applications in industries such as space exploration, defence, and manufacturing. While the field has existed for over 50 years, recent advances such as the Spot and Atlas robots from Boston Dynamics are truly capturing the public's imagination as science fiction becomes reality.

Traditionally, robotics has relied on machine learning techniques such as object recognition. While this has led to huge advancements, the next frontier in robotics is to enable robots to operate in the real world autonomously, with as little human interaction as possible. Such autonomous robots differ to non-autonomous ones as they operate in an open world, with undefined rules, uncertain real-world observations, and an environment — the real world — which is constantly changing. These differ to non-autonomous robots, which operate in a closed-world with defined rules, such as robots in factories.

This article explores how this type of autonomy is enabled by combining the approaches of symbolic AI with machine learning. It discusses why a TypeDB knowledge base provides the right framework for these type of knowledge-enabled robots. It then offers worked examples with code snippets in TypeDB demonstrating how to create robotics modules for environment models and task planning. Finally, using the examples from a paper from TNO, this article offers a full fledged TypeDB schema that anyone can test on their own.

2 Why Combine Symbolic AI with Machine Learning for Robotics?

Symbolic AI refers to the branch of AI research that involves the explicit representation of human knowledge in the form of symbols in computer programs. It was the dominant paradigm in AI research in the post-war era until the early 1990s. However, as this knowledge was represented in the form of rules which had to be explicitly written, symbolic systems weren't able to understand implicit knowledge or improve themselves over time, and the field fell out of favour. AI research became relatively obscure until the early 2010s, when machine learning became the most dominant AI paradigm.

Symbolic AI has primarily focused on building expert systems with rule engines that can be thought of as large system of *if-then* statements to draw conclusions. A rule is represented as sets of conditions that draw a conclusion when matching inputs are seen. Rules can trigger one another, so that a rule engine can compute arbitrary computation.

The upside of rules is also their downside: they are precise and generic rules which are not learned based on experience. Rules should only be added to a system if they will always hold.

Machine learning can be seen as a system of creating weak rules by looking at correlations over a dataset. In fact, machine learning is performing inductive reasoning! The field garners extensive use in robotics for good reason. Wherever the rules required for a task are too many for humans to write then ML is a great tool, for example in tasks such as object recognition.

The downside of learners is that they can typically only operate within the circumstances that they've seen in their training data. This means that achieving autonomous decision-making is difficult, due to the large number of permutations of unseen scenarios that the learner could encounter.

Therefore, the difficulty with learners is their limited generalisation. In contrast, the fundamental benefit of a symbolic approach is that it's able to generalise perfectly, as long as the rules hold.

As more people become aware of the limitations of machine learning, one of the most exciting developments in robotics (and the wider AI research community) is the combination of machine learning with symbolic AI (referred to by some as *neurosymbolic AI*). This means running machine learning algorithms over highly contextualised knowledge bases or knowledge graphs, instead of flat data. This gives the best of both worlds: powerful machine intelligence with logic-based knowledge that improves over time. Other key benefits include:

- Learning over reasoned knowledge improves the learner and reduces the amount of data required
- Improving model accuracy by implicitly embedding context of any data point into the learned model
- Improving the lack of interpretability of machine learning models (*why did the model make this prediction?*)

This combination of machine learning with symbolic AI holds massive and and untapped potential for autonomous robotics. To realise this potential, it becomes necessary to build powerful ontologies and knowledge bases that a robot's machine learning algorithms can leverage.

3 Why Use TypeDB as a Knowledge Base?

Such knowledge bases need to accurately represent the real world and reason over the robot's environment when executing tasks independently. Beetz et al (2012) called this a *knowledge-enabled robot programming paradigm*, which separates the knowledge from the application and modularises it into a robot's knowledge base.

Autonomous robots with knowledge bases require the integration of various types of data, for example sensor data, real world objects, planning schedules, and much more. Due to the complexity inherent in modelling such heterogeneous data, systems in robotics need a database that can accurately model their environment as they perform tasks and make autonomous decisions in real-world settings.

Because of this, some people have resorted to Semantic Web Technologies such as RDF/OWL to build knowledge bases for robotics. However, these technologies fall short as they lack a strong type system and native modelling constructs such as hyper-relations and type hierarchies, which makes them unable to accurately capture the complexity of the real world (a full comparison between Semantic Web and TypeDB can be found here).

TypeDB, on the other hand, makes knowledge-based robotics possible with a three-pronged approach: enabling the creation of symbolic representations, reasoning over them, and inferring new knowledge. TypeDB provides an expressive data model inspired by object-oriented principles, allowing the robot to model the real world and its constantly changing environment. It also offers the robot a built-in reasoning engine to make real-time inferences as it executes its tasks. As a result, TypeDB is a natural choice as a centralised knowledge base to orchestrate and reason over the open world data that an autonomous robot must work with.

4 How to Build a Knowledge Base for Autonomous Robotics

There are many ways to build robotic systems, from so-called sense-act architectures to more complex *sense-decide-act* ones. The degree of a robot's autonomy is based largely on the extent of its own reasoning and decision-making capabilities that try to mirror human cognition.

Autonomous robots function by executing a series of tasks, for example moving from one location to another. A robot must be able to create these plans autonomously — usually through a task planning module — and to do this, they need a description of their environment, provided by an environment model module. What follows is a demonstration of how these two modules can be built using a TypeDB knowledge base for an indoor robotics use case.

5 Environment Modelling

Building an environment model consists of various techniques that create representations of the real world. One of those techniques is object recognition. As the robot navigates through the real world, it obtains sensory features through its multi-modal sensors and uses pre-trained machine learning models to categorise objects.

Besides objects, the robot must also be able to locate itself. As with object recognition, it can use features of its environment and pass those through machine learning modules to determine its location (Lowry et al, 2016, Schubert, Neubert, 2021).

A knowledge-enabled robot identifies those objects and locations, and maps them to its knowledge base. This process, referred to as "symbolic grounding", is essential in creating symbolic representations of real world sensory features and creating contextual knowledge that a robot can use to reason independently (Weiss, 2019, Johnston et al, 2008, Vasudevan et al, 2006).

Traditionally, techniques such as Simultaneous Localisation and Mapping (SLAM) are used to build environmental models and determine a robot's location through geometric maps (Durrant-Whyte, Nailey, 2006). By creating powerful knowledge bases, however, a robot can make decisions independently, making resource-intensive approaches such as SLAM no longer necessary (Sijs, Van Vught, Voogd, 2020).

As a robot maps the observations it makes to its knowledge base, the work of Adão, Magalhães, Peres (2016) serves as a useful reference point to create ontologies of the real-world. In their work, a building is seen from three perspectives: *geometrical, physical, and functional.* The geometric view represents a room as points and lines in the form of a polygon. The physical view represents what things are actually located in a room — whether something is a door or a window. The functional view depicts the various types of rooms — whether it is a living room or a kitchen.

TypeDB's expressive type system makes it easy to build a model for such a system.

Using TNO's paper (Sijs, Van Vught and Voogd, 2020), Adão et al's model of a building can be created directly in TypeDB. In the geometric view, the polygon describes a room as straight lines that refer to a room's walls, and points that refer to corners. A room with a door would be described in the following way:

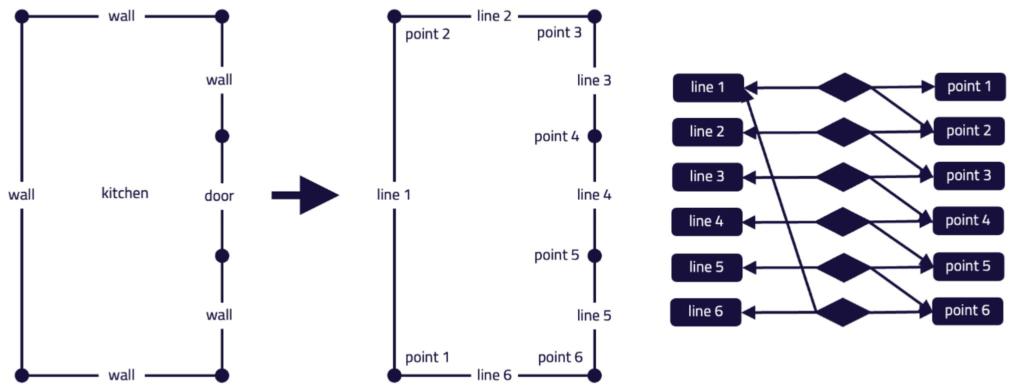

Figure 1 The figure on the left shows the kitchen plan that is mapped to a polygon. Walls and door are mapped to lines, corners are mapped to points. On the right is the TypeQL mapping of this polygon; a line is modelled as an entity playing the role of edge and two points play the roles of vertex in a "math-connecting" relation. This is an example of a ternary relation.

```
$line1 isa line;
$line2 isa line;
$line3 isa line;
$line4 isa line;
$line5 isa line;
$line6 isa line;
$point1 isa point;
$point2 isa point;
$point3 isa point;
$point4 isa point;
$point5 isa point;
$point6 isa point;

(vertex: $point1: edge: $line1, vertex: $point2) isa
    math-connecting;
```

```
(vertex: $point2: edge: $line2, vertex: $point3) isa
   math-connecting;
(vertex: $point3: edge: $line3, vertex: $point4) isa
   math-connecting;
(vertex: $point4: edge: $line4, vertex: $point5) isa
   math-connecting;
(vertex: $point5: edge: $line5, vertex: $point6) isa
   math-connecting;
(vertex: $point6: edge: $line6, vertex: $point1) isa
   math-connecting;
```

The schema required to represent this would be as follows:

```
abstract-entity sub entity;
  line sub abstract-entity,
    math-connecting:edge,
  point sub abstract-entity,
    math-connecting:vertex;
math-connecting sub relation,
  relates edge,
  relates vertex;
```

Leveraging a robot's object recognition modules, this *geometric view* can then be mapped to the *physical* and *functional view* of the building. For the polygon in Figure 1, `line3` can be mapped to a `door` (physical view) through the `taking-form` relation, while the `door` can be mapped to the `living room` and `kitchen` (functional view), through an `adjacent-room-connection` relation. The concept `door` is described as a sub-type of `connector`, in order to create other types of connectors in a building (e.g. stairs).

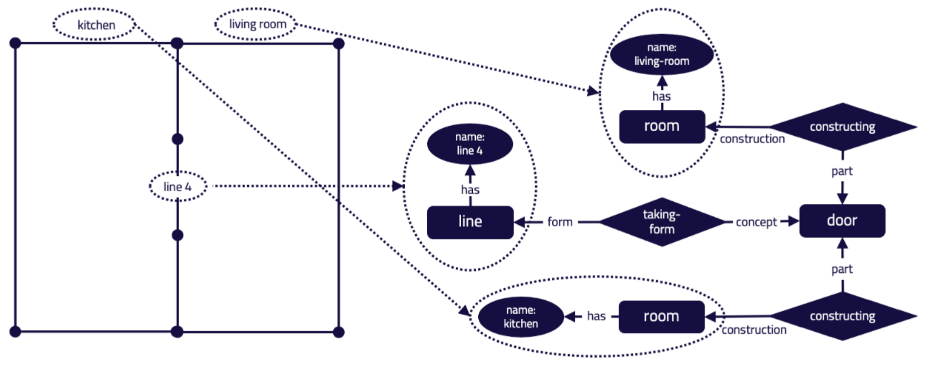

Figure 2 Line 4, which represents the door that divides the kitchen with the living room, is mapped to an entity of type "line" with a "name" attribute "line 4". This is connected through a relation of type "taking-form" with an entity of type "door". This "door" is connected to an entity of type "room" with name "living-room", and another entity of type "room" with a name "kitchen" through relations of type "constructing".

```
$door isa door;
$living isa room, has name "living room";
$kitchen isa room, has name "kitchen";

(concept: $line4, form: $door) isa taking-form;
(construction: $living, part: $door) isa constructing;
(construction: $kitchen, part: $door) isa constructing;
```

```
line sub abstract,
  plays taking-form:concept;

structural-element sub entity,
  plays taking-form:form,
  plays constructing:part;

  connector sub structural-element;
```

```
    door sub connector;
    stairs sub connector;

room sub entity,
   plays constructing:construction;

constructing sub relation,
  relates construction,
  relates part;

taking-form sub relation,
  relates form,
  relates concept;
```

With this schema, the robot can use TypeDB's built-in reasoning engine to infer new knowledge independently — it doesn't need to carry out any machine learning, or depend on human intervention. For example, the following `adjacent-room-connection` rule can be used to infer that two rooms are adjacent to each other other through a connector (e.g. a door or stairs):

```
rule adjacent-room-connection:
when {
  $room1 isa room;
  $room2 isa room;
  not {$room1 is $room2;};
  $connector isa connector;
  (construction: $room1, part: $connector) isa constructing;
  (construction: $room2, part: $connector) isa constructing;
} then {
  (place: $room1, place: $room2, connector: $connector) isa
      adjacent-room-connection;
};
```

With this rule, a robot can simply query its own knowledge base for "What is the connector between the living room and the kitchen?" By asking for `connector`, through (relation) type inheritance, the robot can query for any type of connector — in this case, `door` or stairs. With the query below, therefore, door would be inferred by the reasoner.

```
match
$living-room isa room-name, has name "living-room";
```

```
$kitchen isa room-name, has name "kitchen":
$connector isa connector;
(place: $living-room, place: $kitchen, connector: $connector) isa
    room-connection;
get $connector;
```

Even though the robot doesn't *know* that the kitchen and living room are connected through a door (the data is not persisted), with TypeDB the robot can *independently* infer this contextual knowledge. Therefore, with little effort — merely a single TypeDB rule — the robot can make predictions in a wide range of scenarios.

The full schema of the environment model is provided at the end of this article.

6 Task Planning

Extending the environment model, the robot also uses a TypeDB knowledge base for automatic task planning. To do this, a model can be built that describes the structure to understand and conceptualise the task planning network of a robot in a real-world setting.

Based on TNO's work in this paper, the following task planning model is based on various planning concepts, specifically the Planning Domain Definition Language, Hierarchical Task Network and Markov Decision Process.

In this approach, given a particular task, the initial and final state of a robot in a task can be represented respectively with the types `actual-state` and `goal`— for example, a robot's current location (actual state) and its desired location (goal or final state).

To get to its desired location, the robot performs actions that are modelled as `primitive-tasks`, which start operations that take the system from one state to the next. These are executed by `operators`, system components that enable the robot to execute its primitive tasks.

A `compound-task` models a hierarchy of tasks, of either other compound or primitive tasks. Primitive tasks can only be executed if certain conditions are met, represented through a relation `task-requirement` and an entity `contextual-condition`. The full schema is shown at the end of this article.

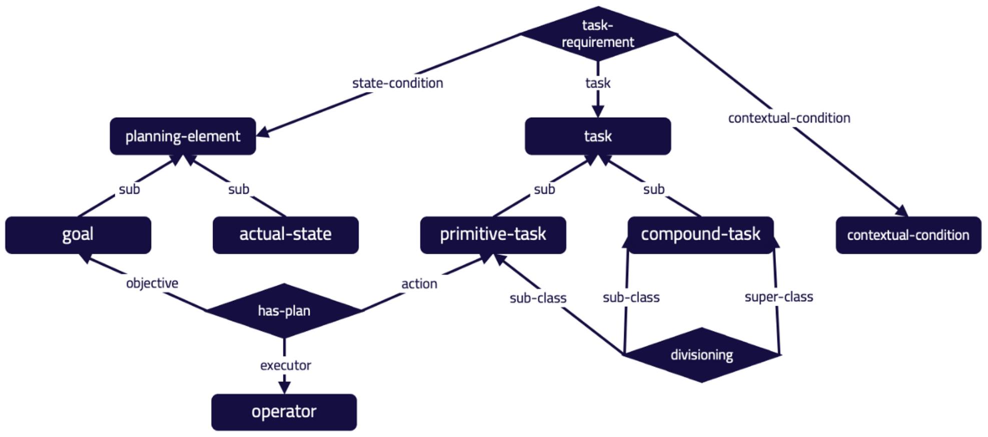

Figure 3 This conceptual model shows a section of the schema for the task planning model. It shows two type hierarchies: “planning element” and “task”. It shows the relation “has-plan”, which is a hyper relation and connects a “goal”, an “operator” and a “primitive-task”. The relation “division” models a hierarchy of tasks, either compound or primitive tasks. Finally, the relation “task-requirement” maps a “planning-element” to a “task” and a “contextual-condition” to denote how tasks can only be executed if certain conditions are met.

In the example that follows, a kitchen is connected to a living room through a door, and the living room is connected to the bedroom, but through stairs. Given a task where the robot needs to go from the kitchen to the bedroom, the robot can leverage TypeDB’s reasoning engine to autonomously create a plan.

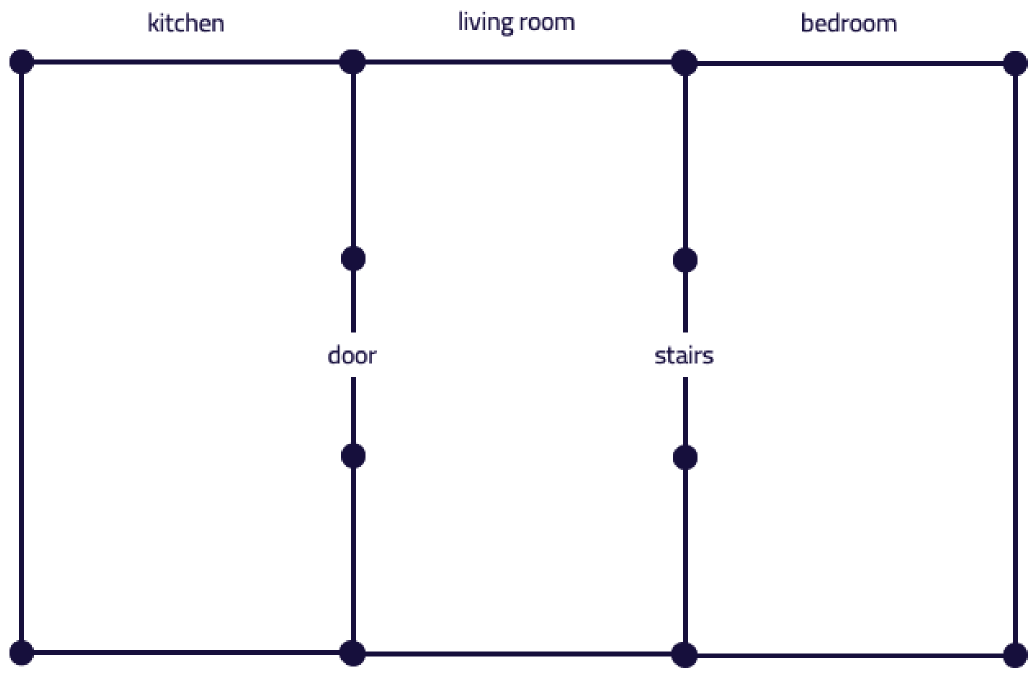

Figure 4 A kitchen is connected to a living room through a door, and a living room is connected to a bedroom with stairs.

To do this, a new relation `indirect-room-connection` is modelled as a sub-type of `room-connection`. This relation is inferred through the rule below if the two rooms are not directly connected. If room 1 is connected to room 2 through a door, and if room 2 is connected to room 3 through another door, then an indirect room connection is inferred between room 1 and room 3.

```
rule indirect-room-connection:
when {
  $room1 isa room;
  $room2 isa room;
  $room3 isa room;
  not {$room1 is $room3;};
  $connector1 isa connector;
  $connector2 isa connector;
  not {$connector1 is $connector2;};
  (place: $room1, place: $room2, connector: $connector1) isa
```

```
    adjacent-room-connection;
  (place: $room2, place: $room3, connector: $connector2) isa
    adjacent-room-connection;
} then {
  (place: $room1, place: $room3, connector: $connector1, connector:
    $connector2) isa indirect-room-connection;
};
```

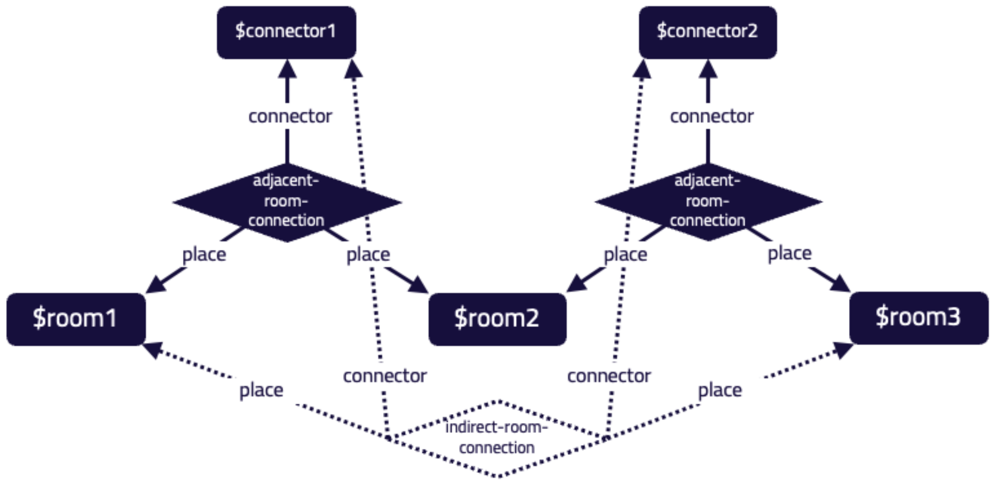

Figure 5 The “indirect-room-connection” rule infers an indirect room connection (inferred relation shown as dotted line) between two rooms if there is a room that they share a connector (door, stairs, etc) with.

The robot can begin to create its plan by checking if a connection exists between the kitchen and the bedroom. The robot can simply query for the relation `room-connection` between the two rooms. Through type inheritance, this can either be an indirect or adjacent room connection. The TypeQL query will be as follows:

```
match
$kitchen isa room, has room-name "kitchen";
$bedroom isa room, has room-name "bedroom";
$connector isa connector;
(place: $kitchen, place: $bedroom, $connector) isa room-connection;
```

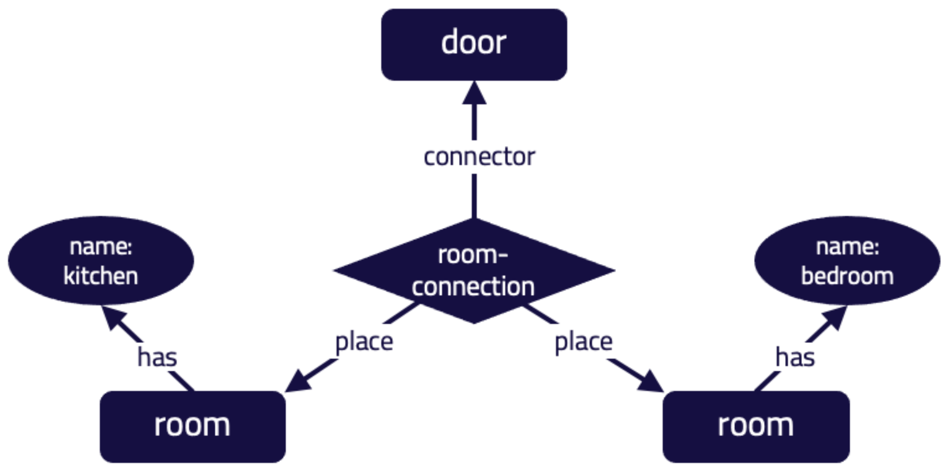

Figure 6 The "room" entity on the left has the name "kitchen", and plays the role of "place" in the relation "room-connection" to another entity "room" called "bedroom" which also plays the role "place"; the entity "door" plays the role of "connector".

The result of this query tells the robot that it needs to go through a door and stairs before it gets to its final location — the bedroom. With this result, it can then further learn that the door connects the kitchen to the living room, and the living room is connected through stairs with the bedroom. The robot is able to infer this knowledge independently without any human intervention before finalising its plan.

Before it can reach the bedroom, the robot needs to go from the kitchen to the living room, via a door. To do this, it can use its operator "exit via door", which would include a primitive task "go to adjacent room" that is dependent on there being a door connection between the two rooms. The plan and its contextual information is represented in the following TypeQL statement:

```
(current-situation: $actual-state, desired-situation:
    $possible-state, executor: $exit-via-door, action:
    $goto-adjacent-room, objective: $goal) isa has-plan;
```

```
$is-at (subject: $robot, location: $kitchen) isa is-at;
(state-element: $is-at, descripted-state: $actual-state) isa
    state-description;
(state-element: $kitchen, descripted-state: $possible-state) isa
    state-description;
(task: $goto-adjacent-room, state-condition: $possible-state,
    contextual-condition: $building-condition) isa task-requirement;
(condition: $building-condition, conditional-element:
    $room-connection) isa consists-of;
$room-connection (connector: $door, place: $kitchen, place:
    $living-room) isa room-connection;
```

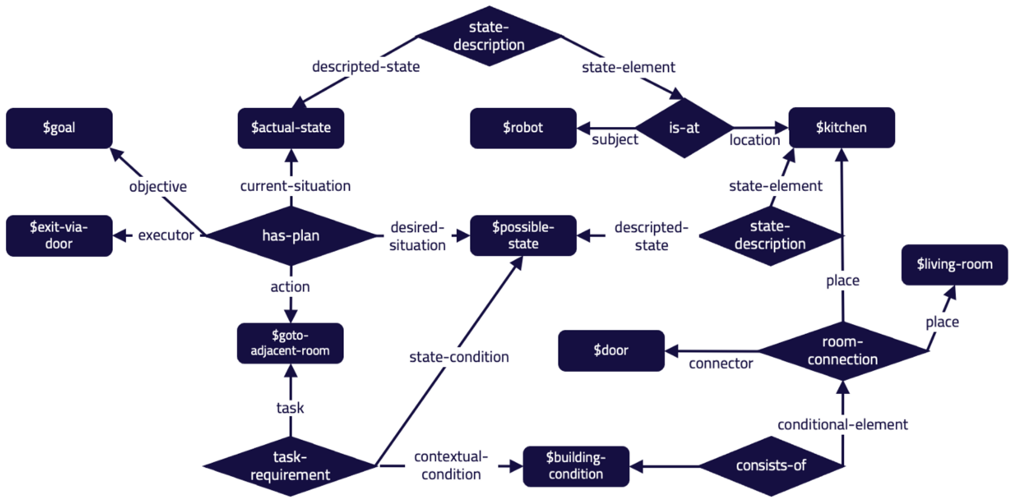

Figure 7 This is the visual representation of the query above. Note the “has-plan”, “task-requirement”, and “room-connection” relations, examples of hyper-relations, a key feature of TypeDB’s expressive type system.

Once it has arrived in the living room, it can create another plan to take it to the bedroom. To do this, it would create a similar plan, but using the operator `exit-via-stairs` and the primitive task `goto-other-floor`.

7 Conclusion

Building autonomous systems requires much more than just a knowledge base, naturally. An important step can be taken, however, by combining symbolic AI with machine learning. This gives robots the ability to reason independently without having to exclusively rely on expensive machine learning approaches.

Taking the earlier `room-connector` rule as a case in point, this symbolic reasoning approach avoids the need of analysing thousands of inputs through a machine learning algorithm to learn that the two rooms are connected, if they both share the same door. This can be seen as the equivalent to zero-shot learning, so naturally robotics should benefit from this symbolic AI approach. Encoding this sort of knowledge in a knowledge base like TypeDB takes little effort and covers a wide range of scenarios, bringing robotics one step closer to autonomy.

Finally, data in robotics is often incredibly heterogeneous as it needs to represent real world data, planning systems, hardware data, and much more. This necessitates a database system that can build the type of knowledge base which can model this data with all its semantic richness. This is why many robotics organisations use TypeDB to natively model constructs such as hyper relations (e.g. `room-connection`), nested relations (e.g. `state-description`), type hierarchies (e.g. `connector`), and much more.

8 TypeDB Robotics Schema

Below is part of the robotics schema according to this paper from TNO. Please refer to that paper for further reading. Many thanks to the TNO team for their great work so far. If you'd like to learn more, they have given several presentations that you can view here:

```
define
# Environment Model Module
robot sub entity,
    plays is-at:subject,
    plays has-plan:actor;
abstract-entity sub entity;
    polygon sub abstract-entity,
        plays mapping:top-view,
        plays math-constructing:mathematical-object;
    line sub abstract-entity,
        plays math-constructing:edge,
        plays math-connecting:straight-connection,
        plays taking-form:concept;
    point sub abstract-entity,
        plays math-connecting:vertex;
physical sub entity;
    building sub physical;
        office sub building;
        residential-house sub building;
    space sub physical;
        open-space sub space;
        closed-space sub space;
            room sub closed-space,
                owns room-name,
                plays constructing:construction,
                plays is-at:location,
                plays state-description:state-element,
                plays room-connection:place,
                plays mapping:enclosed;

structural-element sub physical,
    plays taking-form:form,
    plays constructing:part;
    connector sub structural-element,
        plays room-connection:connector;
        stairs sub connector,
          owns stairs-name;
        door sub connector,
          owns door-name;
    wall sub structural-element;
constructing sub relation,
    relates construction,
    relates part;
is-at sub relation,
  relates subject,
  relates location;
mapping sub relation,
  relates enclosed,
  relates top-view;
room-connection sub relation,
  relates place,
  relates connector,
  plays consists-of:conditional-element;
  indirect-room-connection sub room-connection;
  adjacent-room-connection sub room-connection;
taking-form sub relation,
  relates form,
  relates concept;
math-constructing sub relation,
  relates mathematical-object,
  relates edge;
math-connecting sub relation,
  relates straight-connection,
  relates vertex;

name sub attribute, value string, abstract;
stairs-name sub name;
door-name sub name;
room-name sub name;

# Task Planning Module
planning-element sub entity;
    goal sub planning-element,
        plays desiring:desired-goal,
        plays has-plan:objective;
    state sub planning-element,
        plays task-requirement:state-condition,
        plays state-description:descripted-state;
        possible-state sub state,
            plays desiring:desired-state,
            plays has-plan:desired-situation;
        actual-state sub state,
            plays has-plan:current-situation;
    operator sub planning-element,
        plays has-plan:executor,
        plays action-realizing:realizing-operator;
        exit-via-door sub operator;
        exit-via-stairs sub operator;
    task sub planning-element,
        plays task-requirement:task;
        primitive-task sub task,
            plays action-realizing:realized-action,
            plays has-plan:action,
            plays divisioning:sub-class;
            goto-adjacent-room sub primitive-task;
            goto-other-floor sub primitive-task;
        compound-task sub task,
            plays divisioning:sub-class,
            plays divisioning:super-class;
            goto-room sub compound-task;
    contextual-condition sub planning-element;
        building-condition sub contextual-condition,
            plays task-requirement:contextual-condition,
            plays consists-of:condition;

has-plan sub relation,
    relates objective,
    relates desired-situation,
    relates current-situation,
    relates actor,
    relates action,
    relates executor;
state-description sub relation,
    relates state-element,
    relates descripted-state;
is-at sub relation,
    relates subject,
    relates location,
    plays state-description:state-element;
task-requirement sub relation,
    relates state-condition,
    relates contextual-condition,
    relates task;
desiring sub relation,
    relates desired-goal,
    relates desired-state;
action-realizing sub relation,
    relates realized-action,
    relates realizing-operator;
divisioning sub relation,
    relates sub-class,
    relates super-class;
consists-of sub relation,
    relates condition,
    relates conditional-element;

# Rules
rule adjacent-room-connection:
when {
  $room1 isa room;
  $room2 isa room;
  not {$room1 is $room2;};
  $connector isa connector;
  (construction: $room1, part: $connector) isa constructing;
  (construction: $room2, part: $connector) isa constructing;
} then {
  (place: $room1, place: $room2, connector: $connector) isa
     adjacent-room-connection;
};

rule indirect-room-connection:
when {
  $room1 isa room;
  $room2 isa room;
  $room3 isa room;
  not {$room1 is $room3;};
  $connector1 isa connector;
  $connector2 isa connector;
  not {$connector1 is $connector2;};
  (place: $room1, place: $room2, connector: $connector1) isa
     adjacent-room-connection;
  (place: $room2, place: $room3, connector: $connector2) isa
     adjacent-room-connection;
} then {
  (place: $room1, place: $room3, connector: $connector1, connector:
     $connector2) isa indirect-room-connection;
};
```