Connecting Conceptual Models using Relational Reference Attribute Grammars

René Schöne
TU Dresden, Germany
rene.schoene@tu-dresden.de

Johannes Mey
TU Dresden, Germany
johannes.mey@tu-dresden.de

Sebastian Ebert
TU Dresden, Germany
sebastian.ebert@tu-dresden.de

Uwe Aßmann
TU Dresden, Germany
uwe.assmann@tu-dresden.de

ABSTRACT

Model-driven engineering can be used to create problem-specific, conceptual models abstracting away unwanted details. Models at runtime take this principle to the time a system is running. Connecting and synchronizing multiple models creates several problems. Usually, models used at runtime must communicate with other systems over the network, they are often based on different paradigms, and in most settings a fast and reactive behaviour is required. We aim for a structured way to define and organize such connections in order to minimize development cost, network usage and computation effort while maximizing interoperability. In order to achieve those goals, we present an extension of the paradigm of models based on reference attribute grammars by creating a dedicated problem-specific language for those connections. We show how to connect several runtime models to a robotic system in order to control this robot and to provide guarantees for safe coexistence with nearby humans. We show, that using our approach, connections can be specified more concisely while maintaining the same efficiency as hand-written code.

CCS CONCEPTS

• **Software and its engineering** → *Publish-subscribe / event-based architectures*; **Model-driven software engineering**; **Interoperability**; *Domain specific languages*; • **Theory of computation** → *Grammars and context-free languages*; • **Computer systems organization** → External interfaces for robotics.

KEYWORDS

Reference Attribute Grammar, CPS, Multi-Paradigm Modeling

ACM Reference Format:
René Schöne, Johannes Mey, Sebastian Ebert, and Uwe Aßmann. 2020. Connecting Conceptual Models using Relational Reference Attribute Grammars. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3417990.3421437

MPM4CPS 2020, October 18–20, 2020, Montreal, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3421437

1 INTRODUCTION

Constructing a cyber-physical system (CPS) requires knowledge from various fields such as computer science, electrical engineering, or legal scientists ensuring safety regulations. Likewise, many different people are involved that need to work together and understand each others languages. Thus, finding the right abstractions – or models – is crucial. However, it is often not possible to find one formalism to describe all aspects of a system leading to multiple models in different formalisms - a problem tackled by Multi-Paradigm Modelling [36].

This work aims to realize those cases involving multiple CPS by offering a way to connect systems – more precisely models used by those systems – to other models in an automatic and generic way. As a formal basis, this work focusses on relational Reference Attribute Grammars (relational RAGs) [28], a formalism extending RAGs [19]. Models specified with this formalism unite the definition of structure (by employing grammars and relations) and computation (using attributes). We identified three challenges those systems face and that we want to tackle with our approach:

Distribution. All models involved in the CPS are either local on one machine or are located remotely somewhere in the network. Reasons include performance, i.e., balancing computational load, requirements to special hardware, or security constraints. Therefore, a connection mechanism must be able to transparently communicate with local and remote endpoints.

Multi-Paradigm. Other external models are likely to use other formalism or paradigms. Thus, they are also likely to use other programming languages (if any). Therefore, the connection mechanism must be both paradigm- and language-independent, or at least support sufficiently many languages. As stated in [18], "more flexible approaches are needed in order to address the heterogeneity problem for an open set of modeling languages."

Fast, reactive behaviour. Computations of RAG-based models are normally on-demand, i.e., the computation must be triggered somehow. This is not desirable for an automatic connection mechanism, therefore it must support behaviour similar to reactive programming. In our use case, the robot arm must slow down its speed at the very moment the safety model recognizes it, and not at some evaluation step started later.

One application area of CPS is robotics. The ability of tactile interactions with robots is subject of one decade of research and industrial application [1]. This topic has gained further attention in recent years, because modern robotic systems are able to perform more complex manipulation tasks and thus they are potential co-workers for humans in many scenarios. To enable this form of cooperation, robots have to be aware of nearby humans and self-adapt their functionality to prevent any kind of harm [16].

In our use case, a robotic arm is supposed to safely execute a number of tasks such as moving to certain positions in order to actually execute the task as soon as it has reached this position. Orthogonally, the robot must respect its environment and react to changes accordingly.

One way of achieving the latter are safety zones. They describe areas in which the entry of any entity – human or robot – must lead to some action ensuring safety of the environment. The selection of those actions can be controlled by a designated safety model [16], defining mappings from safety endangering events to such actions. Assurance of safe *coexistence* of robots and humans is the requirement to enable the safe *collaboration* between humans and robots on specific tasks.

In our use case, we focus on safe coexistence. Thus, the speed of robot arm movement is immediately reduced once any part of the robotic arm enters a safety zone while moving to a certain position. As soon the arm has completely left the safety zone, its speed is restored to a normal level. This allows a person to move safely around the robotic arm and allows the robot to fulfil its assigned tasks. Figure 1 shows a simulated robot during its movement and a visualization of safety zones.

The remainder of this work is structured as follows. After introducing background knowledge in section 2, sections 3 and 4 detail the design process and its result, while section 5 shows the implementation. In section 6, we evaluate how our approach works for the use case. Section 7 classifies our work within related works and section 8 concludes this paper.

2 BACKGROUND

The Robot Operating System (ROS) is an open-source middleware which facilitates the development of robotic applications. ROS runs on top of Linux and saw an enormous growth in popularity over the past years, not only in research but also in industry. ROS 2 as the latest version of ROS addresses the shortcomings of its predecessor and introduces real-time support and newly designed programming interfaces [22]. It allows the creation of components which are able to control robotic hardware and of pure software components. In order to make this possible ROS provides a structured communications layer on top of the host operating systems [30]. A ROS-based process is called node. Inside such a node data-processing and control takes place. Nodes are connected based on topics or a service-client pattern, thus they are able to send and receive data with messages [30].

The core of ROS provides several nodes inside of components, communicating based on topics or by advertising services. These components are for example concerned with message passing, management of system-wide parameters and visualization sensor data and robotic motions. There are also many components extending

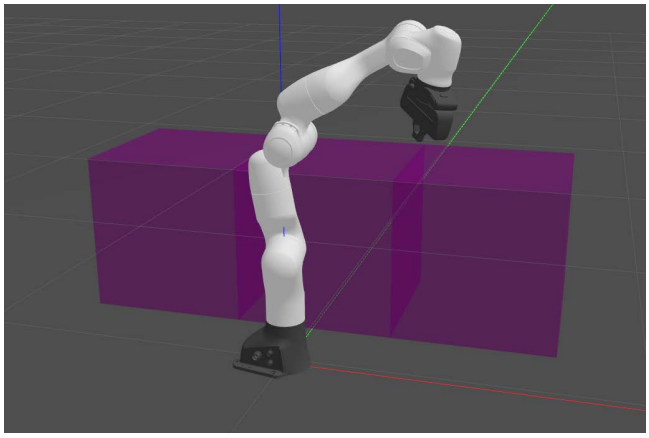


Figure 1: Simulation of a robot and defined safety zones (in purple)

Listing 1: Example grammar with four nonterminal types

```
A ::= One:B ;     // Class A with normal child "One" of type B
B ::= <T:int> ;   // Class B with token T of type int
C:B ;             // Class C extends B
D:B ::= E* ;      // List child of type E (0..* multiplicity)
```

the core of ROS. The MoveIt [11] platform provides components for motion-planning, robotic grasping and simulation. Therefore, a system build based on ROS, contains multiple models, for example to describe robots and their motions.

Attribute Grammars [24] originate from the domain of compiler construction. An attribute grammar specification comprises two parts: First, a context-free grammar is used to describe the structure of the tree by defining possible types of nodes and their children, which can be nonterminal nodes or tokens (i.e., *intrinsic* attributes). Secondly, *computed* attributes define the semantics of abstract syntax tree (ASTs) of a grammar by specifying equations for nonterminal nodes. They can either be synthesized or inherited [24]. Examples of a grammar and its attribution are shown in listings 1 and 2, respectively.

Reference Attribute Grammars (RAGs) [19] allow attributes to point to other nonterminal nodes which, together with the introduction of intrinsic non-containment relations [28] enables the usage of ASTs as conceptual models. Modern RAG frameworks, such as JastAdd [20], offer further features like incremental evaluation [35] or higher-order attributes [37].

3 CONNECTING RAG-BASED MODELS WITH OTHER MODELS

This section investigates ways to connect RAG-based models to other systems, discussing our design, in particular goals, possible options and the process leading to our DSL described in section 3.4.

3.1 Design Goals

To address the challenges mentioned in section 1, and to arrive at an automatic and generic approach, we aim for the following goals.

Listing 2: Example attribution for the grammar in listing 1

```
syn int D.d();     // D declares synthesized attribute d
eq D.d() {         // D has equation defining d
  return getT();   // Right-hand side of equation
}
```

Minimize development effort. Runtime model developers usually need to implement connections between models themselves. Thus, the primary goal is to minimize effort for them. This reduces their workload, but also minimizes errors as generated code is based on the input models.

Minimize network usage. Messages sent using connections must be as small and few as possible to reduce load on the network which is shared between all systems. Usually, systems comprise many sensors each sending their updates over the network. New messages based on those sensor values must only be sent, if they contain new information. This goal allows to use the approach in situations where only little network bandwidth is available.

Minimize computation effort. If no unnecessary computation is processed, results can be propagated more quickly and enable faster overall communication and reaction times. This requires knowing about dependencies of a computation to only compute if new information is available.

Maximize interoperability. As the approach must be generic, it has to be either general enough or easily extendible. Given that, there will be no constraints which external models can be connected to a RAG-based model.

Design goals tend to conflict with each other, e.g., maximize interoperability could involve more computation to support different models, or generated code could be less efficient than hand-written code but requires less development effort. This leads to different design options which are discussed in the following.

3.2 Design Options

There are several aspects where multiple options in the design are possible. We will list them here and discuss their implications.

Scope of connection endpoints. To synchronize distributed, heterogeneous models, several means for information exchange are possible. Due to the potentially different modelling languages and paradigms of the models, a loose coupling using the exchange of update messages is considered. Since this work focuses on grammar-based models as described in section 2, this leads to the question, which parts of the RAG-based model can be a connection endpoint. Conceptually, there are some options: the update of single tokens, entire subtrees, or arbitrary sets of either in a single transaction.

Considering the employed grammar specification presented in section 2, tokens have either primitive (value) types or immutable reference types.[1] Thus, they can be easily updated (and in most cases serialized), but constrain the possible operations because the structure of the AST remains constant. However, RAGs offer means to alleviate this shortcoming: higher-order attributes can be used to derive subtrees from tokens.[2]

Updating entire subtrees offers more flexibility as this includes updating tokens, but induces additional problems. While replacing subtrees of a tree is straightforward, because it is a simple replacement of the subtrees root node, the relational RAG approach adds additional problems, since inconsistencies can occur when there exist (potentially bidirectional) relations into the subtree. Furthermore, a synchronization on the subtree level opens the possibility of overlapping synchronization scopes, requiring additional validation of synchronization relations and increasing the complexity of update management.

Finally, an even more versatile, yet complex, synchronization approach could use transactions to group several updates of the previously described kinds, inheriting their complexity. Update transactions are a requirement for higher-level updates, such as partial updates of trees or unification of multiple models.

Model Connection Mechanisms. The models considered in this work are distributed, follow different paradigms, and, in case of externally defined, existing models, may use different modelling frameworks. This leads to the questions how an adequate connection mechanism is integrated into the modelling framework and how the connection is actually established.

In general, creating a connection between different models can be done in three ways. The first way, *manual, model-specific implementation*, uses the host language of the modelling framework and manually defines the connection points for the specific use case. This might suffice for small examples, but has to be done from scratch for every new use case. Thus, it involves high effort due to its low reusability.

The second way is a *generic connection library* for the modelling framework defining general connection directives. As such a library is agnostic to the specific model, it must make conservative assumptions, e.g., prohibiting further (error) analysis. More importantly, the envisioned *reactive* behaviour is hard to realize, as the intrinsic evaluation mechanisms are fix for RAGs.

As a third way, a *model-specific generative approach* is possible by defining a new configuration language used to weave in required methods directly into the node types of the grammar. This approach combines the other two in that it is specific to the use case, but requires the runtime model developer only to specify the essential parts for the connection. Further it can be extended by the developer of this new language to support new communication methods, such as MQTT [32] or ZeroMQ [2].

Regardless of how the connection is established, there are several communication patterns that can be employed to transmit update messages. One important decision is whether to use a *peer-to-peer* infrastructure with channels established between each sender and receiver of updates or to use a *message broker* as a central entity to distribute the messages. While the first approach has the benefit of less communication overhead, establishing connections is complicated, since additional discovery mechanisms such as a directory service are required. Furthermore, scaling to multiple systems

[1] In *JastAdd*, mutable data types can be used, but this is discouraged as it can lead to undesired behaviour during attribute evaluation.

[2] In principle, the token could contain a serialized version of the subtree which is then expanded by the higher-order attribute. There are some restrictions, though, when using *relational* RAGs

receiving updates from different senders requires separate connections between all of them.

A second method is to use a message broker and the publish/-subscribe pattern. Here, a central system (usually named "broker") is known to all other systems. Information is sent using categorized messages which are distributed by the broker to all receiving systems subscribed to the matching category. This method has benefits, if there are more than two systems and if those systems don't necessarily know each other. However, it requires a central broker possibly degrading overall performance because of the added indirections.

3.3 Design Decisions

After evaluating different options, this section shows how RAG-based models are connected to external models by means of a dedicated domain-specific language (DSL). For this, we briefly discuss the taken decisions based on the discussion in section 3.2, followed by an introduction of the designed DSL.

Because we want to connect to external models, a flexible, extensible, and adaptive approach is needed. Thus, to stay generic and to support many modelling paradigms, a generic message-passing concept is used rather than higher-level connection or synchronization concepts. As an initial step towards model synchronization, we decided to use tokens as connection endpoints to not introduce new problems with the other options, which is already quite powerful when combined with the features of RAGs, as described in section 3.2. Furthermore, a DSL was designed as a specification mechanism for the *model connection* to achieve the greatest reduction of development effort and at the same time the highest interoperability. The selected communication pattern is publish-subscribe, since it offers a lean paradigm and scalable implementations. As an initial implementation, we use broker-based communication, which however is easily exchangeable due to the different interpretation of the new language.

3.4 A connection DSL

Figure 2a provides an overview of a system comprising multiple connected models – the case study is described in detail in section 4. To describe the various types of connections between models, we employ two essential concepts: the definition of the connection endpoints and mappings of the transmitted data.

Connection Endpoint Definitions. The minimal information required to define connection endpoints for a model are: the location of the endpoints, their direction, and how the synchronization should be performed. In our case, the locations differ slightly for both directions. For incoming synchronization, tokens are used (which are also referred to as *intrinsic* attributes); these tokens are updated whenever new information is received. For outgoing connections, in addition to tokens, also *computed* attributes can be used – this enables reactive behaviour, since their value may depend on the state of the entire model – including tokens updated by incoming connections. The related question concerning "how" to update is referred to the second kind of concept, reusable mapping definitions described below. Furthermore, an update definition can configured to always trigger, or only when an actual change of the computed value is detected.

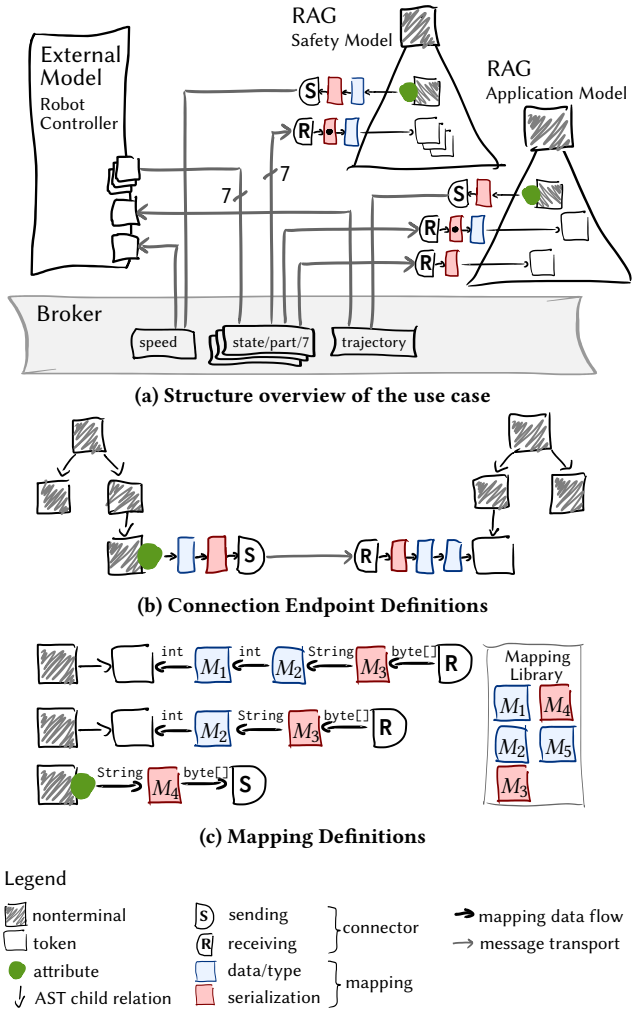


Figure 2: Structure overview and DSL Concepts

Figure 2b illustrates the connection between two RAGs. The left-hand side grammar contains an attribute (shown as a dot next to the nonterminal node) marked as an outgoing connection with an S. On the receiving side, a token is marked with an R. Between the marking and the part of the grammar, sequences of mapping definitions are shown.

The definitions of an endpoint *type* is done in a DSL statement using the following syntax. Note that symbols ending with *-name* are defined names of the respective type in the RAG.

⟨*endpoint*⟩ ::=
  ⟨*direction*⟩ ⟨*nt-name*⟩ '.' ⟨*token-name*⟩ ';'
| ⟨*direction*⟩ ⟨*nt-name*⟩ '.' ⟨*token-name*⟩ `using` ⟨*mappings*⟩ ';'
⟨*direction*⟩ ::= 'send' | `receive`
⟨*mappings*⟩ ::= ⟨*mapping-name*⟩ ',' ⟨*mappings*⟩ | ⟨*mapping-name*⟩

Examples of endpoint definitions can be found in listing 4. The (optional) mapping names refer to the concept of mapping definitions described next.

Value and Type Mapping Definitions. During an update, there can be situations when either the type or format of the information does not fit, or some calculation within a type is needed to get the wanted resulting update. For all cases, a mapping needs to be defined. In our case, those mappings are reusable to avoid high development effort and redundant (thus error-prone) definitions. This way, a mapping has to be defined once and can be reused in different occasions. A special use case is the (de-)serialization of messages received and sent by some communication protocol. For the same message type, this computation should only be written once and later be reused. A mapping has an input and output type, is identified by a unique name for later reference, and specifies the mapping using the host language referring to the input by a selected variable name. An option would be to add a dedicated expression language instead of reusing the host language. This ensures side-effect-freeness, but could constrain the runtime-model developer if more functionality is needed than provided by the given expressions. Furthermore, to support frameworks like Protocol Buffers, method calls need to be supported, which dwarfs the advantage of such an expression language. Therefore, we have decided to reuse the host language to specify the actual mapping definition. Nevertheless, input and output types can still be used for analysis and error checking of specified mappings.

Figure 2c gives examples for mapping definitions for both directions. Two different purposes of mappings are distinguished. Mappings M_3 and M_4 are used for (de-)serialization, whereas the other mappings are used to transform the value (M_1) or the type (M_2) of submitted information. The proposed syntax is as follows.

⟨*mapping*⟩ ::=
⟨*mapping-name*⟩ 'maps' ⟨*type-use*⟩ ⟨*ident*⟩ 'to'
⟨*type-use*⟩ '{:' ⟨*mapping-content*⟩ ':}'

The actual definition of the mapping function is done in the ⟨*mapping-content*⟩, which uses the value of ⟨*ident*⟩ as input. Examples of mappings definitions can be found in listing 4.

The following section illustrates the concept and usage of the connection DSL by describing a detailed case study using the models shown in fig. 2a.

4 A DISTRIBUTED MODEL-BASED ROBOTIC APPLICATION

To show the feasibility of our approach, we have implemented and evaluated the example case study as shown in fig. 2a.[3] It was performed using the *Gazebo* simulator [25] to increase reproducibility, facilitate development, and visualize the safety zones. [4]

This section subsequently introduces the involved models and their interactions. Ultimately, a system as shown in fig. 2a will result. The actual external model of the robot is on the left side and has information on its speed, position and target trajectory. Both safety and application model are RAG-based models. The broker mediates messages between all three systems based on the message topic. A RAG-based model comprises the AST, where some tokens are marked to receive information and other to send them. Additionally, mappings are used to transform incoming and outgoing data.

[3] An executable artefact is available at connector.relational-rags.eu.
[4] Note that the use case runs on both the simulator and on real hardware.

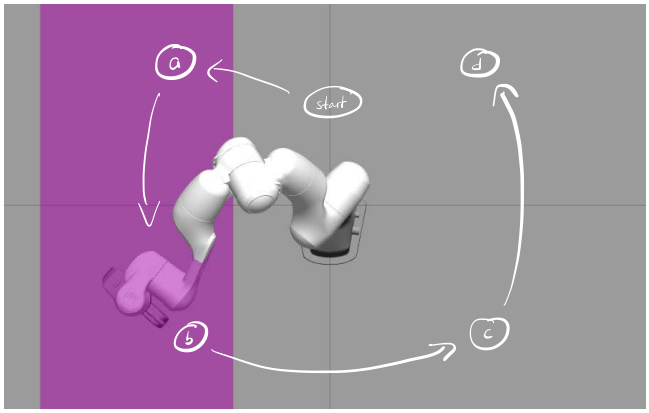


Figure 3: Robot workfow: perform work at four locations

4.1 Hardware and low-level Robot Control

To control a robot in a cohabitation scenario, low-level robot control and path planning capabilities are required, which are provided by several components the ROS infrastructure shown in the left part of fig. 2a. Those of the components concerned with planning robot motions, i.e., computing the inverse kinematics required to move a robot along a trajectory while respecting various constraints, use models to represent the robots shape, dynamic behaviour, and surroundings relevant to detect collisions. In ROS, these are the *Unified Robot Description Format (URDF)* [9], the *Semantic Robot Description Format (SRDF)* [10], and a *planning scene* containing collision objects and kinematic constraints. These models are synchronized with the physical state of the robot and can provide it to external components.[5] To connect these models with external models, we have constructed an MQTT connector that provides the robot state and accepts *trajectories* to be executed by the robot as well as a *speed factor* to configure the velocity of the robot using an extension to the motion planning system that adapts the trajectory execution. Note that the robot state is updated at a high frequency.

4.2 A Safety Context Model

As mentioned in section 1, collaborative robots can increase productivity and widen potential application areas of robotics. Therefore, a safety model is added to the system that is able to adapt the behaviour of the robot to an environment. In the presented case this means an adaptation of the robot movement speed to the area it operates in. Listing 3 shows the context: a work area of the robot is partitioned into zones, some of which are highlighted as *safety zones* which require a reduced robot speed. These zones could, e.g., be positioned in areas where both robot and human co-worker operate. Like the application model, the safety model requires access to the robot state, but in this case to all movable parts, since it should be checked for all of them if they are within a safety zone. Additionally, a connection to the robot control model is required to update the allowed speed. Since the component receives very frequent updates for the states of all parts of the robot, it is important that these are

[5] The state is provided both in joint space (how much each joint is rotated) and Cartesian space (the Cartesian coordinates of each component of the robot). Note that the latter is computed using information in the URDF robot model.

Listing 3: The grammar used for the safety model

```
Model ::= RobotArm ZoneModel ;
ZoneModel ::= <Size:IntPosition> SafetyZone:Zone* ;
Zone ::= Coordinate* ;
RobotArm ::= Link* EndEffector /<NewSpeed:double>/ ;
Link ::= <Name:String> <CurrentPosition:IntPosition> ;
EndEffector : Link;
Coordinate ::= <Position:IntPosition> ;
```

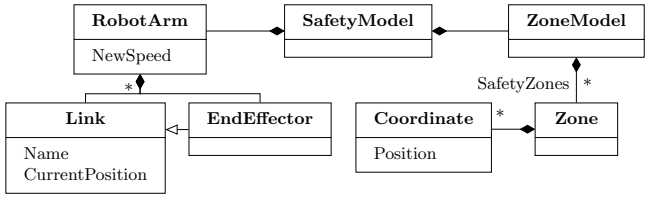
Listing 4: Using the DSL for the safety model

```
// --- update definitions ---
receive Link.CurrentPosition using ParseState, Transform;
send RobotArm.NewSpeed
  using CreateSpeedMessage, SerializeRobotConfig;
// --- mapping definitions ---
ParseState maps byte[] bytes to RobotState {:
  return RobotState.parseFrom(bytes);
:}
CreateSpeedMessage maps double speed to RobotConfig {:
  return config.Config.RobotConfig.newBuilder()
    .setSpeed(speed)
    .build();
:}
```

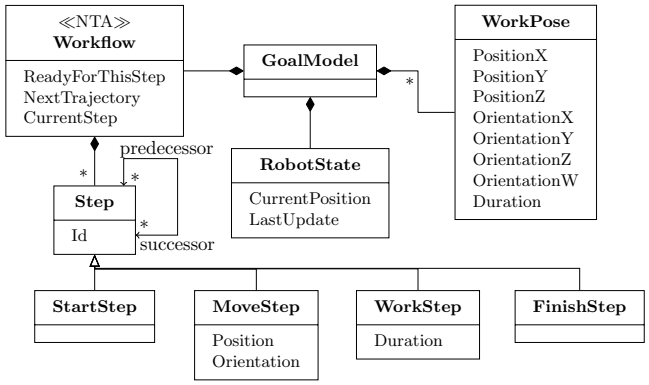
handled efficiently. Therefore, the mapping definitions presented in section 3 are used to transform the received Cartesian coordinates into zone coordinates. Since the model is only updated when the result of the final mapping in the chain, i.e., here the zone coordinate, has actually changed, this happens much less frequently than the overall updates. Therefore, the simplified state in combination with the mappings vastly reduces the computation effort required in the model, as shown in section 6. The model to realize this is shown in listing 3 using a textual syntax and fig. 4a using elements similar to a UML class diagram. Here, a robot arm comprises links between joints and an end effector, e.g., a gripper.

Listing 4 shows the update definitions and two of the required four mapping definitions used in the safety model. Here, the current position of every *Link* is defined to be updated by an external model, and every computed update for the speed of the robot arm shall be sent to external models. To accomplish this, incoming messages containing the position are first parsed using the method provided by Protocol Buffers [15], and then transformed into a suitable format (not shown). To send the new speed, the raw number is used to build an object of type *RobotConfig*, which is converted into an array of bytes (not shown). Note, that the computation of the speed depends on the current position of all links, such that updating one of those positions with a new value triggers this computation. However, only if a different speed value than the current one is calculated, it is actually sent to the external model.

Computing the new speed is based on the current state of the AST, as shown in listing 5. The attribute *isInSafetyZone* is defined for the *RobotArm* and iterates over its links and end-effector checking for each whether they themselves are inside any safety zone. Finally, either a low or high value is returned, depending on within or without a safety zone, respectively. This value could be more fine-grained and involve another, complex computation taking into account the current speed, objects in the zone or other context information. However, for our simple use case, two values suffice.



(a) Safety model.



(b) Goal model.

Figure 4: Grammars for the running example. Containments, relations and inheritance are in UML syntax, nonterminals are classes, and tokens are members of those classes.

Listing 5: Attribution for the safety model

```
syn boolean RobotArm.isInSafetyZone() {
  for (Link link : getLinkList())
    if (model().getZoneModel()
        .isInSafetyZone(link.getCurrentPosition()))
      return true;
  return model().getZoneModel().isInSafetyZone(
    getEndEffector().getCurrentPosition());
}
syn boolean ZoneModel.isInSafetyZone(IntPosition pos) {
  for (Zone sz : getSafetyZoneList())
    for (Coordinate coordinate : sz.getCoordinateList())
      if (coordinate.getPosition().equals(pos))
        return true;
  return false;
}
syn double RobotArm.getNewSpeed() {
  return isInSafetyZone() ? LOW_SPEED : NORMAL_SPEED;
}
```

To actually create a connection for parts of the application, the code shown listing 6 could be used. Here, one link is created and set up to receive updates from the external robot model and the computation of the speed for the robot is set to sent its update to a topic "`robot/speed`".

4.3 A Model Controlled Robotic Application

Thus far, the robot provides a control interface but is otherwise static. Therefore, a third, model-based component is provided, here called *application model*. This model, shown in fig. 4b, is defined as a relational RAG and consists of two parts. First, a *goal definition* declaratively describes a set of tasks the robot should perform. In

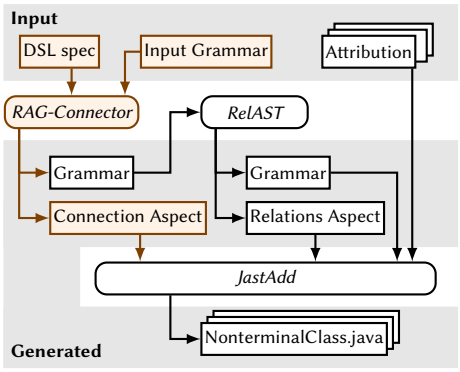


Figure 5: Code Generation Process, Contributions of this paper highlighted in orange

this very simple scenario, these tasks comprise a pose of the robot and a duration for which the robot should perform work in this pose. Real world examples for such tasks could, for example, be soldering jobs or simply inspection jobs, recording close-up images of a work piece. Note that the tasks are independent of each other, so they could be performed in any order, though there may be a preferred order, e.g., the one that required the least amount of time.

The execution order of the tasks is therefore specified by a second part of the model. Here, this model is defined as a state machine, potentially allowing alternative execution sequences. This second model is derived from the goal model using a higher-order attribute [37]. In our case, this attribute uses a greedy approach always selecting the closest pose, but more sophisticated algorithms could be implemented, such as the shortest path (cf. the travelling salesman problem) or a use-case specific order. The state machine works purely reactively: since the pose of the robot and the current state in the state machine both are known to the model, an attribute can compute the trajectory to be sent to the robot control model. As a result, no external inputs or control other than the state updates sent from the robot controller are required to perform the task – the reactive model acts completely independent, utilizing only its attributes and the presented synchronization mechanisms.

4.4 Summary of the use case

In the provided case study all challenges identified in section 1 occur. The issue of distribution exists, since there may exist different resource requirements for different models; particularly the robot control component and model must run close to the actual hardware to maintain a very fast and low-latency connection to the robot. Likewise, if a simulator is used, this imposes high computational resource requirements, also demanding specific hardware. These requirements of cyber-physical systems are not always met in a single resource – and might in some cases be contradictory. The support for multiple modelling paradigm is also required in the presented case: all models have very different purposes and thus also must be described differently. Finally, reactive behaviour is required both for the (event-triggered) application model and the safety model; in the latter case, response time is also essential to ensure the safety of the system.

Listing 6: Creating a connection for the safety model

```
RobotArm robotArm = ...;
Link link1 = ...;
robotArm.addLink(link1);
robotArm.addDependency1(link1);
link1.connectCurrentPosition("state/part/7");
// true to send current value immediately
robotArm.connectNewSpeed("robot/speed", true);
```

5 GENERATING CODE FROM THE DSL

In this section, we detail the inner workings of our approach.

As described in section 3.4, a DSL is used to define which tokens are used for updates and how their values are transformed using mapping definitions. To implement such a DSL and add functionality to the formalism, one either has to extend the underlying RAG-framework or transform the given definitions to a plain RAG specification. We decided to employ the second variant and therefore built a preprocessor taking a grammar and an instance of the DSL as input and produce a *JastAdd* specification. This has the advantage of reusing existing, validated tools, i.e., relying on their correctness and maturity. However, as detailed later, it also has the disadvantage of not being able to change all internal processes of the RAG-framework.

The complete process is shown in fig. 5. Actually, our approach does not directly output artefacts conform to *JastAdd* but instead for another preprocessor handling relations (RelAST) [28].

The implementation of our approach has to ensure that the resulting specification actually employs reactive behaviour, which is one of the identified challenges listed in section 1. In particular this means that upon changing some token, all necessary attributes involved for all sending update definition have to be triggered. As their dependencies are kept internally in *JastAdd*, we employed a workaround by requiring the developer to explicitly specify the dependencies of those attributes. Such a dependency can be specified in the DSL as shown below.

```
RobotArm.NewSpeed canDependOn Joint.CurrentPosition
                  as dependency1;
```

With this information, it is possible to react to changes of tokens.

Internally, wrapper code is generated for every token that receives updates. This code is executed after the token was updated and triggers all computations set to depend on this token. Similarly, whenever an attribute of a sending update definition computes a new value, additional code ensures that this value is actually sent. In both cases, caches are used to check, if the new value differs from the old one and an action has to take place at all.

Mapping definitions are transformed to simple methods that are invoked when needed. For code generation, we employ the template engine *Mustache* [38] using an higher-order attribute as input containing only information needed for that generation step.

6 EVALUATING CONNECTED RAG-BASED MODELS FOR THE USE-CASE

In this section, we will show, how well our approach can implement the use case described in sections 1 and 4 in terms of quantitative and qualitative metrics.

6.1 Observations for the Use Case

Although the system in our use case contains three models, application and safety model work completely independent of each other. Figure 6 shows recorded values from one run of the use case. As shown in fig. 3, the robot moves to four different points, stopping at each of them to work for four seconds and then moves on. Around and between the first two positions a safety zone slows down the robot's movement, as shown in fig. 3. In fig. 6a, the position of the end effector of the robot over time is shown. Figure 6b shows the resulting speed, whether the robot is in a safety zone, the velocity scaling factor (sent from the safety model) and the executed steps (at the bottom of the diagram, sent from the application model). Here, the velocity scaling factor is a factor between 0 and 1 which is used by the planning component of ROS to adjust the time to move to the next point on its way. Several observations can be made: (1) the safety model reacts fast and correct; (2) the speed is reduced within the safety zones, and is reverted to normal outside; (3) there is some movement in the work step, because the robot not quite at its target position yet, (4) perhaps surprisingly, the robot moves out of a safety zone on its way to ⓑ, because it moves "above" the safety zone (which ends at 0.5 m); (5) also surprisingly, the robot moves back in a safety zone on its way to ⓒ, in this case, because the planner has decided a more sweeping route - an effect that can and will happen when using heuristic, sampling-based planning [34]. Especially for the last point, the advantage of having separate models becomes clear: one could expect not to move through safety zones only knowing points ⓑ and ⓒ, and thus possibly leave out those safety checks.

6.2 Addressing the Design Goals

In this part, we will demonstrate if and how the goals derived from the challenges have been achieved. The programming effort required to efficiently connect different models should be minimized by the choice of the generative approach to construct the connections. Assuming the individual models to be already defined, each connection type simply required a one-lined update definition and potentially one or more mapping definitions (which consist almost entirely of program logic). Each connection instance is established with a single command[6] as shown in listing 6. In addition to the very concise way to describe connections between model, it can be noted that also the model semantics can be defined concisely as shown in listing 5, using the declarative attribute specification methods provided by the RAG system, such as collection attributes, higher-order attributes, and circular attributes (circular attributes are not required in the presented case study). Comparing the total lines of code needed to specify the connection aspect, it can be seen from table 1, that for instance the application model only needs 18 lines which get expanded to 281 lines of Java code. For comparison, all generated Java code for the complete application model comprises over 2500 lines of code.

Since the RAG-based models memoize computed attributes, they are able to reduce the number of messages sent between models, thus, reducing network load. An example for this can be observed in the safety model only updating the robot speed in the other models when it has actually changed. During the run shown in fig. 6, more than 38 000 position updates lead to exactly six speed updates.

[6]Note that in the current implementation, additional statements are required to define dependencies. This is a purely technical problem and can be solved by employing the existing mechanisms for dynamic dependency analysis in *JastAdd*.

Table 1: Lines of Code written in the DSL, and generated code

Input files	Application model	Goal model
Definitions in DSL	18	28
Aspect code (generated)	89	100
Java code (generated)	281	701
Total Java code (generated)	2534	3647

The main benefit related to this, however, shall be the minimized computational effort. Besides fewer messages that must be processed, the message processing itself happens in a filtered manner and stops if the transformed message resembles the previous one. A good example for this is the update of the safety zone states. While the exact position of the robot changes continuously, the model is only updated when the mappings derived a new safety zone for a part of a robot, which happens much less frequently. During the run shown in fig. 6, from the thousands of raw position updates, only 54 lead to a recomputation of the safety zone check, which in turn lead to the aforementioned six speed updates. Although this does not matter in our simple use case, for more complex computations it might reduce computation cost significantly.

Finally, the case study shall illustrate the interoperability. The general selected message-based communication model is widely supported by many programming and modelling paradigm. In practice, the choice of MQTT and Protobuf as very common technologies further supports this. As explained in section 3.2, those technologies can be also exchanged if needed to even further increase interoperability. Also, note that changing them does in principle not affect existing definitions specified in our DSL, except the concrete methods to be invoked, e.g., for Protocol Buffers. Furthermore, interoperability between models is also facilitated by the concept of mapping definitions, which allow a reusable specification of transformations required to connect heterogeneous models.

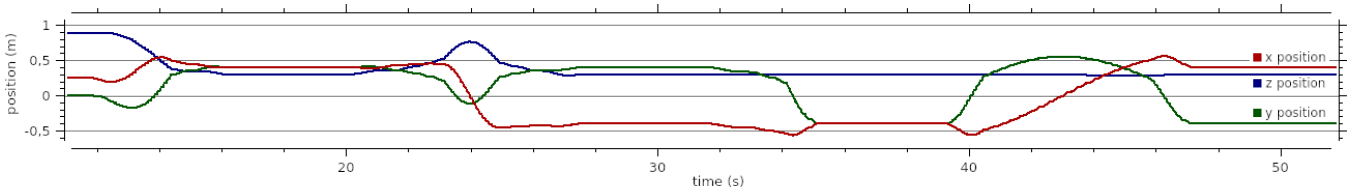
6.3 Discussion

Using the insights won through the case study, some conclusion on the strengths, weaknesses as well as suitable application areas of the proposed approach can be drawn.

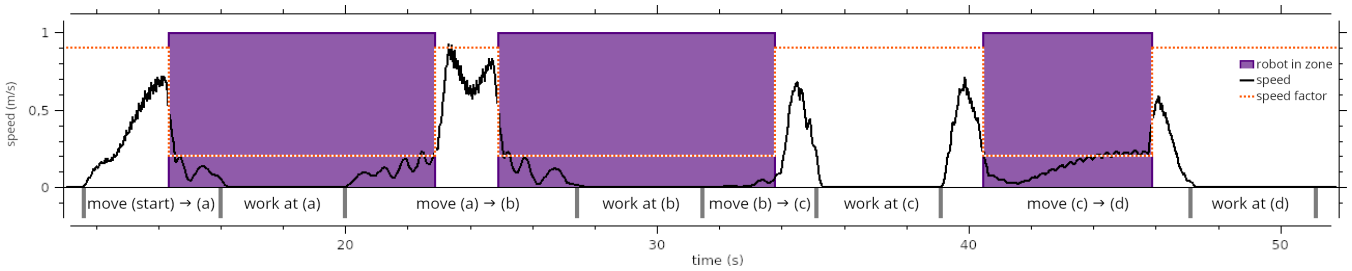
Strengths. The major advantage of our approach is from our point of view the explicit definition of connections between models. With that option, connection logic is not hidden in the code and can be easily checked or updated to new requirements. Furthermore, using the features of RAGs, the connection types and instances can be subjected to static and runtime analysis, respectively.

Additionally, helpful features like caching of intermediate transformation results of mappings can be generated from those definitions, minimizing both computational effort and network usage.

Weaknesses. The major weakness of our approach is the current (prototypical) implementation, which does not meet all expectations of our concept. Dependencies between attributes and their usage tokens have to be added manually creating a small, yet unnecessary



(a) x, y, and z coordinates of the robot end effector



(b) Speed of the end effector, (measured) presence of the robot in a safety zone, and speed scaling factor computed by the safety model

Figure 6: Workflow execution by the robot

development effort. The needed information are already present in *JastAdd* using incremental evaluation [35], but on the one hand require complex changes in the underlying framework itself, and on the other hand might introduce new corner cases when having an "eager" attribute evaluation. Examples of such cases are that during a long attribute computation new received messages could trigger other or the same computation in parallel, or that multiple update definitions might be affected by the change of one token value and their messages to be sent have to be ordered.

Unlike other model synchronization approaches [27], we focus on low-level connection endpoint definitions based on messages.

Opportunities. With our approach, the different paradigms mentioned in section 1 can be used together in a CPS. Models of those paradigms can be kept separate, but at the same time be connected during runtime.

As mentioned in section 3, there are different design options worth exploring. Because of our generative approach, we can still extend our solution to support new features later, e.g., let complete subtrees be targets of updates. Likewise, as mentioned earlier in this section, we are able to support other communication protocols or other serialization formats than Protocol Buffers by simply exchanging needed parts, either of the generation or in the mappings.

A last opportunity awaits when connecting more models, especially ones not implement by ourselves. We believe, that it is easily possible to connect to other systems using an appropriate serialization format. Then, new use cases can be developed using RAG-based models in combination with other, existing models.

Threats. Currently, the resulting system shown in our use case is reactive, but without being able to guarantee real-time behaviour. This might be needed when deploying models to real production sides. However, neither of the used systems (MQTT, ROS) and approaches (JastAdd and our approach) support real-time behaviour. Furthermore, as shortly mentioned in the beginning of section 4, ROS uses several models to represent the state of the robot. However, only a fraction of this knowledge is exposed, but possibly more could be exploited, e.g., for collision detection in our use case.

7 RELATED WORK

This work is touching many research areas: reactive programming, CPS, multi-paradigm modelling, (complex) event processing, stream processing, robotics, models at runtime, adaptive systems. In the following, we describe related work, compare to our approach and the three challenges distribution described in section 1.

Classification within surveys. To classify this work better within the mentioned fields of research, we shortly describe two survey papers, one from CPS and another from Multi-Paradigm Modelling. The first survey [4] attempts to classify approaches processing flows of information, which includes CPS, complex event processing and database systems, based on different aspects. Please see the survey for details of the following classification. Our approach uses the AST as a knowledge base, processes single elements for both selection and consumption, and does not feature recursion nor a dynamic rule language (only compile-time). It uses a central deployment and employs push for observation, notification and forwarding. As for items, we handle data in the format of objects, use a heterogeneous nature of flow and do not support uncertainty. Time is stream-only, and our transforming rules are not probabilistic.

As a second survey, [18] classifies different approaches for multi-paradigm modelling. Our approach is one addressing the heterogeneity problem. More specifically, we aim to compose models, i.e., "the coherent coupling of several heterogeneous models" [18]. In the same category, other related work was presented.

Using *ModHel'X* [17], different heterogeneous parts of a model can be specified to interact. Similar to our approach, such models are viewed as black-boxes and only communicate through interfaces. However, they do not focus on distribution and reactive behaviour, but rather one holistic view on the complete model and relations

between parts of it using interaction patterns. Another related work is the '42' approach [26] for specifying models of computation for embedded systems. They focus on fine-grained temporal behaviour and how to synchronize different components, thus neglecting distribution. An important part is reactive behaviour by specifying inputs, outputs and their connection for both data and control.

Connecting different models at runtime. In a previous work [31], the connection of RAG-based models to smart home middlewares and machine learning algorithms was focus of research. However, the connection mechanism was implemented manually there. Similar to mapping definitions presented in our approach, special connectors were used in [31] to capture the semantics of inputs and outputs of machine learning algorithms. Distribution was not focus there, but two different paradigms were connected and a reactive behaviour was realized using a MAPE feedback loop [23].

Dávid et al. [7] investigates a related problem, where frequent sensor changes needed to be handled by a model using complex event processing. They transform the needed event patterns into deterministic finite automatons, and employ VIATRA [3] for pattern matching. Hence, their focus is on fast, reactive behaviour and partly on distribution, while neglecting multi-paradigm modelling. Comparing to our approach, they have more sophisticated description of our mapping definitions, but focus only one model.

Combining different *models of computation* (MoC) is the idea of Ptolemy II [14] and Ptera [8]. A MoC describes, how parts (*actors*) of a system shall be executed and implies certain abstract semantics. Actors are similar to what we refer to as a model. In contrast, they have more sophisticated understanding of characteristics of those actors and possibilities of their composition, hence tackling also multi-paradigm. However, they do not focus on distribution (which might violate characteristics) nor on interoperability (connecting to other systems not conforming to their view of actors and directors).

Model-based adaptive robotics. Approaches in this field focus on adapting structure and behaviour of a robotic system at runtime based on their environment. The approach by Zhong and DeLoach [39] includes two models for distributed agents describing the current configuration of a system abstractly and the systems objectives. A system based on these models can act reactive.

The work of Inglés Romero et al. [21] models system variability, context, reasoning and architectural configuration in four dedicated models, based on different meta-models. Thus, it is a multi-paradigm modelling approach. These models are used at runtime to adapt the system architecture to context-changes reactively.

RRA [13] uses a set of orthogonal models representing a system architecture, variability and its current context. Additionally, it provides algorithms that reason continuously about the knowledge represented in the models, to adapt the robotic system reactively.

All three approaches are not applicable in environments where models are physically distributed, and differentiate between model updates and architectural reconfiguration, whereas our approach is more generic, because it treats all model synchronizations equally.

Ziafati et al. state that ROS does "currently [...] not provide much support for the implementation of required high-level event-processing operations" [40]. Thus, they describe three problems for processing sensor information: controlling behaviour of sensor components/their parameters, managing and processing data from sensors to additional relevant information, and reacting to sensor information. These problems can be mapped to the phase M, A and P of a MAPE loop [23]. Furthermore, they list different preprocessing operations similar to our mapping rules.

In [5, 6], an extension of ROS called *DyKnow* to allow changing the flow of information is presented. The approach focuses on usability, adaptability, performance, and reconfigurability. Similar to our approach, *DyKnow* extends existing models with new functionality. However, they only focus on models provided by ROS, whereas in our approach no such constrains exist. *DyKnow* allows for the same distribution mechanism as ROS provide.

The approach by Moutinho and Gomes [29] models distributed systems, including the communication in between, based on Petri nets extended with signal processing. This allows physically distributed Petri nets to communicate with each other based on channels and make reactive decisions. Similar to our work, they focus on one modelling technique, but in contrast stay within this technique, whereas our approach connects to others.

Summary. There are approaches tackling some of the three challenges, but not all at once. Certainly, existing technologies like EMF [33] or KMF [12] could be extended, but would lack features intrinsic to RAGs, like incremental evaluation.

8 CONCLUSION

Constructing cyber-physical systems (CPS) is a difficult endeavour. Different domain experts coming from different fields of research are needed to all necessary knowledge. They tend to use different terminology, different tools and different models. Furthermore, CPS have intrinsic difficulties such as participating systems being distributed, and the requirement to react fast during runtime.

In this paper, we want to support this process of designing and running a CPS by providing a flexible and generic approach to connect models of possibly different paradigm. We identified three challenges – distribution, multiple paradigms and fast, reactive behaviour – and four goals – minimize development effort, network usage, computation effort while maximize interoperability – and addressed each of them. Our approach uses models based on relational Reference Attribute Grammars [28] and offers a way to explicitly specify connections to other external models. A connection comprises an endpoint as the part to update (or used for updates) and a number of needed mappings to either parse, transform or serialize updated values. We showed the feasibility of our approach with a use case involving a robotic arm tasked to move to certain position while respecting safety zones where it is supposed to act slowly. There, two separate models controlling the application logic (where to move next) and the safety logic (when to move slowly) are connected to the internal robotic models controlling the robotic arm. We showed, that defining a connection between the models and the robotic arm can be done with minimal effort and resulting in minimized computation and network usage.

As already mentioned in section 6.3, different lines of future work are possible. The approach itself can be extended by exploring more design options like synchronizing complete subtrees in the model. Furthermore, new case studies can be evaluated involving models not implemented by ourselves as long as the exchange format of the values to be updated is known.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is partly supported by the German Research Foundation (DFG) as part of Germany's Excellence Strategy – EXC 2050/1 – Project 390696704 – "Centre for Tactile Internet with Human-in-the-Loop" (CeTI), the project "RISCOS" (project number 280611750) and the project "HybridPPS" (project number 418727532), and using tax money based on the budget approved by the Saxon state parliament for the project "PROSPER".

REFERENCES

[1] Brenna D Argall and Aude G Billard. 2010. A survey of tactile human–robot interactions. *Robotics and autonomous systems* 58, 10 (2010), 1159–1176.
[2] The ZeroMQ authors. 2020. ZeroMQ – An open-source universal messaging library. https://zeromq.org/. Accessed: 2020-07-20.
[3] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. 2015. Viatra 3: A Reactive Model Transformation Platform. In *Theory and Practice of Model Transformations*, Dimitris Kolovos and Manuel Wimmer (Eds.). Springer International Publishing, 101–110.
[4] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (June 2012). https://doi.org/10.1145/2187671.2187677
[5] Daniel de Leng. 2019. *Robust Stream Reasoning Under Uncertainty.* PhD Thesis. Linköping University Electronic Press.
[6] Daniel de Leng and Fredrik Heintz. 2016. DyKnow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system. In *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR).* https://doi.org/10.1109/SIMPAR.2016.7862375
[7] István Dávid, István Ráth, and Dániel Varró. 2016. Foundations for Streaming Model Transformations by Complex Event Processing. *Software & Systems Modeling* (May 2016). https://doi.org/10.1007/s10270-016-0533-1
[8] Thomas Huining Feng, Edward A. Lee, and Lee W. Shruben. 2010. Ptera: an event-oriented model of computation for heterogeneous systems. In *Proceedings of the tenth ACM international conference on Embedded software (EMSOFT '10).* Association for Computing Machinery, 219–228. https://doi.org/10.1145/1879021.1879050
[9] Open Source Robotics Foundation. 2012. XML Robot Description Format (URDF). http://wiki.ros.org/urdf/XML/model. Accessed: 2020-07-20.
[10] Open Source Robotics Foundation. 2019. Semantic Robot Description Format (SRDF). https://wiki.ros.org/srdf. Accessed: 2020-07-20.
[11] Open Source Robotics Foundation. 2020. MoveIt. https://moveit.ros.org/. Accessed: 2020-07-20.
[12] F Francois, G Nain, B Morin, E Daubert, O Barais, N Plouzeau, and J-M Jézéquel. 2014. Kevoree Modeling Framework (KMF): Efficient modeling techniques for runtime use. http://arxiv.org/abs/1405.6817
[13] L. Gherardi and N. Hochgeschwender. 2015. RRA: Models and tools for robotics run-time adaptation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).* 1777–1784.
[14] Antoon Goderis, Christopher Brooks, Ilkay Altintas, Edward A. Lee, and Carole Goble. 2009. Heterogeneous composition of models of computation. *Future Generation Computer Systems* 25, 5 (2009), 552–560. Publisher: Elsevier.
[15] Google. 2020. Protocol Buffers. https://developers.google.com/protocol-buffers. Accessed: 2020-07-20.
[16] Sami Haddadin, Michael Suppa, Stefan Fuchs, Tim Bodenmüller, Alin Albu-Schäffer, and Gerd Hirzinger. 2011. Towards the robotic co-worker. In *Robotics Research.* Springer, 261–282.
[17] Cécile Hardebolle and Frédéric Boulanger. 2008. ModHel'X: A Component-Oriented Approach to Multi-Formalism Modeling. In *Models in Software Engineering (Lecture Notes in Computer Science)*, Holger Giese (Ed.). Springer, Berlin, Heidelberg, 247–258. https://doi.org/10.1007/978-3-540-69073-3_26
[18] Cécile Hardebolle and Frédéric Boulanger. 2009. Exploring Multi-Paradigm Modeling Techniques. *SIMULATION* 85, 11-12 (Nov. 2009). https://doi.org/10.1177/0037549709105240
[19] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.
[20] Görel Hedin and Eva Magnusson. 2003. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0
[21] Juan Francisco Inglés Romero, Cristina Vicente Chicote, Brice Morin, Olivier Barais, et al. 2010. Using models@ runtime for designing adaptive robotics software: An experience report. (2010).
[22] Jackie Kay. 2016. Proposal for Implementation of Real-time Systems in ROS 2.
[23] J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50. https://doi.org/10.1109/MC.2003.1160055
[24] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.
[25] N. Koenig and A. Howard. 2004. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Vol. 3. 2149–2154 vol.3.
[26] Florence Maraninchi and Tayeb Bouhadiba. 2007. 42: programmable models of computation for a component-based approach to heterogeneous embedded systems. In *Proceedings of the 6th international conference on Generative programming and component engineering (GPCE '07).* Association for Computing Machinery, Salzburg, Austria, 53–62. https://doi.org/10.1145/1289971.1289981
[27] Johannes Meier, Christopher Werner, Heiko Klare, Christian Tunjic, Uwe Aßmann, Colin Atkinson, Erik Burger, Ralf Reussner, and Andreas Winter. 2020. Classifying Approaches for Constructing Single Underlying Models. In *Model-Driven Engineering and Software Development (Communications in Computer and Information Science)*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić (Eds.). Springer International Publishing, Cham, 350–375. https://doi.org/10.1007/978-3-030-37873-8_15
[28] Johannes Mey, René Schöne, Görel Hedin, Emma Söderberg, Thomas Kühn, Niklas Fors, Jesper Öqvist, and Uwe Aßmann. 2020. Relational Reference Attribute Grammars: Improving Continuous Model Validation. *Journal of Computer Languages* (Jan. 2020). https://doi.org/10.1016/j.cola.2019.100940
[29] Filipe Moutinho and Luis Gomes. 2014. Asynchronous-channels within petri net-based GALS distributed embedded systems modeling. *IEEE Transactions on Industrial Informatics* 10, 4 (2014), 2024–2033. https://doi.org/10.1109/TII.2014.2341933
[30] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. 5.
[31] René Schöne, Johannes Mey, Boqi Ren, and Uwe Aßmann. 2019. Bridging the Gap between Smart Home Platforms and Machine Learning using Relational Reference Attribute Grammars. In *Proceedings of the 14th International Workshop on Models@run.time.* Munich, 533–542. https://doi.org/10.1109/MODELS-C.2019.00083
[32] OASIS Standard. 2019. MQTT. https://mqtt.org. Accessed: 2020-07-20.
[33] D Steinberg, F Budinsky, E Merks, and M Paternostro. 2008. *EMF: Eclipse Modeling Framework* (2 ed.). Addison-Wesley Professional.
[34] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. 2012. The open motion planning library. *IEEE Robotics & Automation Magazine* 19, 4 (2012), 72–82. https://doi.org/10.1109/MRA.2012.2205651
[35] Emma Söderberg. 2012. *Contributions to the Construction of Extensible Semantic Editors.* PhD Thesis. Lund University. http://lup.lub.lu.se/record/3242518
[36] Hans Vangheluwe and Juan de Lara. 2003. Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling: meta-modelling and graph transformation. In *Proceedings of the 35th conference on Winter simulation: driving innovation (WSC '03).* Winter Simulation Conference, New Orleans, Louisiana, 595–603.
[37] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. 1989. Higher Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, USA) *(PLDI '89).* ACM, New York, NY, USA, 131–145. https://doi.org/10.1145/73141.74830
[38] Chris Wanstrath et al. 2020. Mustache - Logic-less templates. https://mustache.github.io. Accessed: 2020-07-20.
[39] Christopher Zhong and Scott A. DeLoach. 2011. Runtime Models for Automatic Reorganization of Multi-Robot Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '11).* 20–29. https://doi.org/10.1145/1988008.1988012
[40] Pouyan Ziafati, Mehdi Dastani, John-Jules Meyer, and Leon van der Torre. 2013. *Event-processing in Autonomous Robot Programming.* International Foundation for Autonomous Agents and Multiagent Systems.