Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Rafael Paiva Moura

Encoding and Analysis of Variational ROS Computation Graphs

February 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Rafael Paiva Moura

Encoding and Analysis of Variational ROS Computation Graphs

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
Professor Manuel Alcino Cunha

February 2022

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

In robotic applications, it is common to develop several variants of the same system (also known as a *software product line*), for example, to support different configurations of a robot. ROS is the most popular framework for developing robotic applications, where each application is implemented as a distributed system of computation nodes that communicate through message passing. HAROS is a framework for static analysis of ROS-based code. It can extract an abstract model of a ROS system's architecture (called the *computation graph*) and perform an analysis on that model. However, it can only analyse one configuration at a time.

In this thesis, we present three different approaches for encoding various ROS computation graphs in a single variational data structure, which contains the information related to the whole system and not just a configuration. Additionally, we also define a variational execution algorithm for each approach, along with a small query language, so that we can query and perform some analysis on said data structures. Lastly, we evaluate these algorithms and data structures so that we can reach some conclusions on which approaches work best, and in what conditions.

KEYWORDS Variability, Variational Software, Variational Data Structures, Variational Query Languages, Robotics, Software Product Lines

RESUMO

Nas aplicações robóticas, é comum desenvolver diversas variantes do mesmo sistema (também conhecido como uma *software product line*) para, por exemplo, suportar diferentes configurações de um robot. O ROS é a *framework* mais popular no que toca ao desenvolvimento de aplicações robóticas, onde cada aplicação é implementada como um sistema distribuído de nós de computação que comunicam entre si através do envio de mensagens. O HAROS é uma *framework* de análise estática de código ROS. Consegue extrair um modelo abstrato de uma arquitetura de um sistema ROS (chamado *grafo de computação*) e executar nesse modelo uma análise.

Nesta tese, apresentamos três diferentes abordagens para codificar vários grafos de computação ROS numa única estrutura de dados variacional, que contém a informação relativa a todo o sistema e não apenas a uma configuração. Adicionalmente, também definimos um algoritmo de execução variacional para cada abordagem, juntamente com uma pequena linguagem de *query*, de forma a que possamos analisar e pesquisar nessas estruturas de dados. Por fim, avaliámos estes algoritmos e estruturas de dados de modo a que possamos chegar a algumas conclusões sobre que abordagens funcionam melhor, e em que situações.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

Nowadays, robots are everywhere, as they are "intelligent" machines ultimately designed to help and assist humans in their day-to-day lives. The design, construction and development of these robots are embedded in a discipline called robotics.

Creating truly robust robot software is hard because there are many things to take account for and that vary constantly. For this reason, the *Robot Operating System* (ROS) framework was created. ROS is the most popular framework for developing robotic applications, providing a collection of tools, libraries and conventions that aim to simplify this task. In ROS, a robotic application is implemented as a distributed system of computation nodes that communicate through message passing following the publish-subscribe paradigm.

It is common to develop several variants of the same ROS system, for example, to support different configurations of a robot. In ROS, these different configurations are typically managed ad-hoc through different launch files. These files include the configuration of the system that will be executed, as well as which computation nodes exist.

One of robotics' major concerns is safety, as some robots are being used in safety-critical contexts. Currently, this safety relies mainly on software, in particular when robots operate in unstructured environments requiring a flexibility not possible with hardware based security. As a way to ensure it, the use of formal methods is essential, as it allows for an appropriate mathematical analysis to be made, contributing to the overall reliability of the system's design.

High-assurance ROS (HAROS) is a framework for static analysis of ROS-based code. Static analysis consists in extracting information from the source code without executing it. This allows an early detection of problems in the software development life cycle, which would go unnoticed in later stages of the development. HAROS can extract an abstract model of a ROS system's architecture (called computation graph) from a launch file and from the source code of the computation nodes referred to in that launch file. This ROS computation graph can have architectural restrictions imposed by custom queries defined by the user through one of HAROS plugins.

Currently, this HAROS plugin for architectural queries only analyses one configuration at a time. However, a robot can comprise hundreds of different configurations, making this approach unfeasible in practice. The overall goal of this thesis is precisely to improve this HAROS plugin, so that it can analyse multiple configurations at the same time. With this goal in mind, we will first define different possible techniques for encoding the various ROS computation graphs in a single variational data structure. Then, we will define a minimal query language and the respective variational execution algorithms, so that we can run analyses on such data structures, thus laying the foundations for a new implementation of the HAROS query plugin that will be able to efficiently analyse all configurations of a robot at once.

This thesis is organized as follows.

- In Chapter 2, we create a running example of a robotic application, that will be used throughout the entire thesis. We also implement this running example in ROS, showcasing the different features of the framework, and we use HAROS to perform an analysis on the implemented application.
- In Chapter 3, we present the concepts of *software product lines* and *variability*. We discuss their origin and importance, and give motivation for the need to perform analysis on them and with them.
- In Chapter 4 we present three different approaches for modeling variational data in a single data structure.
- In Chapter 5 we present the query language created for specifying properties, and also define the variational algorithms for executing these queries in variational data structures.
- In Chapter 6 we discuss how these data structures and algorithms where implemented in Python.
- In Chapter 7 we define a couple of techniques for generating synthetic variational databases, so that we can then perform an evaluation of the time and space efficiency of the different algorithms.
- Finally, in Chapter 8, we conclude the work, making a brief summary of the results obtained and presenting ideas for future work.

2

DEVELOPMENT AND ANALYSIS OF A ROS APPLICATION

In this chapter, we briefly describe how to develop a simple robotic application with the *Robot Operating System* (ROS) (Quigley, 2009), nowadays the most popular middleware for developing robotic software. We also show how to perform some static analyses of the developed code with HAROS, a plugin based tool for evaluating the quality of ROS software (Santos et al., 2016, 2019). Note that this description is only focused on the functionalities that are relevant to the theme of this thesis. For a more detailed description about these tools the reader should check the above references.

2.1 RUNNING EXAMPLE

The running example to be implemented in this chapter is the **Controlled Turtlesim (CT)** system, a simulation that contains a turtlebot[1] whose movement can be controlled both manually and autonomously. The manual control is made by the user through the keyboard, while the autonomous one is made by the turtlebot, randomly. If both types of movement are enabled, the manual movement has always priority over the autonomous one, the latter only being activated if the former has been inactive for a certain pre-defined time, and deactivated whenever manual instructions are being received. Furthermore, the simulation's space is limited, meaning it has borders. In this system, the turtlebot can also be aware of its surroundings, having a safety protocol that prevents it from hitting the border. Also, when moving, the turtle leaves a colored trail of the movement. When the safety protocol is activated, the turtle's pen color is changed.

As we can see, we do not want to implement a single application, but rather a family of applications, each with its own configuration. The CT system has three possible features, that can be represented by the following set:

$$F = \{\mathit{CM}, \mathit{RM}, \mathit{SC}\}$$

where

- ***CM*, Controlled Movement** - The turtlebot movement is manually controlled by the user using the keyboard.
- ***RM*, Random Movement** - The turtlebot moves autonomously and randomly.

[1] https://www.turtlebot.com

- ***SC*, Safety Controller** - Prevents the turtlebot from hitting walls, and changes its pen color.

Since all features are optional, the set of all possible configurations for the CT system is represented by the powerset of F:

$$C = \mathcal{P}(F) = \{\emptyset, \{CM\}, \{RM\}, \{SC\}, \{CM, RM\}, \{CM, SC\}, \{RM, SC\}, \{CM, RM, SC\}\}$$

However, there are two configurations that do not make much sense: the one with no features, because the turtle is going to do nothing; and the one with only the *SC* feature enabled, because if the turtle can not move, the safety protocol will never be activated.

So, that leaves a total of six configurations:

$$C = \{\{CM\}, \{RM\}, \{CM, RM\}, \{CM, SC\}, \{RM, SC\}, \{CM, RM, SC\}\} \tag{1}$$

which can alternatively be represented by Table 1.

	CM	*RM*	*SC*
Config. 1	✓		
Config. 2		✓	
Config. 3	✓	✓	
Config. 4	✓		✓
Config. 5		✓	✓
Config. 6	✓	✓	✓

Table 1: CT system configurations table.

2.2 ROBOT OPERATING SYSTEM

The *Robot Operating System*, also known as *ROS*, is a framework for writing robotic software. Introduced by Quigley (2009), it is designed with *open-source* in mind, allowing its users to choose and use their own configuration of standalone tools and libraries to interact with the core of the framework. Being standalone, these libraries should hold no dependencies on ROS, and are encouraged to contain all the complexity of things like algorithms or drivers. ROS can then be used to create small executables that expose library functionality. This allows for not only easier code extraction and unit testing, but also for *code reuse*, which was one of the main motivations for the development of the framework. Lastly, ROS also aims to be *multi-lingual*, and indeed it supports various languages.

2.2.1 *Computation Graph*

A ROS system is a set of processes, called *nodes*, connected in a peer-to-peer topology, called the *computation graph*. The nodes communicate with each other through *message* passing, following the publish-subscribe[2] paradigm, i.e. a node can send a message by publishing it to a *topic*, and the nodes subscribed to that topic will receive that message. A topic is just a name used to identify the content of a message.

The use of graphs makes for an easier and much more intuitive understanding of a system. For example, Figure 1 depicts a computation graph of a system in which some node A communicates via topic t with some other node B.

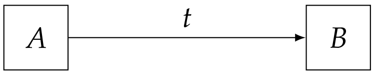


Figure 1: Example of a computation graph.

The graph in Figure 1 can also be represented with other notation, as it is shown in Figure 2. Depending on the case, we may use one notation or the other, for the sake of simplicity.

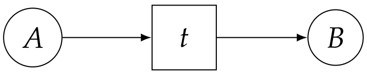


Figure 2: Figure 1's computation graph with a different notation.

Each configuration of a system has its own computation graph. Previously, in our running example, we represented each configuration as a set of features (see Table 1). However, we should represent it as a set of nodes instead, as the *node* is the core mechanism to implement features in a ROS system.

NODES Nodes are processes that perform computation, and each one is responsible for some kind of function in the system. For instance, in the CT example, we can have a node to handle the keyboard input for the *CM* feature, a node to implement the *RM* feature, another to run the simulator, and so on. This idea of isolating functionalities makes the system somewhat *fault-tolerant*, because crashes are local. It also reduces *code complexity*, because it hides it inside each node, which in turn provides only a minimal API for the rest of the graph to interact with.

So, each set of features has to map to a set of nodes, or more abstractly, a multiset of nodes, because we can have more than one node of the same type. At first, it may seem that we only need to define four nodes:

[2] https://en.wikipedia.org/wiki/Publish-subscribe_pattern

one to run the simulator (S), and three for each feature (CM, RM and SC), resulting in the following set of all configurations of nodes:

$$\begin{aligned} C_N = \{&\{CM \mapsto 1, S \mapsto 1\},\\ &\{RM \mapsto 1, S \mapsto 1\},\\ &\{CM \mapsto 1, RM \mapsto 1, S \mapsto 1\},\\ &\{CM \mapsto 1, SC \mapsto 1, S \mapsto 1\},\\ &\{RM \mapsto 1, SC \mapsto 1, S \mapsto 1\},\\ &\{CM \mapsto 1, RM \mapsto 1, SC \mapsto 1, S \mapsto 1\}\} \end{aligned}$$

However, as the reader may recall from the system's description, the features are organized in a priority-based hierarchy. The *CM* feature has priority over the *RM* feature, and the *SC* feature has priority over both of them. As such, we need a fifth type of node, called *multiplexer* or MP, that handles the priority between two features. This node subscribes two topics, one that represents *high priority* and other that represents *low priority*. It has a timer that, while active, makes the node forward only the messages from the high priority topic. When the timer finishes, it starts forwarding also the low priority messages, and it resets every time it receives an high priority message. The duration of the timer is given as a parameter.

Figure 3 depicts the interface of the MP node, where T denotes the timer duration parameter.

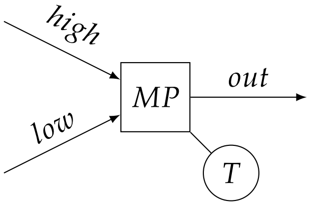


Figure 3: Graph representation of the *multiplexer*.

With this, we can easily implement the priority between two nodes, like the CM and the RM nodes, for example. All we have to do is to make sure that the high priority topic is the advertised topic of the CM node, and the low priority topic is the one from the RM node. To handle the priority between more than two nodes, we can use multiple multiplexers, by "cascading" them, i.e, one multiplexer handles two nodes, and its output will be subscribed by other multiplexer, and so on. In fact, we can handle the priority between n nodes by using $n - 1$ multiplexers.

```
Vector3 linear
Vector3 angular
```

Listing (2.1) *Twist* message.

```
float64 x
float64 y
float64 z
```

Listing (2.2) *Vector3* message.

With this, the set of all configurations of nodes becomes:

$$
\begin{aligned}
C_N = \{&\{CM \mapsto 1, S \mapsto 1\},\\
&\{RM \mapsto 1, S \mapsto 1\},\\
&\{CM \mapsto 1, RM \mapsto 1, MP \mapsto 1, S \mapsto 1\},\\
&\{CM \mapsto 1, SC \mapsto 1, MP \mapsto 1, S \mapsto 1\},\\
&\{RM \mapsto 1, SC \mapsto 1, MP \mapsto 1, S \mapsto 1\},\\
&\{CM \mapsto 1, RM \mapsto 1, SC \mapsto 1, MP \mapsto 2, S \mapsto 1\}\}
\end{aligned}
$$

and it can also be represented by a table. Table 2 shows the configuration table of features and the respective configuration table of nodes.

	Features			**Nodes**				
	CM	*RM*	*SC*	*CM*	*RM*	*SC*	*MP*	*S*
Config. 1	✓			1				1
Config. 2		✓			1			1
Config. 3	✓	✓		1	1		1	1
Config. 4	✓		✓	1		1	1	1
Config. 5		✓	✓		1	1	1	1
Config. 6	✓	✓	✓	1	1	1	2	1

Table 2: Configurations' table.

MESSAGES, TOPICS AND SERVICES As aforementioned, nodes communicate with each other through message passing. A message is a simple typed data structure that supports several primitive types, arrays of primitive types, other messages, and arrays of other messages. They are defined in files with the "*.msg*" extension.

One of the messages used in our running example is the *Twist* message, which expresses velocity in free space broken into its linear and angular parts. Listing 2.1 shows the specification for this type of message.

Generally, in a publish-subscribe paradigm, nodes are not aware of who they are communicating with. A node sends a message by publishing it to a topic, and receives messages by subscribing to topics. A topic is just a name, that must be unique, and each one can have multiple subscribers and publishers. A topic is also typed by the message that was used to publish on it.

This type of communication is unidirectional, which does not allow for request/reply communication, although these synchronous transactions are often required in a distributed system. As a way to support this, ROS has *services*. A service is defined by a string name, like a topic, and a pair of message types: one for the request, and the other for the reply.

With all this components defined, we can now create the computation graph for each configuration of our system. Figure 5 shows an abstract computation graph of the maximal *CT* system, i.e. Config. 6. In there, we can see the six different nodes of the configuration, along with some topics and a service. The vel topic, is an abstract representation of a topic with the *Twist* type. As it was mentioned before, each topic should (and must) be unique, so this identifier is not the topic name but an indication of its type. The $pose$ topic is used for the safety node to know the position of the turtle on the map. And lastly, the set_pen service is responsible for changing the color of the turtle's pen.

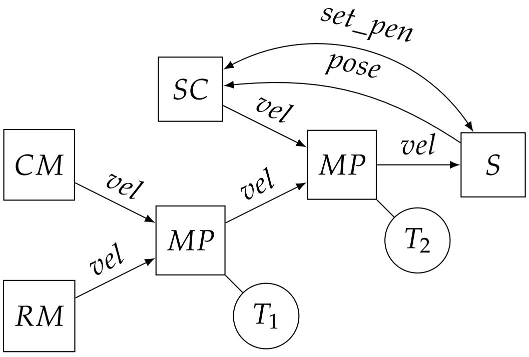


Figure 5: CT system's computation graph.

ROS MASTER The publish-subscribe paradigm normally has a message broker that serves has an intermediary in the communication between nodes. Its function consists in receiving messages with a given topic from publishers and routing them to the subscribers of that respective topic. But this is a peer-to-peer system, or in other words, a "brokerless" system. How does ROS establish the connection between the nodes then?

The ROS *Master* is a node that is present in all ROS systems, and is responsible for giving the nodes of the system the capability of locating each other, through naming and registration services. As an example, we will now present a possible sequence of events that could establish the computation graph depicted in Figure 1.

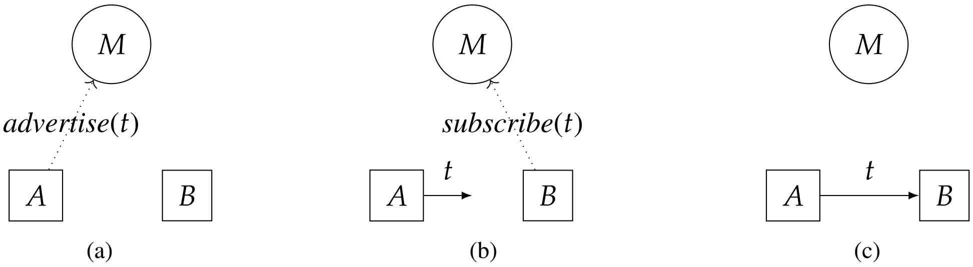


Figure 6: Example of the establishment of a connection between two nodes.

When a node wants to publish messages in a topic, it first has to notify the *Master* node. This is called *advertising* the topic and it is what is happening in figure 6a. From the moment that the *Master* node is notified, the publisher node can start publishing the messages in the respective topic. If no nodes have yet subscribed to that topic, the publisher does not send any data. To subscribe to a topic, a node also needs to notify the *Master* node. In figure 6b, node A is already publishing, and node B is notifying the *Master* node that it wants to subscribe to the topic. Finally when a topic has both a publisher and a subscriber, the *Master* node notifies both nodes about each other's existence and they can start communicating peer-to-peer (figure 6c).

Implementation of a node

To create a node, we need to specify it in a program, and then use the ROS framework to execute it. ROS provides client libraries that can be used to interact with the framework. In our case, we use `roscpp`, which is the library for the C++ language.

Let us consider the SC node of our CT system, for example. If we recall the maximal system's computation graph (Figure 5) we see that this node: publishes in one topic (vel), subscribes one topic ($pose$), and has one service (set_pen). Listing 2.3 shows our class declaration and constructor for this node.

First we declare and use some objects that are essential in a ROS application, namely:

- The `NodeHandle` object (line 11) is an handle to process the node. It gives us access to the `advertise` and `subscribe` functions, that will be used to communicate with the *ROS Master* and, when first created, it also initializes the node.

- In lines 12 and 13, we declare the node's `Subscriber` and `Publisher` objects. These, as the names say, are used to subscribe and publish to topics.

- Lastly, at line 14, we declare the `ServiceClient` object, that is responsible for interacting with services.

In the class constructor (lines 29-54), we initialize and setup the declared variables.

- The `Subscriber` is initialized with the `subscribe` function (lines 23-28). This function notifies the Master about the topic the node wants to subscribe, specifying the type of message it hopes to receive, and the callback function that will be executed when a message is received. Here, we subscribe to topic `cmd_vel`, expecting to receive messages of type `Twist`, and with the callback that is defined in lines 24-38.

- The `Publisher` is initialized with the `advertise` function (line 40). This function makes the advertising of a topic to the Master, specifying the type of message that will be published. Here, we want to publish in topic `cmd_vel` with messages of type `Twist`.

- The `ServiceClient` is initialized with the `serviceClient` function (line 42), that tells the Master which service the node is going to use, and what type of message will be passed.

```
class SafetyController {

  public:
    SafetyController(void);
    void loop(void);

  private:
    bool safety_activated = false;
    turtlesim::Pose turtle_pose;

    ros::NodeHandle nh;
    ros::Subscriber pose_sub;
    ros::Publisher cmd_vel_pub;
    ros::ServiceClient pen_client;

    bool turtleInCenter(turtlesim::Pose pose);
    geometry_msgs::Twist calculateTwistMessage(turtlesim::Pose pose);
    void setPenColor(int r, int g, int b);
};

SafetyController::SafetyController(void) {

  pose_sub = nh.subscribe<turtlesim::Pose>("pose", 10,
    [this](const turtlesim::Pose::ConstPtr &msg) {
      float x = msg->x;
      float y = msg->y;
      if (x < 1.0 or x > 10.0 or y < 1.0 or y > 10.0) {
        ROS_INFO("SAFETY INITATED");
        safety_activated = true;
        setPenColor(255, 0, 0);
      }

      turtle_pose.x = x;
      turtle_pose.y = y;
      turtle_pose.theta = msg->theta;
      turtle_pose.linear_velocity = msg->linear_velocity;
      turtle_pose.angular_velocity = msg->angular_velocity;
    });

  cmd_vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel", 10);

  pen_client = nh.serviceClient<turtlesim::SetPen>("turtle1/set_pen");
}
```

Listing 2.3: Class definition and contructor of the *SC* node

After defining the class and the constructor, we now define a loop function so that the node is always active and ready for communication. And then, we are ready to create the main function. Listing 2.4 shows the implementation of these 2 functions.

In the loop function (lines 1-23), we start by declaring a `ros::Rate` object, which is used to define the rate, in Hz, at which the loop will run. In this case, we want the rate to be 10Hz. Then, we define the main loop. Once `ros::ok()` returns false, all ROS calls will stop. `ros::ok()` returns false if: (a) a SIGINT is received; (b)

```
 1 void SafetyController::loop(void) {
 2 
 3   ros::Rate rate(10);
 4   while (ros::ok()) {
 5 
 6     if (not safety_activated) {
 7       setPenColor(255,255,255);
 8     }
 9 
10     bool in_center = turtleInCenter(turtle_pose);
11     if (safety_activated and not in_center) {
12       geometry_msgs::Twist msg = calculateTwistMessage(turtle_pose);
13       cmd_vel_pub.publish(msg);
14     }
15 
16     if (in_center) {
17       safety_activated = false;
18     }
19 
20     ros::spinOnce();
21     rate.sleep();
22   }
23 }
24 
25 int main(int argc, char *argv[]) {
26 
27   ros::init(argc, argv, "safety_controller");
28 
29   SafetyController controller;
30   controller.loop();
31 
32   return 0;
33 }
```

Listing 2.4: Loop and main functions of the *SC* node.

another node with the same name kicks this node out of the network; (c) `ros::shutdown()` is called by another part of the application; (d) all `ros::NodeHandle`s have been destroyed.

In the main loop, various things happen. More specifically, in lines 12 and 13, we use the `publish()` function to publish a message to the topic. At the end of the loop we call the `ros::spinOnce()` function, so that we can keep calling callbacks, and the `Rate::sleep()` function, that sleeps for the correct amount of time, according to the rate that we defined previously.

Finally, in the `main` function (lines 25-33), we start by calling the `ros::init` function. This function initializes ROS. It can receive command line arguments and it is also responsible for naming the node. Node names must be unique in the system. In this case, the node has the name `safety_controller`.

Next, we declare an `SafetyController` object, that, as it was said before, represents our node. After that, we execute the object's loop function, starting the execution of the node.

```
<package format="2">
  <name>controlled_turtlesim</name>
  <version>0.0.0</version>
  <description>
    This package provides the capability of controlling the turtlesim
    randomly and manually.
  </description>
  <maintainer email="pedrorpmoura@gmail.com">Pedro Moura</maintainer>
  <licence>BSD</license>

  <depend>roscpp</depend>
  <depend>rospy</depend>
  <depend>std_msgs</depend>

</package>
```

Listing 2.5: Example of a package manifest with dependencies.

2.2.2 *Packages*

Software in ROS is organized in **packages**. A ROS package is a collection of files that constitute a module with a specific purpose. At its minimal state, it is just a directory containing an XML file, called the **manifest**, that describes the package and its dependencies. This open-ended nature of ROS packages allows for great variation in their structure and purpose. A ROS package might contain ROS nodes, a ROS-independent library, datasets, third-party software, configuration files, anything that makes sense in the constitution of a module.

The use of packages makes it possible to divide ROS-based software into small chunks that are easily manageable, and that can be maintained and developed concurrently by its own developers.

PACKAGE MANIFEST As mentioned above, every package has a manifest. The manifest is a XML file called `package.xml`, and it gives information about the package, such as its name, version, authors, maintainers and dependencies.

For our *CT* system, we can create a package with the manifest in Listing 2.5.

LAUNCH FILES Initializing the nodes one by one can be a tedious process, especially if we are dealing with tens or hundreds of nodes. To avoid this, ROS provides a way to start the master and many nodes all at once, using **launch files**.

A launch file is an XML file used to specify a certain configuration of a ROS system, by listing a group of nodes that should be started at the same time. As shown in table 2, the CT system can have six different configurations. And each configuration has its own launch file. Listing 2.6 shows the launch file for Config. 6.

```
<launch>

  <node pkg="example_controlled_turtlesim" name="safety" type="safety_controller"
      >
    <remap from="pose" to="turtle1/pose"/>
  </node>

  <group ns="move">
    <node pkg="turtlesim" name="teleop" type="turtle_teleop_key"/>
    <node pkg="example_controlled_turtlesim" name="random" type="random"/>

    <node pkg="example_controlled_turtlesim" name="binary_multiplexer" type="
        binary_multiplexer_twist">
      <param name="time" type="int" value="3"/>
      <remap from="high" to="turtle1/cmd_vel"/>
      <remap from="low"  to="cmd_vel"/>
    </node>
  </group>

  <node pkg="example_controlled_turtlesim" name="binary_multiplexer" type="
      binary_multiplexer_twist">
    <param name="time" type="int" value="1"/>
    <remap from="high" to="cmd_vel"/>
    <remap from="low"  to="move/out"/>
  </node>

  <node pkg="turtlesim" name="sim" type="turtlesim_node">
    <remap from="turtle1/cmd_vel" to="out"/>
  </node>

</launch>
```

Listing 2.6: Launch file for the complete CT system

2.3 (H)IGH (A)SSURANCE ROS

The **High Assurance ROS**, or **HAROS**, is a framework for static analysis of ROS-based code. Static analysis is the analysis of computer software that is performed without actually executing programs. Usually, static analysis is performed in *source code*, but it can also be performed in *object code*.

One of the most important uses of static analysis is in the verification of software properties. This is essential in the robotic context because robots are being increasingly used in *safety-critical* systems[3], in which software reliability is a necessity. HAROS offers a broad range of static analysis techniques, such as code metrics extraction, property testing and even model extraction. This is possible because this framework is not self-contained in ROS, but instead makes use of third-party analysis tools, encapsulated as HAROS plugins. Being a plugin orientated tool, it allows for any regular ROS developer to perform a good static analysis, without having great knowledge about it.

2.3.1 *Project files*

In order to analyse a system, HAROS needs **project files**. A project file is a YAML file, and it functions more or less like ROS's package manifest, informing HAROS which packages and configurations are to be analysed.

To specify which packages are to be analysed, we use the `packages` key, that maps to a list of values, each one representing a package. After this, we inform HAROS which configurations are to be present in the analysis, by using the `configurations` key. Listing 2.7 shows a project file for our CT package.

2.3.2 *Visualizer*

To help with the presentation of the analysis results, HAROS provides a **visualizer**. This visualizer not only has the results of the analysis, but also has a graph diagram of the system's packages and their dependencies, and the computation graphs of the system's configurations.

When we run the analysis, a web page will be opened in the default web browser, where we can see the visualizer and its different views of the analysis result. There are four views: **Dashboard**, **Packages**, **Issues** e **Models**.

The Dashboard (Figure 7) gives an overall result of the analysis and the number of issues found.

In the Packages tab (Figure 8), we can see the package graph view, that shows the analysed packages and their dependencies.

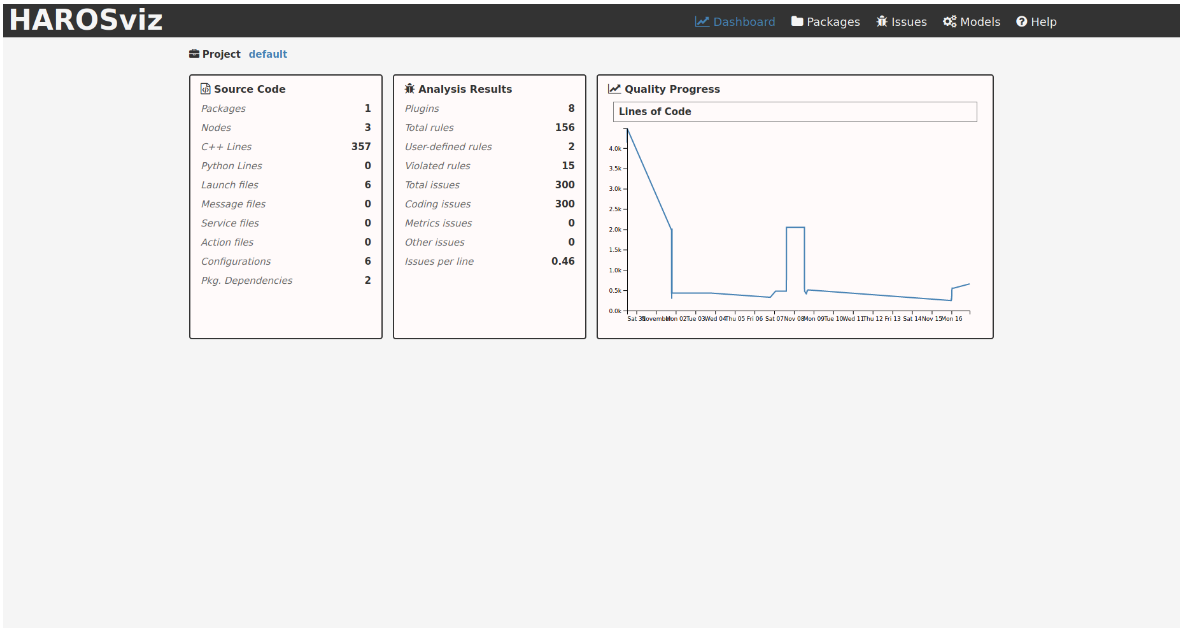
The Issues page (Figure 9) shows the list of issues, with all the rules that were violated.

Lastly, there is the Models tab (Figure 10), where we can see the different computation graphs of the system's configurations.

[3] `https://en.wikipedia.org/wiki/Safety-critical_system`

```
%YAML 1.1
---
packages:
- example_controlled_turtlesim

configurations:
    config1:
        launch: [example_controlled_turtlesim/launch/config1.launch]

    config2:
        launch: [example_controlled_turtlesim/launch/config2.launch]

    config3:
        launch: [example_controlled_turtlesim/launch/config3.launch]

    config4:
        launch: [example_controlled_turtlesim/launch/config4.launch]

    config5:
        launch: [example_controlled_turtlesim/launch/config5.launch]

    config6:
        launch: [example_controlled_turtlesim/launch/config6.launch]
```

Listing 2.7: HAROS project file example.



Project default

Source Code	
Packages	1
Nodes	3
C++ Lines	357
Python Lines	0
Launch files	6
Message files	0
Service files	0
Action files	0
Configurations	6
Pkg. Dependencies	2

Analysis Results	
Plugins	8
Total rules	156
User-defined rules	2
Violated rules	15
Total issues	300
Coding issues	300
Metrics issues	0
Other issues	0
Issues per line	0.46

Figure 7: HAROS Visualizer - Dashboard.

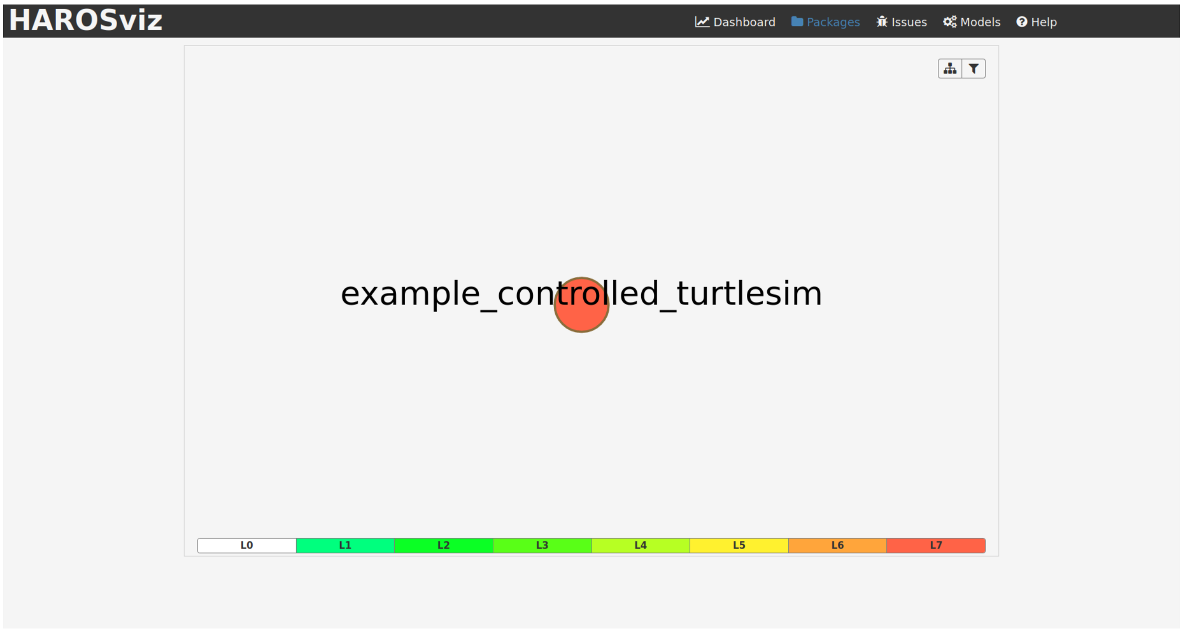
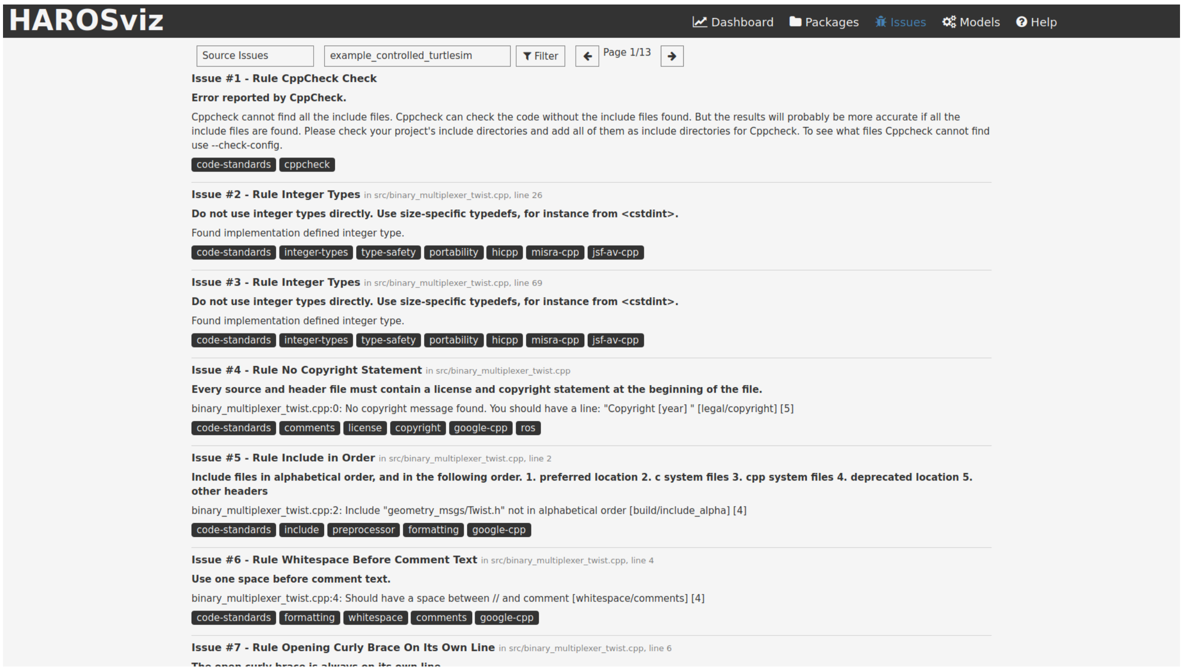


Figure 8: HAROS Visualizer - Packages.



Issue #1 - Rule CppCheck Check

Error reported by CppCheck.

Cppcheck cannot find all the include files. Cppcheck can check the code without the include files found. But the results will probably be more accurate if all the include files are found. Please check your project's include directories and add all of them as include directories for Cppcheck. To see what files Cppcheck cannot find use --check-config.

code-standards cppcheck

Issue #2 - Rule Integer Types in src/binary_multiplexer_twist.cpp, line 26

Do not use integer types directly. Use size-specific typedefs, for instance from <cstdint>.

Found implementation defined integer type.

code-standards integer-types type-safety portability hicpp misra-cpp jsf-av-cpp

Issue #3 - Rule Integer Types in src/binary_multiplexer_twist.cpp, line 69

Do not use integer types directly. Use size-specific typedefs, for instance from <cstdint>.

Found implementation defined integer type.

code-standards integer-types type-safety portability hicpp misra-cpp jsf-av-cpp

Issue #4 - Rule No Copyright Statement in src/binary_multiplexer_twist.cpp

Every source and header file must contain a license and copyright statement at the beginning of the file.

binary_multiplexer_twist.cpp:0: No copyright message found. You should have a line: "Copyright [year] " [legal/copyright] [5]

code-standards comments license copyright google-cpp ros

Issue #5 - Rule Include in Order in src/binary_multiplexer_twist.cpp, line 2

Include files in alphabetical order, and in the following order. 1. preferred location 2. c system files 3. cpp system files 4. deprecated location 5. other headers

binary_multiplexer_twist.cpp:2: Include "geometry_msgs/Twist.h" not in alphabetical order [build/include_alpha] [4]

code-standards include preprocessor formatting google-cpp

Issue #6 - Rule Whitespace Before Comment Text in src/binary_multiplexer_twist.cpp, line 4

Use one space before comment text.

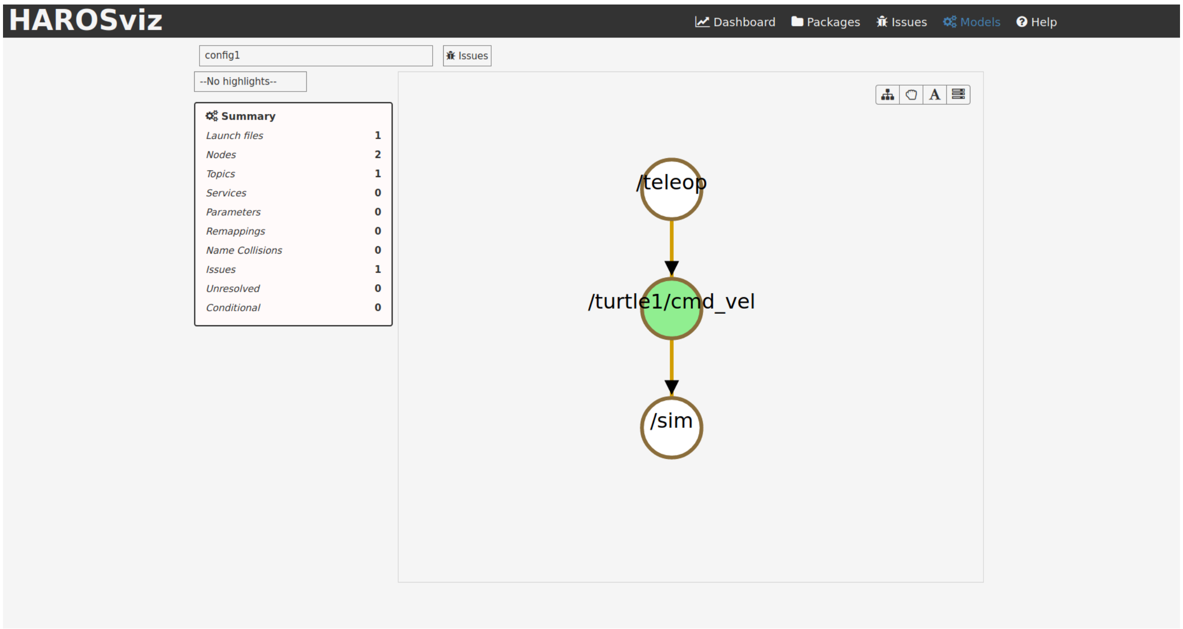
binary_multiplexer_twist.cpp:4: Should have a space between // and comment [whitespace/comments] [4]

code-standards formatting whitespace comments google-cpp

Issue #7 - Rule Opening Curly Brace On Its Own Line in src/binary_multiplexer_twist.cpp, line 6

The open curly brace is always on its own line

Figure 9: HAROS Visualizer - Issues.



Summary	
Launch files	**1**
Nodes	**2**
Topics	**1**
Services	**0**
Parameters	**0**
Remappings	**0**
Name Collisions	**0**
Issues	**1**
Unresolved	**0**
Conditional	**0**

Figure 10: HAROS Visualizer - Models.

2.3.3 *Query engine*

One of HAROS main features, and a focal point in this thesis, is the **query engine**. The query engine allows the specification of user-defined custom queries to run over the extracted data. This engine is integrated in HAROS, i.e. is not a plug-in, although it could be implemented like one.

In most cases, using a query language is simpler and more desirable than implementing an analysis plugin, but the range of properties that can be specified is more limited. Being implemented as a core feature also allows the graphical visualizer to provide graphical feedback of the executed queries.

The HAROS query language is based on `pyflwor`[4], which is a query language for querying python objects. To define queries, we can use the `rules` section in the project file, as it is shown in Listing 2.8.

[4] https://github.com/timtadh/pyflwor

```
...
rules:
  type_check_topics:
    name: Message Types Must Match
    description: All nodes using a topic must communicate using the
    same message type.
    tags:
      - type-check
      - ros-comm
      - custom-filter-tag
    scope: configuration
    query: "for p in <nodes/publishers | nodes/subscribers>,
                s in <nodes/publishers | nodes/subscribers>
            where p.topic_name == s.topic_name and p.type != s.type
            return p, s"
```

Listing 2.8: Example of a query in HAROS.

3

VARIABILITY MANAGEMENT IN SOFTWARE PRODUCT LINES

When developing large-scale software systems it is common to adopt the paradigm of *Feature-Oriented Software Development* (FOSD). As the name suggests, the key concept of FOSD is a feature. A *feature* is a unit of functionality that satisfies requirements, implements design decisions, and offers a configuration option. FOSD tries to decompose a software system in terms of the features it provides. If correctly implemented, it is possible to generate many variants of the system from a set of features. The set of the generated systems is called a *Software Product Line* (SPL). Since their introduction, SPLs have become a popular way of enhancing quality, supporting reuse, and taking advantage of variability to derive different product variants efficiently. *Variability* is the ability of a software artifact to vary its behavior at some point in its lifecycle (Svahnberg et al., 2005).

An example of a SPL is our running example from the previous chapter, since we had a robotic system that could have six different configurations, where each configuration could be represented by a set of features.

In fact, SPLs are very common in robotics, and managing the variability of these applications is one of the main current challenges in robotic development. Unlike in other system domains, robotic systems do not have a general system architecture, as robots can have different shapes, features, and behaviors, that also depend on the environment in which they find themselves in. This diversity leads to the existence of multiple types of variability, which come from different points in the robot's lifecycle. According to García et al. (2019) there are four sources of variability in a robotic application:

- **Customer Requirements** - Differences in the costumer requirements influence which hardware and software components are deployed to the robot. This is known as static variability, and it leads to having different variants of the same model. This is exactly the type of variability that happens in our example from the previous chapter.

- **Environment** - A big part of the robot behavior depends on its environment. This originates a variability of environmental conditions, such as the design of objects and possible obstacles. Although some of this variability can be solved at design time, most of it can only be solved at runtime. Runtime variability is still a challenge to be solved.

- **Robot Hardware** - A robot can have different types of hardware components, which leads to changes at the software level.

- **Middleware** - The same middleware can have different versions, with different requirements. Some companies might want to use the same robotic application on different versions of the middleware (for example, different versions of ROS).

Most techniques for implementing SPLs usually fall in one of two categories: *annotative* and *compositional*. In annotative approaches, features are implemented by annotating the source code, either explicitly or implicitly. A common example of an annotative approach is the use of the `#ifdef` and `#endif` directives of the C preprocessor to surround feature code, as shown in Listing 3.1.

```
int foo(int a #ifdef B, int b #endif) {
  int c = a;
  if (c) {
    c += a;
    #ifdef B c += b; #endif
  }
  return c;
}
```

Listing 3.1: Example of the use of C preprocessor directives to implement features.

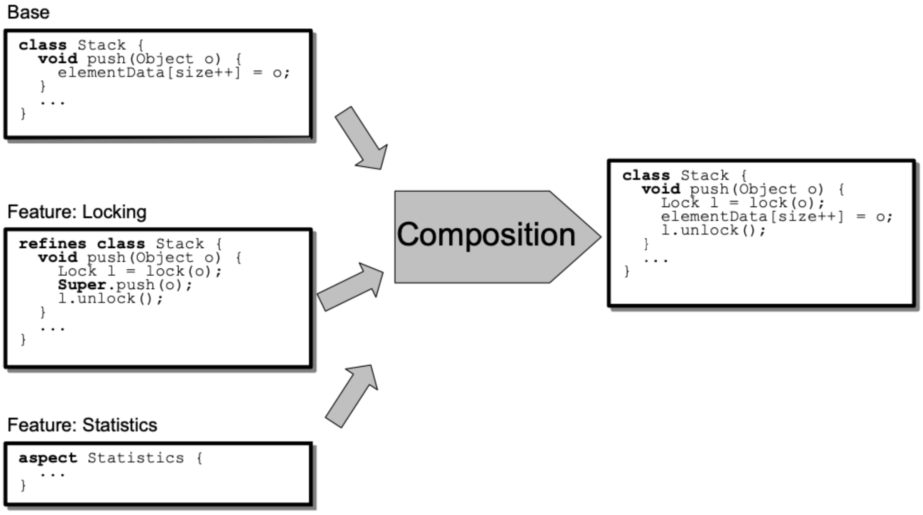
However, such annotation style "taints" the code with a large amount of boilerplate code, making it hard to understand and maintain. This is commonly referred to as the `#ifdef` *hell*[1]. A common tactic to avoid the use of `#ifdef` statements is to use colors to annotate the code. *CIDE* (Kästner et al., 2008) is a tool that implements this idea to annotate Java source code. This strategy has been implemented not only in programming languages, but also in modeling languages. An example of this is *Colorful Alloy* (Liu et al., 2019), which is a extension for the modeling language Alloy to support SPL design. Other examples of annotative approaches include *explicit programming* (Bryant et al., 2002), *software plans* (Coppit et al., 2007), *metaprogramming with traits* (Reppy and Turon, 2007), and *annotation-based aspects* (Kiczales and Mezini, 2005).

On the other hand, compositional approaches use a "separations of concerns" methodology when implementing variability. Each feature is implemented as a separated code unit and a software product line can be generated through feature composition. Figure 11 illustrates the general idea of feature composition. *AHEAD* (Batory et al., 2004) and *FeatureAlloy* (Apel et al., 2010) are examples of this type of approach for Java and Alloy, respectively. Feature-oriented programming (Prehofer, 2001) and aspect-oriented programming (Kiczales et al., 2001) are other examples that use an compositional approach to implement features.

3.1 VARIABILITY MODELING WITH FEATURE MODELS

A **Feature Model** is a representation of all possible products in a SPL in terms of features and the relationships among them. Its main purpose is to provide a structure to model and analyse the commonality and variability of a SPL. They were first introduced by Kang et al. in 1990 (Kang et al., 1990), as part of the *Feature-Oriented Domain Analysis* method.

[1] https://www.cqse.eu/en/news/blog/living-in-the-ifdef-hell/



```
class Stack {
  void push(Object o) {
    elementData[size++] = o;
  }
  ...
}
```

```
refines class Stack {
  void push(Object o) {
    Lock l = lock(o);
    Super.push(o);
    l.unlock();
  }
  ...
}
```

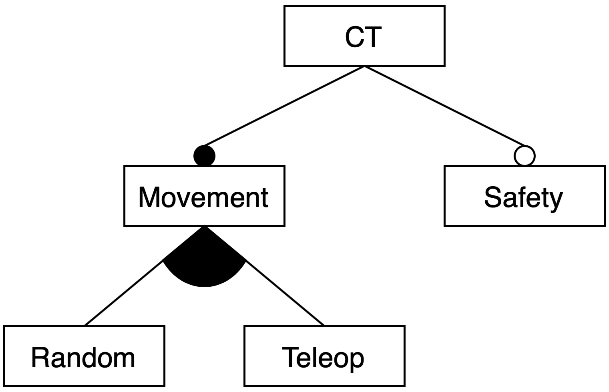
```
aspect Statistics {
  ...
}
```

```
class Stack {
  void push(Object o) {
    Lock l = lock(o);
    elementData[size++] = o;
    l.unlock();
  }
  ...
}
```

Figure 11: General idea of composition (Kästner and Apel, 2008).

Usually, in a feature model, the set of features is hierarchically arranged in a tree-like structure, in which the root node indicates the SPL that is described, the inner nodes represent the parent (compound) features, and the leaf nodes represent child features (subfeatures). A feature model can also have cross-tree constraints, that establish dependencies between features.

The most common way to represent a feature model is through a *feature diagram*, which is just a visual notation for feature models. Figure 12a shows the feature diagram of the CT software product line. According to it, all systems must possess at least one type of movement activated, and may or may not have the safety feature activated. However, there are other ways to specify feature models, such as languages (Classen et al., 2010; Deursen and Klint, 2002; Collet, 2014) or propositional formulas[2]. Figure 12b shows how to describe the same feature model in the TVL language (Classen et al., 2010).



(a) Feature diagram of the CT SPL.

```
root CT {
  group allOf {
    group someOf {
      Random,
      Teleop
    }
    opt Safety
  }
}
```

(b) TVL specification for the CT SPL.

[2] This is discussed in a later section

In both figures, we used the classical (or basic) notation of feature models (Kang et al., 1990). Since their introduction, various notation extensions were purposed, like cardinality-based feature models (Czarnecki et al., 2005) and extended feature models (Benavides et al., 2005a). However, these are not really relevant for this work, and shall not receive much focus. As for the classical notation, Table 3 shows the relationships between parent features and child features, and also the type of dependencies that exist between features.

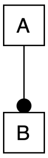
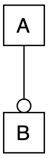
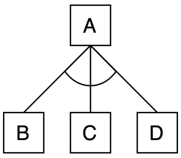
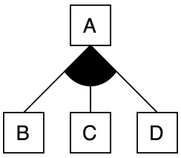
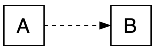
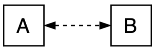
	Name	**Description**	**Diagram**
Relationships	*Mandatory*	A mandatory relationship means that a subfeature must be included if its parent is selected. In other words, the child feature is included in every product in which its parent feature appears.	
	Optional	An optional relationship means that a subfeature may be included in the product in which its parent feature appears.	
	Alternative	An alternative (or xor) relationship between a set of subfeatures and their parent means that exactly one subfeature must be included in the product in which its parent appears.	
	Or	An or relationship between a set of subfeatures and their parent means that at least one of the child features must be included in the product in which its parent appears.	
	Requires	Feature A requires feature B means that the selection of feature A implies the selection of B.	
	Excludes	Feature A excludes feature B means that A and B can not be part of the same product.	

Table 3: Classical feature diagram notation.

3.2 VARIABILITY-AWARE ANALYSIS

Variability-aware analysis is a strategy to analyse software product lines that operates not on individual products (as opposed to *product-based analysis* (Thüm et al., 2015)), but on domain artifacts, that still contain variability and available configuration knowledge. This type of analysis avoids redundancy and prevents duplicated analyses, because there is no need to generate and analyse individual products. Furthermore, in principle, it only depends on the number and size of features, and not on the number of valid configurations.

However, changing at least one feature, or changing the variability model, means that we need to analyse the whole product line again (Cordy et al., 2012). Also, variability-aware analyses assume a closed world, meaning that all features must be known at the moment of the analyses.

3.2.1 *Automated analysis of feature models*

Manually analysing feature models is a tedious and error-prone task that becomes impractical in a large-scale context. Although feature models have gained a lot of popularity since they were introduced, for a long time, automated tool support was ad-hoc, offering little to no support for debugging and optimizations. Work from people like Mannion (2002) and Batory (2005) showed how feature models could be specified using propositional logic, enabling the use of off-the-shelf tools, like SAT-solvers, to verify properties of the models. These, and other similar work (for example, (Benavides et al., 2005b)) opened up new possibilities for tools to specify products in software product lines. However, there is not a consensus on which operations should be included in the analysis of feature models. Some of the most commonly found in research works are:

- *Void feature model* - A feature model is void if it does not represent any product.
- *Valid product* - A product is valid if it belongs to the set of products defined by the feature model.
- *Dead features* - A dead feature is one that does not appear in any of the products of the software product line.
- *Number of products* - The number of products that can be represented by the feature model.

Benavides et al. (2010) gives a nice overview of the proposed operations, along with a review of the work made on implementing those operations until that moment.

Semantics

As aforementioned, the set of configurations represented by a feature model can be described by a propositional logic formula. This formula is defined over a set $\mathbb{F}$ of boolean variables, where each variable corresponds to a feature. Table 4 shows the general mapping of the feature model relationships to propositional logic.

If we apply this mappings to the feature model of Figure 12a, for instance, we get the following formula:

$$(\text{Movement} \Leftrightarrow \text{CT}) \land (\text{Safety} \Rightarrow \text{CT}) \land (\text{Random} \lor \text{Teleop} \Leftrightarrow \text{Movement}) \tag{2}$$

Feature Model Primitive	**Propositional Formula**
r is the root feature	r
f_1 is an optional feature of f_2	$f_1 \Rightarrow f_2$
f_1 is a mandatory feature of f_2	$f_1 \Leftrightarrow f_2$
$f_1, ..., f_n$ are alternative sub-features of f	$(f_1 \vee ... \vee f_n \Leftrightarrow f) \wedge \bigwedge_{1 \leq i < j \leq n} \neg(f_i \wedge f_j)$
$f_1, ..., f_n$ are or sub-features of f	$f_1 \vee ... \vee f_n \Leftrightarrow f$
f_1 requires f_2	$f_1 \Rightarrow f_2$
f_1 excludes f_2	$\neg(f_1 \wedge f_2)$

Table 4: Mapping between the feature model primitives and propositional logic.

Having the feature model as a propositional formula allows us to easily specify some analysis operations. For example, the four operations that were mentioned above can be defined as:

$$\text{Void feature model} = \neg\text{SAT}(FM) \tag{3}$$

$$\text{Valid product } p = \text{SAT}(FM \wedge p) \tag{4}$$

$$\text{Dead feature } f = \neg\text{SAT}(FM \wedge f) \tag{5}$$

$$\text{Number of products} = \#\text{SAT}(FM) \tag{6}$$

Here, SAT denotes a procedure that checks the *satisfiability* of a boolean formula, and #SAT a procedure that counts the number of interpretations that satisfy a boolean formula. This is known as the *Sharp Satisfiability Problem* (Valiant, 1979).

3.2.2 *Variational software analysis*

There has been a considerable amount of work on the static analysis of variational computer programs. Most of it has been in data-flow analysis, be it intraprocedural (Brabrand et al., 2012; Liebig et al., 2013; Midtgaard et al., 2015) or interprocedural (Bodden, 2012). Other people have proposed new static analyses, specific to variational programs (Ribeiro et al., 2010; Tartler et al., 2011; Adelsberger et al., 2013; Sabouri and Khosravi, 2014). To note that the majority of these works are on annotative approaches of implementation, with exception of the work by Adelsberger et al. (2013) and Sabouri and Khosravi (2014).

Usually, when performing static analysis on programs, we operate on data structures, like queues, abstract syntax trees, and control-flow graphs. With variational programs, such as software product lines, this analysis needs to be lifted in order to handle the variation. More precisely, we need to lift the data structures and, consequently, the algorithms, needed to perform it. In particular, Brabrand et al. (2012) show how to lift a classic intraprocedural data-flow analysis to a variability-aware analysis.

3.2.3 *Variational Data Structures*

Presence Conditions

Presence conditions are formulas that represent a set of configurations.

$$\mathbf{Form} \ni \phi \quad ::= \quad \bot \mid \top \mid f \in \mathbb{F} \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

For example, if we look at the feature model from our previous example, the formula *SC* $\wedge$ *RM* represents all the configurations in which both the *SC* and *RM* features are present, i.e. $\{$*RM, SC*$\}$ and $\{$*CM, RM, SC*$\}$. This means that, if we are in one of these configurations, the presence condition *SC* $\wedge$ *RM* will always evaluate to **true**.

The function $\mathsf{eval}_c : \mathbf{Form} \rightarrow \mathbb{B}$ tells us if a boolean formula ϕ is **true** in a configuration $c \subseteq \mathbb{F}$, and can be defined inductively as follows:

$$\begin{aligned}
\mathsf{eval}_c\ \bot = \mathbf{false} \quad & \mathsf{eval}_c\ \top = \mathbf{true} \\
\mathsf{eval}_c\ f\ = \mathbf{true} \quad & \equiv \quad f \in c \\
\mathsf{eval}_c\ \neg\phi = \mathbf{true} \quad & \equiv \quad \mathsf{eval}_c\ \phi = \mathbf{false} \\
\mathsf{eval}_c\ (\phi_1 \wedge \phi_2) = \mathbf{true} \quad & \equiv \quad \mathsf{eval}_c\ \phi_1 = \mathbf{true}\ \text{ and }\ \mathsf{eval}_c\ \phi_2 = \mathbf{true} \\
\mathsf{eval}_c\ (\phi_1 \vee \phi_2) = \mathbf{true} \quad & \equiv \quad \mathsf{eval}_c\ \phi_1 = \mathbf{true}\ \text{ or }\ \mathsf{eval}_c\ \phi_2 = \mathbf{true}
\end{aligned}$$

With this, we can define function $\mathsf{solve}_{\mathcal{FM}}$, which computes the set of configurations represented by a presence condition ϕ in a feature model $\mathcal{FM}$.

$$\begin{aligned}
&\mathsf{solve}_{\mathcal{FM}} : \mathbf{Form} \rightarrow \mathbf{Set}\ \mathbb{F} \\
&\mathsf{solve}_{\mathcal{FM}}\ \phi = \{c \mid c \subseteq \mathbb{F} \wedge \mathsf{eval}_c\ \phi\}
\end{aligned}$$

Here, we have some examples of presence conditions and the respective set of configurations, for a $\mathcal{FM} = \mathbf{true}$, and with features $\{A, B\}$:

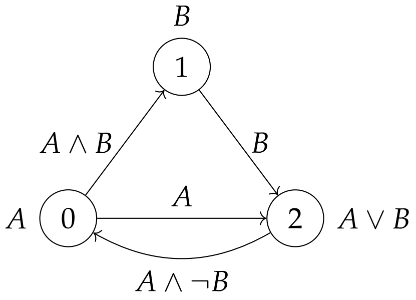
$$\begin{aligned}
\mathsf{solve}_{\mathcal{FM}}\ \mathbf{true} &= \{\emptyset, \{A\}, \{B\}, \{A, B\}\} \\
\mathsf{solve}_{\mathcal{FM}}\ \mathbf{false} &= \emptyset \\
\mathsf{solve}_{\mathcal{FM}}\ A &= \{\{A\}, \{A, B\}\} \\
\mathsf{solve}_{\mathcal{FM}}\ (\neg A) &= \{\emptyset, \{B\}\} \\
\mathsf{solve}_{\mathcal{FM}}\ (A \wedge B) &= \{\{A, B\}\} \\
\mathsf{solve}_{\mathcal{FM}}\ (A \vee B) &= \{\{A\}, \{B\}, \{A, B\}\}
\end{aligned}$$

Variational Graphs

According to Erwig et al. (2013), a *variational graph* $\vec{G}$ is a graph in which both the nodes and the edges are annotated with presence conditions. More formally, $\vec{G}$ is a structure $(\vec{V}, \vec{E})$ in which $\vec{V} : V \rightarrow \mathbf{Form}$ and $\vec{E} : E \rightarrow \mathbf{Form}$ are mappings from nodes to presence conditions and edges to presence conditions, respectively. Figure 13 shows an example of a variational graph.

$$
\begin{aligned}
&\vec{G} = (\vec{V}, \vec{E}) \ \mathbf{where} \\
&\quad \vec{V} = \{0 \mapsto A, 1 \mapsto B, 2 \mapsto A \vee B\} \\
&\quad \vec{E} = \{(0,1) \mapsto A \wedge B, (0,2) \mapsto A, \\
&\qquad\quad (1,2) \mapsto B, (2,0) \mapsto A \wedge \neg B\}
\end{aligned}
$$

(a) Definition.

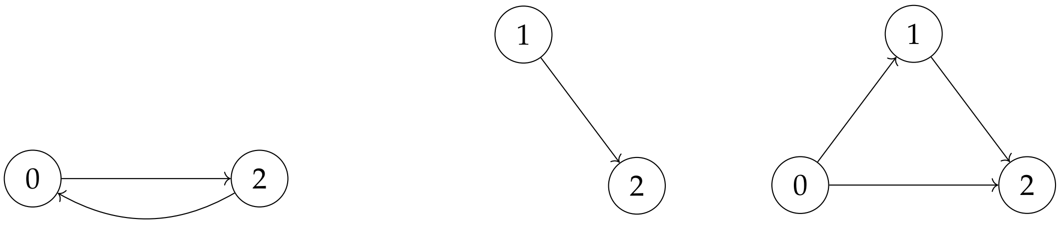


(b) Diagram.

Figure 13: Variational graph.

Each variational graph represents a set of simple graphs, where each one can be obtained by projecting a certain configuration in the variational graph (Figure 14). A *projection* p_c with a configuration c consists of a simple graph that can be obtained by filtering the nodes and edges of a variational graph whose presence condition evaluates to $\mathbf{true}$ in configuration c, i.e.

$$
\begin{aligned}
&p_c\ (\vec{V}, \vec{E}) = (V, E) \ \mathbf{where} \\
&\quad V = \{v \mid v \mapsto \phi \in \vec{V} \wedge \mathsf{eval}_c\ \phi\} \\
&\quad E = \{e \mid e \mapsto \phi \in \vec{E} \wedge \mathsf{eval}_c\ \phi\}
\end{aligned}
$$



(a) $\{A\}$ (b) $\{B\}$ (c) $\{A, B\}$

Figure 14: Some projections of the variational graph from Figure 13.

4

ENCODING INFORMATION IN A VARIATIONAL DATABASE

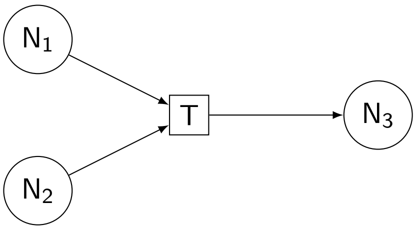
In this chapter we define a structure that can hold variational information, called a *variational database*.

4.1 NON-VARIATIONAL DATABASES

Before defining a structure that can represent variational data, we should first define one that can represent non-variational data. In the context of this thesis, we want to define a structure that represents a single product of a SPL. We call this structure a *simple database*.

4.1.1 *Relations*

At the core of a simple database is the concept of *relation*. A *n-ary relation* R^n of type $T_1 \times \cdots \times T_n$ is a set of tuples $(x_1, \ldots, x_n)$, so that $x_1 \in T_1, \ \ldots, \ x_n \in T_n$. We say that $x_1, \ldots, x_n$ are all related by R^n. With such databases we can represent almost everything. For example, a directed graph is defined by two relations: an unary relation that represents the set of vertices of the graph (V); and a binary relation that specifies the edges of the graph (E). A computation graph can also be represented in terms of relations. Let's see,



Here, we have three nodes, N_1, N_2 and N_3, and one topic T. Nodes N_1 and N_2 publish in T and T is subscribed by N_3. This computation graph can be represented by defining four relations:

1. Two unary relations (sets): $\{\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3\}$ and $\{\mathsf{T}\}$, that define the nodes and topics, respectively, of the computation graph.

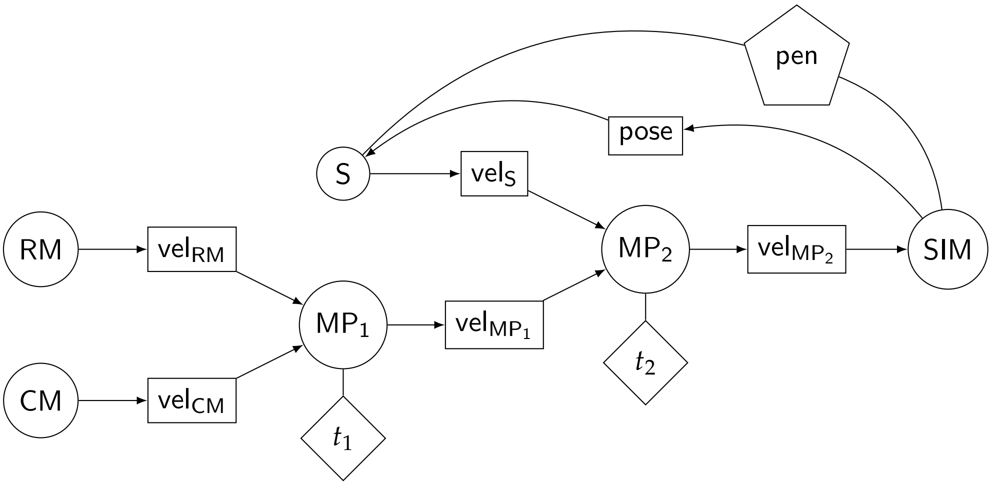
2. Two binary relations: $\{(\mathsf{N}_3, \mathsf{T})\}$ and $\{(\mathsf{N}_1, \mathsf{T}), (\mathsf{N}_2, \mathsf{T})\}$, that specify which nodes are subscribed to which topics and which nodes publish in which topics, respectively.

4.1.2 *Definition*

With this concept, we define a simple database $D : \mathbf{Id} \hookrightarrow R$ as a partial function that maps names to relations, or, in other words, a collection of named relations. Looking at the previous example, we can represent the computation graph with the following database:

$$\begin{aligned} \mathsf{db} = \{ \quad & \text{Node} \mapsto \{\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3\}, \\ & \text{Topic} \mapsto \{\mathsf{T}\}, \\ & \text{subscribes} \mapsto \{(\mathsf{N}_3, \mathsf{T})\}, \\ & \text{publishes} \mapsto \{(\mathsf{N}_1, \mathsf{T}), (\mathsf{N}_2, \mathsf{T})\} \quad \} \end{aligned}$$

Although this is a simple example, the same idea can be applied to more complex ones. Recall the computation graph of Configuration 6 from our running example.



In this computation graph, there are also services and parameters. We can simply accommodate to them by adding more relations.

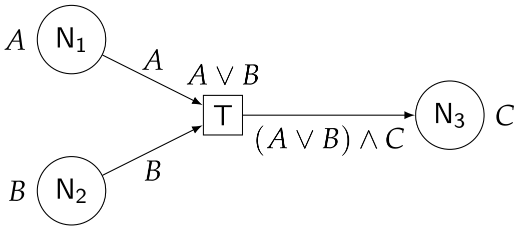
$$\begin{aligned} \mathsf{C6} = \{ \quad & \text{Node} \mapsto \{\mathsf{RM}, \mathsf{CM}, \mathsf{S}, \mathsf{MP}_1, \mathsf{MP}_2, \mathsf{SIM}\}, \\ & \text{Topic} \mapsto \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \\ & \text{Param} \mapsto \{t_1, t_2\}, \quad \text{Service} \mapsto \{\mathsf{pen}\}, \\ & \text{subscribes} \mapsto \{(\mathsf{MP}_1, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{CM}}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP}_1}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{S}}), \\ & (\mathsf{SIM}, \mathsf{vel}_{\mathsf{MP}_2}), (\mathsf{S}, \mathsf{pose})\}, \\ & \text{publishes} \mapsto \{(\mathsf{RM}, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{CM}, \mathsf{vel}_{\mathsf{CM}}), (\mathsf{S}, \mathsf{vel}_{\mathsf{S}}), \\ & (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{MP}_1}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP}_2}), (\mathsf{SIM}, \mathsf{pose})\}, \\ & \text{params} \mapsto \{(\mathsf{MP}_1, t_1), (\mathsf{MP}_2, t_2)\}, \quad \text{services} \mapsto \{(\mathsf{S}, \mathsf{pen}, \mathsf{SIM})\} \quad \} \end{aligned}$$

4.2 VARIATIONAL DATABASES - THE NON-VARIATIONAL WAY

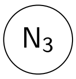
Unlike simple databases, that represent only a product, variational databases are supposed to represent the whole SPL. We have two approaches for creating a variational database.

Although it sounds a bit contradictory, our first approach for defining a variational database is called the *non-variational* approach. The reason for this is that, in this version, we do not create a structure that has variational information (with presence conditions). Instead, we define a structure that holds all the different products of the SPL, separately.

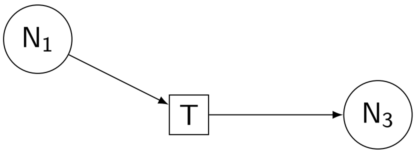
A *non-variational variational database* $\vec{D}_{nv} : \mathbb{C} \hookrightarrow D$ is a partial function that maps a configuration to a simple database, that in turn holds all the information of a single product. Suppose we have the following computation graph,



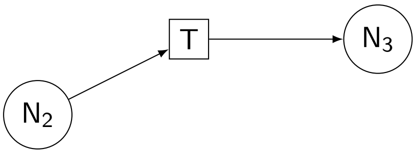
that is the implementation of a SPL with the following feature model $\mathcal{FM} = (A \Rightarrow C) \wedge (B \Rightarrow C) \wedge C$. $\mathcal{FM}$ has four possible configurations – $\{C\}$, $\{A, C\}$, $\{B, C\}$ and $\{A, B, C\}$ – which means there are four different computations graphs (and four simple databases), respectively:



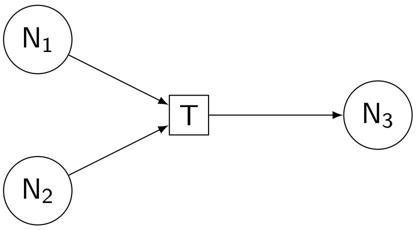
$$
\begin{aligned}
\text{db1} = \{\quad & \text{Node} \mapsto \{\mathsf{N_3}\}, \\
& \text{Topic} \mapsto \emptyset, \\
& \text{subscribes} \mapsto \emptyset, \\
& \text{publishes} \mapsto \emptyset \quad \}
\end{aligned}
$$



$$
\begin{aligned}
\text{db2} = \{\quad & \text{Node} \mapsto \{\mathsf{N_1}, \mathsf{N_3}\}, \\
& \text{Topic} \mapsto \{\mathsf{T}\}, \\
& \text{subscribes} \mapsto \{(\mathsf{N_3}, \mathsf{T})\}, \\
& \text{publishes} \mapsto \{(\mathsf{N_1}, \mathsf{T})\} \quad \}
\end{aligned}
$$



$$
\begin{aligned}
\text{db3} = \{\quad & \text{Node} \mapsto \{\mathsf{N_2}, \mathsf{N_3}\}, \\
& \text{Topic} \mapsto \{\mathsf{T}\}, \\
& \text{subscribes} \mapsto \{(\mathsf{N_3}, \mathsf{T})\}, \\
& \text{publishes} \mapsto \{(\mathsf{N_2}, \mathsf{T})\} \quad \}
\end{aligned}
$$



$$\mathsf{db4} = \{ \quad \text{Node} \mapsto \{\mathsf{N}_1, \mathsf{N}_2, \mathsf{N}_3\}, \\ \text{Topic} \mapsto \{\mathsf{T}\}, \\ \text{subscribes} \mapsto \{(\mathsf{N}_3, \mathsf{T})\}, \\ \text{publishes} \mapsto \{(\mathsf{N}_1, \mathsf{T}), (\mathsf{N}_2, \mathsf{T})\} \quad \}$$

The resulting variational database is:

$$\mathsf{db} = \{ \quad \{C\} \mapsto \mathsf{db1}, \\ \{A, C\} \mapsto \mathsf{db2}, \\ \{B, C\} \mapsto \mathsf{db3}, \\ \{A, B, C\} \mapsto \mathsf{db4} \quad \}$$

4.3 VARIATIONAL DATABASES - THE PURE VARIATIONAL WAY

The other approach for defining a variational database is called the *pure variational* approach. Here, instead of saving all products separately in various databases, we have just a single database, that is a collection of *variational relations*.

4.3.1 *Variational Relations*

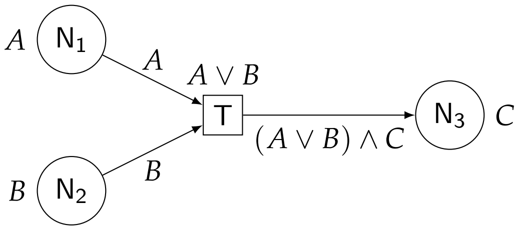
A *variational relation* is a relation in which all its elements are annotated with presence conditions. More formally, a n*-ary variational relation* $\vec{R^n}$ of type $T_1 \times \cdots \times T_n$ is a partial function that maps tuples $(x_1, \ldots, x_n)$ to presence conditions, of type:

$$\vec{R^n} : T_1 \times \cdots \times T_n \hookrightarrow \textbf{Form}$$

Given $x_1 \in T_1, \ \ldots, \ x_n \in T_n$ if we have the mapping

$$(x_1, \ldots, x_n) \mapsto \phi$$

we say that $x_1, \ldots, x_n$ are all related by $\vec{R^n}$ if ϕ is true. With these, we can represent variational data. In fact, a variational graph can be encoded in a variational database with two variational relations, as we could see in Figure 13a. Predictably, we can also use variational relations to represent variational computation graphs. Take a look at the previous example:



In order to represent this with normal relations, we had to create a set of relations for each of the different products. Now, with variational relations, we only need to define four (variational) relations to describe it entirely:

1. Two unary variational relations - $\{\mathsf{N}_1 \mapsto A, \mathsf{N}_2 \mapsto B, \mathsf{N}_3 \mapsto C\}$, that represents the nodes, and $\{\mathsf{T} \mapsto A \vee B\}$, which represents the set of topics.

2. Two binary variational relations - $\{(\mathsf{N}_3, \mathsf{T}) \mapsto (A \vee B) \wedge C\}$, for representing the subscribers, and $\{(\mathsf{N}_1, \mathsf{T}) \mapsto A, (\mathsf{N}_2, \mathsf{T}) \mapsto B\}$, that represents the publishers.

4.3.2 *Definition*

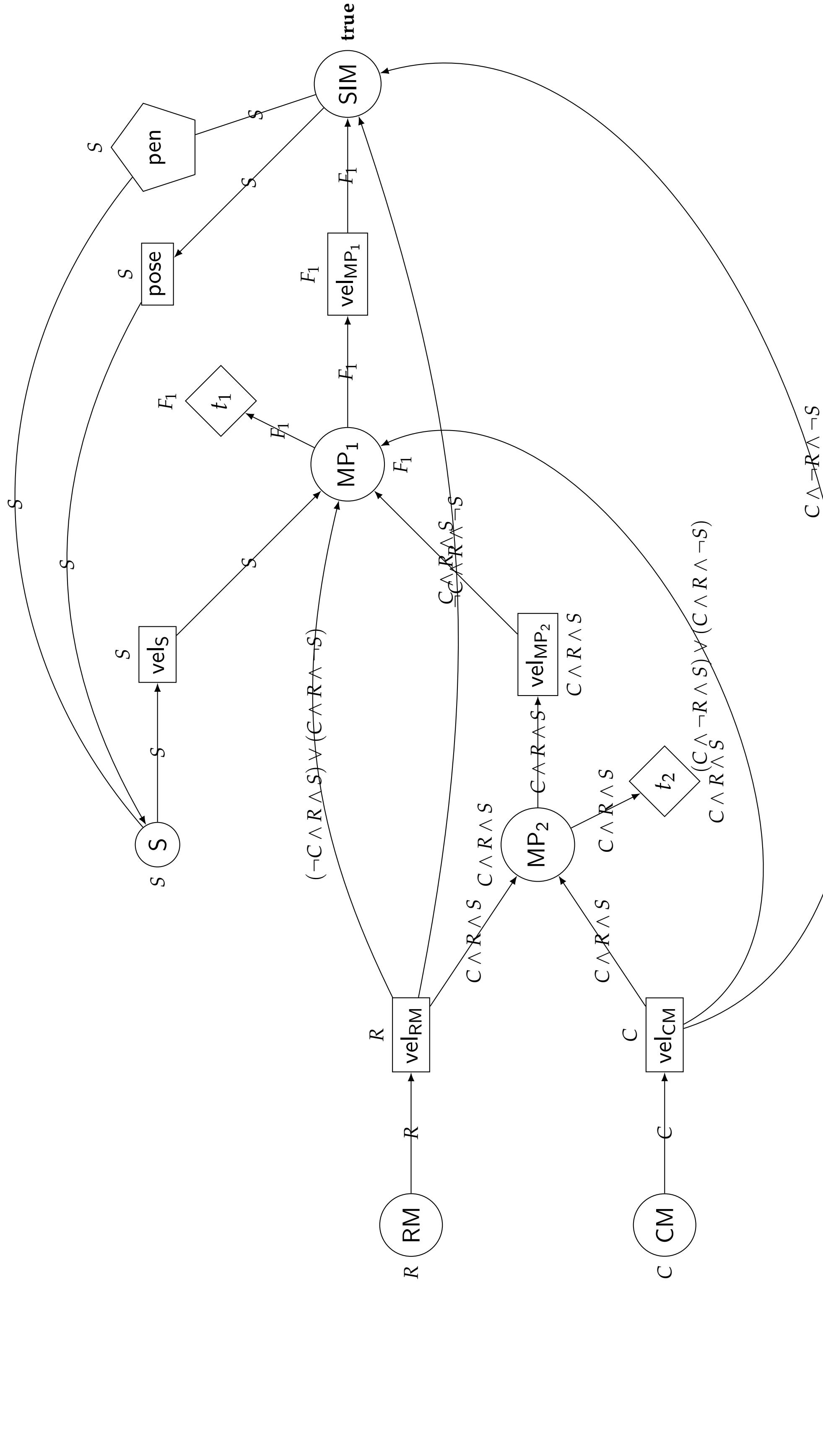
Similarly to the definition of a simple database, a *pure variational database* $\vec{D}_{pv} : \mathbf{Id} \hookrightarrow \vec{R}$ is a partial function that maps names to variational relations. With this, we can represent the previous example with the following database:

$$
\begin{aligned}
\mathsf{db} = \{ \quad & \text{Node} \mapsto \{\mathsf{N}_1 \mapsto A, \mathsf{N}_2 \mapsto B, \mathsf{N}_3 \mapsto C,\} \\
& \text{Topic} \mapsto \{\mathsf{T} \mapsto A \vee B\}, \\
& \text{subscribes} \mapsto \{(\mathsf{N}_3, \mathsf{T}) \mapsto (A \vee B) \wedge C\}, \\
& \text{publishes} \mapsto \{(\mathsf{N}_1, \mathsf{T}) \mapsto A, (\mathsf{N}_2, \mathsf{T}) \mapsto B\} \quad \}
\end{aligned}
$$

We can also represent the CT system database:

$$
\begin{aligned}
\mathsf{db} = \{ \quad & \text{Node} \mapsto \{\mathsf{RM} \mapsto R, \mathsf{CM} \mapsto C, \mathsf{S} \mapsto S,\\
& \qquad \mathsf{MP}_1 \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S),\\
& \qquad \mathsf{MP}_2 \mapsto (C \wedge R \wedge S), \mathsf{SIM} \mapsto \textbf{true}\}\\
& \text{Topic} \mapsto \{\mathsf{vel}_{\mathsf{RM}} \mapsto R, \mathsf{vel}_{\mathsf{CM}} \mapsto C, \mathsf{vel}_{\mathsf{S}} \mapsto S,\\
& \qquad \mathsf{vel}_{\mathsf{MP1}} \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S),\\
& \qquad \mathsf{vel}_{\mathsf{MP2}} \mapsto (C \wedge R \wedge S), \mathsf{pose} \mapsto S\}\\
& \text{Param} \mapsto \{t_1 \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S), t_2 \mapsto (C \wedge R \wedge S)\},\\
& \text{Service} \mapsto \{\mathsf{pen} \mapsto S\},\\
& \text{subscribes} \mapsto \{(\mathsf{SIM}, \mathsf{vel}_{\mathsf{MP1}}) \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S),\\
& \qquad\quad (\mathsf{SIM}, \mathsf{vel}_{\mathsf{RM}}) \mapsto \neg C \wedge R \wedge \neg S, (\mathsf{SIM}, \mathsf{vel}_{\mathsf{CM}}) \mapsto \wedge \neg R \wedge \neg S,\\
& \qquad\quad (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{S}}) \mapsto S, (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{RM}}) \mapsto (\neg C \wedge R \wedge S) \vee (C \wedge R \wedge \neg S),\\
& \qquad\quad (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{CM}}) \mapsto (C \wedge \neg R \wedge S) \vee (C \wedge R \wedge \neg S),\\
& \qquad\quad (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{MP2}}) \mapsto (C \wedge R \wedge S), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{RM}}) \mapsto (C \wedge R \wedge S),\\
& \qquad\quad (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{CM}}) \mapsto (C \wedge R \wedge S), (\mathsf{S}, \mathsf{pose}) \mapsto S\},\\
& \text{publishes} \mapsto \{(\mathsf{SIM}, \mathsf{pose}) \mapsto S, (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{MP1}} \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S),\\
& \qquad\quad (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP2}} \mapsto (C \wedge R \wedge S), (\mathsf{S}, \mathsf{vel}_{\mathsf{S}}) \mapsto S,\\
& \qquad\quad (\mathsf{RM}, \mathsf{vel}_{\mathsf{RM}} \mapsto R, (\mathsf{CM}, \mathsf{vel}_{\mathsf{CM}}) \mapsto C\},\\
& \text{params} \mapsto \{(\mathsf{MP}_1, t_1) \mapsto (C \wedge R) \vee (C \wedge S) \vee (R \wedge S), (\mathsf{MP}_2, t_2) \mapsto C \wedge R \wedge S\},\\
& \text{services} \mapsto \{(\mathsf{SIM}, \mathsf{pen}, \mathsf{S}) \mapsto S\},\\
\}&
\end{aligned}
$$

The next figure shows the diagram relative to the CT database.



$$F_1 = (C \wedge R) \vee (C \wedge S) \vee (R \wedge S)$$

4.3.3 *Pure variational databases with sets*

Instead of using formulas as presence conditions, we can use sets of configurations. The idea is the same as before, only the representation of presence conditions changes. In this case, a variational relation has the type:

$$\vec{R^n} : T_1 \times \cdots \times T_n \hookrightarrow \textbf{Set}\ (\textbf{Set}\ \mathbb{F})$$

and

$$(x_1, \ldots, x_n) \mapsto S$$

means that $(x_1, \ldots, x_n)$ exists in any configuration $c \in S$.

To avoid ambiguity, we will call the pure variational database that uses formulas as *pure - formulas* variational database, and this one, that uses sets, as *pure - sets* variational database.

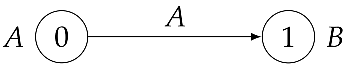
5

PROPERTY EVALUATION IN VARIATIONAL DATABASES

In this chapter we introduce a query language for specifying properties, and create two different algorithms for executing them on variational databases.

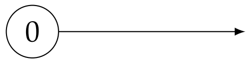
5.1 A QUERY LANGUAGE FOR SPECIFYING PROPERTIES

In the previous chapter, we introduced variational databases that can hold variational information, for example a variational ROS computation graph. However, the information recorded in such variational databases may not always be correct. Take a look at the following variational graph G,



$$\begin{aligned} G = \{V &\mapsto \{0 \mapsto A, 1 \mapsto B\} \\ E &\mapsto \{(0,1) \mapsto A\}\} \end{aligned}$$

If we project this graph with configuration $\{A\}$, we get the following result,



$$\begin{aligned} G|_{\{A\}} = \{V &\mapsto \{0\} \\ E &\mapsto \{(0,1)\}\} \end{aligned}$$

which is an invalid graph because we have an edge that does not have a valid node as its destination. These types of problems can occur rather frequently when dealing with variational data. To see if a variational data structure, or in our case, a variational database is correct, we can specify properties to be checked. For example, in this case, we could say that a property for the correctness of variational graphs is "*An edge always connects two existing nodes.*". As another example, if we look at computation graphs, "*A topic which is subscribed must also be published.*" or "*A computation graph must always have at least one node.*" are examples of properties we may want to verify.

5.1.1 *Language Syntax*

For the purpose of specifying properties, we created a small language, inspired by the Alloy Specification Language[1], that has the following *syntax*:

$$\begin{array}{rcl} \mathbf{Query} \ni q & ::= & q_1 \;\mathbf{or}\; q_2 \mid q_1 \;\mathbf{and}\; q_2 \mid \mathbf{not}\; q \mid \mathbf{some}\; r \mid \mathbf{no}\; r \mid r_1 \;\mathbf{in}\; r_2 \\ \mathbf{RExp} \ni r & ::= & x \mid {\sim}r \mid r_1 + r_2 \mid r_1 \,\&\, r_2 \mid r_1 - r_2 \mid r_1 \,.\, r_2 \end{array}$$

A property (or query) is defined in terms of relational and boolean expressions. Relational expressions allow us to manipulate data. Boolean expressions provide us with the result we expect from a property verification: a **true** or a **false**.

As an example, recall the "*A computation graph must always have at least one node.*" property for computation graphs. This property can be specified in our language by:

$$\mathbf{some}\ \mathsf{Node}$$

In turn, "*A topic which is subscribed must also be published.*" could translate to:

$$\mathsf{Node.subscribes}\ \mathbf{in}\ \mathsf{Node.publishes}$$

5.1.2 *Language Semantics*

As we said earlier, the execution of a query returns a boolean value. To determine this value, we need to know the meaning of the operations, i.e. the language's *semantics*. The semantics of our language are defined by two functions:

$$\mathcal{Q}[\![\cdot]\!] : \mathbf{Query} \to D \to \mathbb{B}$$
$$\mathcal{R}[\![\cdot]\!] : \mathbf{RExp} \to D \to R$$

$\mathcal{Q}[\![\cdot]\!]$ is the semantic function of the boolean expressions, and consequently, the queries. And $\mathcal{R}[\![\cdot]\!]$ is the semantic function for the relational expressions. Both functions are defined by pattern-matching in terms of the respective derivations in the syntax.

Let's start with $\mathcal{R}[\![\cdot]\!]$. This function receives an relational expression r and a simple database d, and returns a relation. Depending on the expression, we will have different function definitions. Let's look at the first one:

$$\mathcal{R}[\![x]\!]_d \;\doteq\; d\; x \qquad (7)$$

[1] https://alloytools.org

Here, our expression is x, which represents an identifier, or, in other words, a name. In this case, we want to go to the database, and retrieve the relation that has the name x. What we do to achieve that is simply call the database function with x as its argument.

Next, we have the *converse* expression, that essentially inverts the result of an expression:

$$\mathcal{R}[\![\sim r]\!]_d \quad \doteq \quad (\mathcal{R}[\![r]\!]_d)^\circ \tag{8}$$

where,

$$\begin{aligned} &(\_)^\circ : R^n \rightarrow R^n \\ &r^\circ = \{(x_n, \ldots, x_1) \mid (x_1, \ldots, x_n) \in r\} \end{aligned}$$

The next definitions are trivial, with the expressions translating directly to set operations:

$$\mathcal{R}[\![r_1 + r_2]\!]_d \quad \doteq \quad \mathcal{R}[\![r_1]\!]_d \cup \mathcal{R}[\![r_2]\!]_d \tag{9}$$

$$\mathcal{R}[\![r_1 \,\&\, r_2]\!]_d \quad \doteq \quad \mathcal{R}[\![r_1]\!]_d \cap \mathcal{R}[\![r_2]\!]_d \tag{10}$$

$$\mathcal{R}[\![r_1 - r_2]\!]_d \quad \doteq \quad \mathcal{R}[\![r_1]\!]_d \setminus \mathcal{R}[\![r_2]\!]_d \tag{11}$$

Lastly, we have the composition expression:

$$\mathcal{R}[\![r_1 \,.\, r_2]\!]_d \quad \doteq \quad \mathcal{R}[\![r_1]\!]_d \cdot \mathcal{R}[\![r_2]\!]_d \tag{12}$$

We separated this from the previous ones because the $\cdot$ operator is a little different from the normal relational composition. Usually, the relational composition is between binary relations. In this case, we want it to be between n-ary relations. So we will make the composition with regards to the first and last elements of the tuples, and return all the other elements in one tuple.

$$\begin{aligned} &(\cdot) : R^n \rightarrow R^m \rightarrow R^{n+m-2} \\ &r \cdot s = \{(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) \mid (x_1, \ldots, x_n) \in r \wedge (y_1, \ldots, y_m) \in s \wedge x_n = y_1\} \end{aligned}$$

This operator allow us to compose relations with different arity, which is something that can be useful.

On the other hand, $\mathcal{Q}[\![\cdot]\!]$ is the semantic function of the boolean operations. It receives a query q, a simple database d, and returns the result (boolean) of executing q in d. The first three expressions represent the boolean operations: *disjunction*, *conjunction* and *negation*; and their definitions are trivial:

$$\mathcal{Q}[\![q_1 \textbf{ or } q_2]\!]_d \quad \doteq \quad \mathcal{Q}[\![q_1]\!]_d \vee \mathcal{Q}[\![q_2]\!]_d \tag{13}$$

$$\mathcal{Q}[\![q_1 \textbf{ and } q_2]\!]_d \quad \doteq \quad \mathcal{Q}[\![q_1]\!]_d \wedge \mathcal{Q}[\![q_2]\!]_d \tag{14}$$

$$\mathcal{Q}[\![\textbf{not } q]\!]_d \quad \doteq \quad \neg\mathcal{Q}[\![q]\!]_d \tag{15}$$

The remaining ones are those which interact with relational expressions. The $\mathbf{some}\ r$ expression tells if a relation r has at least one element, i.e. is not empty:

$$\mathcal{Q}[\![\mathbf{some}\ r]\!]_d \quad \doteq \quad \mathcal{R}[\![r]\!]_d \neq \emptyset \tag{16}$$

The $\mathbf{no}\ r$ is precisely the opposite of $\mathbf{some}\ r$, as it returns $\mathbf{true}$ when r is empty:

$$\mathcal{Q}[\![\mathbf{no}\ r]\!]_d \quad \doteq \quad \mathcal{R}[\![r]\!]_d = \emptyset \tag{17}$$

Lastly, we have the $r_1\ \mathbf{in}\ r_2$ expression, that corresponds to testing if r_1 is a subset of r_2:

$$\mathcal{Q}[\![r_1\ \mathbf{in}\ r_2]\!]_d \quad \doteq \quad \mathcal{R}[\![r_1]\!]_d \subseteq \mathcal{R}[\![r_2]\!]_d \tag{18}$$

Let's use the C6 configuration

$$\begin{aligned}
\text{C6} = \{ \quad & \text{Node} \mapsto \{\mathsf{RM}, \mathsf{CM}, \mathsf{S}, \mathsf{MP}_1, \mathsf{MP}_2, \mathsf{SIM}\}, \\
& \text{Topic} \mapsto \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \\
& \text{Param} \mapsto \{t\}, \quad \text{Service} \mapsto \{\mathsf{pen}\}, \\
& \text{subscribes} \mapsto \{(\mathsf{MP}_1, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP}_1}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{S}}), \\
& (\mathsf{SIM}, \mathsf{vel}_{\mathsf{MP}_2}), (\mathsf{S}, \mathsf{pose})\}, \\
& \text{publishes} \mapsto \{(\mathsf{RM}, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{CM}, \mathsf{vel}_{\mathsf{CM}}), (\mathsf{S}, \mathsf{vel}_{\mathsf{S}}), \\
& (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{MP}_1}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP}_2}), (\mathsf{SIM}, \mathsf{pose})\}, \\
& \text{params} \mapsto \{(\mathsf{MP}_1, t_1), (\mathsf{MP}_2, t_2)\}, \quad \text{services} \mapsto \{(\mathsf{S}, \mathsf{pen}, \mathsf{SIM})\} \quad \}
\end{aligned}$$

to test some queries:

$$
\begin{aligned}
& [\![\textbf{some}\ \mathsf{Node} + \mathsf{Topic}]\!] \\
\equiv\ & (16) \\
& [\![\mathsf{Node} + \mathsf{Topic}]\!] \neq \emptyset \\
\equiv\ & (9) \\
& [\![\mathsf{Node}]\!] \cup [\![\mathsf{Topic}]\!] \neq \emptyset \\
\equiv\ & (7)\ \times 2 \\
& \{\mathsf{RM}, \mathsf{CM}, \mathsf{S}, \mathsf{MP}_1, \mathsf{MP}_2, \mathsf{SIM}\} \cup \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \neq \emptyset \\
\equiv\ & \textit{def.}\ \cup \\
& \{\mathsf{RM}, \mathsf{CM}, \mathsf{S}, \mathsf{MP}_1, \mathsf{MP}_2, \mathsf{SIM}, \mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \neq \emptyset \\
\equiv\ & \textit{def.}\ \neq \\
& \textbf{true}
\end{aligned}
$$

$$
\begin{aligned}
& [\![\mathsf{Node.subscribes}\ \textbf{in}\ \mathsf{Topic}]\!] \\
\equiv\ & (18) \\
& [\![\mathsf{Node.subscribes}]\!] \subseteq [\![\mathsf{Topic}]\!] \\
\equiv\ & (12) \\
& [\![\mathsf{Node}]\!] \cdot [\![\mathsf{subscribes}]\!] \subseteq [\![\mathsf{Topic}]\!] \\
\equiv\ & (7)\ \times 3 \\
& \{\mathsf{RM}, \mathsf{CM}, \mathsf{S}, \mathsf{MP}_1, \mathsf{MP}_2, \mathsf{SIM}\} \cdot \{(\mathsf{MP}_1, \mathsf{vel}_{\mathsf{RM}}), (\mathsf{MP}_1, \mathsf{vel}_{\mathsf{CM}}), (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{MP}_1}), \\
& (\mathsf{MP}_2, \mathsf{vel}_{\mathsf{S}}), (\mathsf{SIM}, \mathsf{vel}_{\mathsf{MP}_2}), (\mathsf{S}, \mathsf{pose})\} \subseteq \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \\
\equiv\ & \textit{def.}\ \cdot \\
& \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \subseteq \{\mathsf{vel}_{\mathsf{RM}}, \mathsf{vel}_{\mathsf{CM}}, \mathsf{vel}_{\mathsf{S}}, \mathsf{vel}_{\mathsf{MP}_1}, \mathsf{vel}_{\mathsf{MP}_2}, \mathsf{pose}\} \\
\equiv\ & \textit{def.}\ \subseteq \\
& \textbf{true}
\end{aligned}
$$

5.2 EXECUTING QUERIES ON VARIATIONAL DATABASES

The goal of this query language is to specify properties to be checked on variational databases. For now, it is only obvious how to execute them on simple databases because those are the ones for which the semantics are defined for. As such, we need to define algorithms that will enable us to execute the queries on the variational databases.

First, we have to decide what the execution of these algorithms will return. In simple databases, the execution of a query returns a boolean value that tells us if the specified property is valid or not in that database. However, in a variational context, what does it mean to return a boolean value? Suppose that the execution of a certain query q in a variational database d returns **true**. This means that the property specified by q is valid in all possible configurations of d. On the other hand, returning **false** tells us that there is at least one configuration of d where the specified property is not valid. The downside to this is that we do not know which configurations failed, we just know that some did and some did not. A better option would be to somehow return something that would give us information about which configurations satisfied the property, for example the set of configurations in which the property is valid.

Depending on the type of the variational database (non-variational or pure), we have different algorithms to achieve this execution.

5.2.1 *Non-variational*

First, we have the *non-variational* variational database. If we recall, this variational database is a collection of simple databases, where each one has a configuration associated. The execution strategy for this type of databases is pretty straightforward. We execute the query in each simple database, and if the result is **true** we add the configuration of that specific database to the resulting set of configurations, if it is **false**, we do nothing.

This algorithm can be easily defined by set comprehension:

$$\begin{aligned} &\mathsf{run}_{nv} : \vec{D}_{nv} \rightarrow \mathbf{Query} \rightarrow \mathbf{Set}\,\mathbb{C} \\ &\mathsf{run}_{nv}\ d\ q = \{c \mid c \subseteq \mathbb{F} \land [\![q]\!]_{(d\ c)} = \mathbf{true}\} \end{aligned}$$

Let's see the execution of the "**some** Node" query on the non-variational variational database of CT:

$$\begin{aligned} & \mathsf{run}_t\ d\ (\mathbf{some}\ \mathsf{Node}) \\ \equiv \quad & \textit{def.}\ \mathsf{run}_t \\ & \{c \mid c \subseteq \mathbb{F} \land [\![q]\!]_{(d\ c)} = \mathbf{true}\} \end{aligned}$$

Now we have to execute the query on each database:

- **case 1**:

$$\begin{aligned} c &= \{\mathsf{C}\} \\ d &= \mathsf{db1} \end{aligned}$$

$$\begin{aligned}
& [\![\mathbf{some}\ \mathsf{Node}]\!] \\
\equiv\ & (16) \\
& [\![\mathsf{Node}]\!] \neq \emptyset \\
\equiv\ & (7) \\
& \{\mathsf{CM}, \mathsf{SIM}\} \neq \emptyset \\
\equiv\ & \textit{def.} \neq \\
& \mathbf{true}
\end{aligned}$$

- ...

We are left with:

$$\begin{aligned}
& \{c \mid c \subseteq \mathbb{F} \wedge [\![q]\!]_{(d\ c)} = \mathbf{true}\} \\
\equiv\ & \\
& \{\{C\}, \{R\}, \{C, R\}, \{C, S\}, \{R, S\}, \{C, R, S\}\}
\end{aligned}$$

5.2.2 *Pure Variational*

Next, we have the purely defined variational databases, which are collections of variational relations. As it is expected, the algorithms for these ones should be a little more complicated than the non-variational one, because we are dealing with more complex data structures. Just like before, we want the queries to return information about the configurations. For their execution, we came up with two different approaches.

Trivial

The first one is what we call the *trivial* approach. In this algorithm, we will execute two steps for each configuration of the variational database:

1. Project the variational relations mentioned in the query with the current configuration.
2. Execute the query on the projected relations and, if it returns **true**, add the configuration to the result.

$$\begin{aligned}
& \mathsf{run}_t : \vec{D}_{pv} \rightarrow \mathbf{Query} \rightarrow \mathbf{Set}\,\mathbb{C} \\
& \mathsf{run}_t\ d\ q = \{c \mid c \subseteq \mathbb{F} \wedge [\![q]\!]_{d|_c} = \mathbf{true}\}
\end{aligned}$$

Pure - formulas & sets

Unlike the other approaches, this one is purely variational, since we will not make any projections. Instead, we will make use of the presence conditions present in each variational relation of the variational database, and build a final formula that, when solved, it will return the set of configurations where the property is valid.

The algorithm has the following type:

- *pure - formulas*

$$\mathsf{run}_{fp} : \vec{D}_{fv} \to \textbf{Query} \to \textbf{Form}$$

- *pure - sets*

$$\mathsf{run}_{sp} : \vec{D}_{sv} \to \textbf{Query} \to \textbf{Set}\ (\textbf{Set}\ \mathbb{F})$$

In this algorithm, executing a boolean expression returns a propositional formula. On the other hand, the execution of a relational expression returns a variational relation.

Once again, let's begin with the relational expressions. The first one, x, is trivial. The variational database is a collection of named variational relations, so we just return the variational relation with said name x:

- *pure - formulas*

$$\mathsf{run}_{fp}\ d\ x = d\ x \tag{19}$$

- *pure - sets*

$$\mathsf{run}_{sp}\ d\ x = d\ x \tag{20}$$

Regarding the others, the idea is the same as when defining the semantics, i.e, each expression will "associate" with a relation operation. Only this time, instead of simple relations, we are dealing with variational relations. So, we need to define the union, intersection, etc. operations for variational relations.

Let's start with the most simple one, the *variational converse* operation. Here, we want to convert all tuples from a variational relation in their symmetrical, without changing their presence conditions.

- *pure - formulas*

$$\begin{aligned}&(\_)^{\vec{\circ}} : \vec{R^n} \to \vec{R^n}\\ &r^{\vec{\circ}} = \{(x_n, \ldots, x_1) \mapsto \phi \mid r\ (x_1, \ldots, x_n) = \phi\}\end{aligned}$$

- *pure - sets*

$$\begin{aligned}&(\_)^{\vec{\circ}} : \vec{R^n} \to \vec{R^n}\\ &r^{\vec{\circ}} = \{(x_n, \ldots, x_1) \mapsto X \mid r\ (x_1, \ldots, x_n) = X\}\end{aligned}$$

Next, we have the *variational union* operation[2].

- *pure - formulas*

$$\begin{aligned}
\vec{\cup} &: \vec{R^n} \to \vec{R^n} \to \vec{R^n} \\
r \,\vec{\cup}\, s &= \{t \mapsto \phi \vee \psi \mid r\, t = \phi \wedge s\, t = \psi\} \\
&\cup \{t \mapsto \phi \mid r\, t = \phi \wedge s\, t = \bot\} \\
&\cup \{t \mapsto \psi \mid r\, t = \bot \wedge s\, t = \psi\}
\end{aligned}$$

- *pure - sets*

$$\begin{aligned}
\vec{\cup} &: \vec{R^n} \to \vec{R^n} \to \vec{R^n} \\
r \,\vec{\cup}\, s &= \{t \mapsto X \cup Y \mid r\, t = X \wedge s\, t = Y\} \\
&\cup \{t \mapsto X \mid r\, t = X \wedge s\, t = \bot\} \\
&\cup \{t \mapsto Y \mid r\, t = \bot \wedge s\, t = Y\}
\end{aligned}$$

Next, we have the *variational intersection* operation.

- *pure - formulas*

$$\begin{aligned}
\vec{\cap} &: \vec{R^n} \to \vec{R^n} \to \vec{R^n} \\
r \,\vec{\cap}\, s &= \{t \mapsto \phi \wedge \psi \mid r\, t = \phi \wedge s\, t = \psi\}
\end{aligned}$$

- *pure - sets*

$$\begin{aligned}
\vec{\cap} &: \vec{R^n} \to \vec{R^n} \to \vec{R^n} \\
r \,\vec{\cap}\, s &= \{t \mapsto X \cap Y \mid r\, t = X \wedge s\, t = Y\}
\end{aligned}$$

Next, we have the *variational difference* operation.

- *pure - formulas*

$$\begin{aligned}
\vec{\setminus} &: \vec{R^n} \to \vec{R^n} \to \vec{R^n} \\
r \,\vec{\setminus}\, s &= \{t \mapsto \phi \wedge \neg\psi \mid r\, t = \phi \wedge s\, t = \psi\} \\
&\cup \{t \mapsto \phi \mid r\, t = \phi \wedge s\, t = \bot\}
\end{aligned}$$

[2] $s\ t = \bot$ means that tuple t is not present in relation s

- *pure - sets*

$$
\begin{aligned}
\vec{\setminus} &: \vec{R^n} \rightarrow \vec{R^n} \rightarrow \vec{R^n} \\
r \mathbin{\vec{\setminus}} s &= \{t \mapsto X \cap \overline{Y} \mid r\, t = X \wedge s\, t = Y\} \\
&\quad \cup \{t \mapsto X \mid r\, t = X \wedge s\, t = \bot\}
\end{aligned}
$$

Lastly, we have the *variational composition* operation.

- *pure - formulas*

$$
\begin{aligned}
\vec{\cdot} &: \vec{R^n} \rightarrow \vec{R^m} \rightarrow R^{n \vec{+} m-2} \\
r \mathbin{\vec{\cdot}} s &= \{(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) \mapsto \phi \wedge \psi \\
&\qquad \mid r\,(x_1, \ldots, x_n) = \phi \wedge s\,(y_1, \ldots, y_m) = \psi \wedge x_n = y_1\}
\end{aligned}
$$

- *pure - sets*

$$
\begin{aligned}
\vec{\cdot} &: \vec{R^n} \rightarrow \vec{R^m} \rightarrow R^{n \vec{+} m-2} \\
r \mathbin{\vec{\cdot}} s &= \{(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) \mapsto X \cap Y \\
&\qquad \mid r\,(x_1, \ldots, x_n) = X \wedge s\,(y_1, \ldots, y_m) = Y \wedge x_n = y_1\}
\end{aligned}
$$

With these operations, we can define our execution function[3]:

$$\mathsf{run}_p\, d\, (\sim R) = (\mathsf{run}_p\, d\, R)^{\vec{\circ}} \tag{21}$$

$$\mathsf{run}_p\, d\, (R_1 + R_2) = (\mathsf{run}_p\, d\, R_1) \mathbin{\vec{\cup}} (\mathsf{run}_p\, d\, R_2) \tag{22}$$

$$\mathsf{run}_p\, d\, (R_1 \,\&\, R_2) = (\mathsf{run}_p\, d\, R_1) \mathbin{\vec{\cap}} (\mathsf{run}_p\, d\, R_2) \tag{23}$$

$$\mathsf{run}_p\, d\, (R_1 - R_2) = (\mathsf{run}_p\, d\, R_1) \mathbin{\vec{\setminus}} (\mathsf{run}_p\, d\, R_2) \tag{24}$$

$$\mathsf{run}_p\, d\, (R_1 \,.\, R_2) = (\mathsf{run}_p\, d\, R_1) \mathbin{\vec{\cdot}} (\mathsf{run}_p\, d\, R_2) \tag{25}$$

Now, only the boolean expressions are left. If you recall, the execution of these expressions must return a boolean formula.

First, we have the **some** expression. In this one, in the context of *pure - formulas* our formula is the disjunction of all the presence conditions of the relation of the expression. In *pure - sets*, its the union of all presence conditions.

- *pure - formulas*

$$\mathsf{run}_{fp}\, d\, (\mathbf{some}\ R) = \bigvee \{\phi \mid \exists t.\, (\mathsf{run}_{fp}\, d\, R)\, t = \phi\} \tag{26}$$

[3] run_p means that, depending on the context, it can be run_{fp} or run_{sp}

- *pure - sets*

$$\mathsf{run}_{sp}\ d\ (\mathbf{some}\ R) = \bigcup\{X \mid \exists t.\ (\mathsf{run}_{sp}\ d\ R)\ t = X\} \tag{27}$$

Next, we have the $\mathbf{no}$ expression, which is the opposite of the previous one.

- *pure - formulas*

$$\mathsf{run}_{fp}\ d\ (\mathbf{no}\ R) = \bigwedge\{\neg\phi \mid \exists t.\ (\mathsf{run}_{fp}\ d\ R)\ t = \phi\} \tag{28}$$

- *pure - sets*

$$\mathsf{run}_{sp}\ d\ (\mathbf{no}\ R) = \bigcap\{\overline{X} \mid \exists t.\ (\mathsf{run}_{sp}\ d\ R)\ t = X\} \tag{29}$$

Next, we have the $\mathbf{in}$ expression.

- *pure - formulas*

$$\begin{aligned}\mathsf{run}_{fp}\ d\ (R_1\ \mathbf{in}\ R_2) = \bigwedge\{\{\phi \Rightarrow \psi \mid \exists t.\ (\mathsf{run}_{fp}\ d\ R_1)\ t = \phi \wedge (\mathsf{run}_{fp}\ d\ R_2)\ t = \psi\} \\ \vee\{\neg\phi \mid \exists t.\ (\mathsf{run}_{fp}\ d\ R_1)\ t = \phi \wedge (\mathsf{run}_{fp}\ d\ R_2)\ t = \bot\}\}\end{aligned} \tag{30}$$

- *pure - sets*

$$\begin{aligned}\mathsf{run}_{sp}\ d\ (R_1\ \mathbf{in}\ R_2) = \bigcap\{\{\overline{X} \cup Y \mid \exists t.\ (\mathsf{run}_{sp}\ d\ R_1)\ t = X \wedge (\mathsf{run}_{sp}\ d\ R_2)\ t = Y\} \\ \cup\{\overline{X} \mid \exists t.\ (\mathsf{run}_{sp}\ d\ R_1)\ t = X \wedge (\mathsf{run}_{sp}\ d\ R_2)\ t = \bot\}\}\end{aligned} \tag{31}$$

Finally, we have the logical connectives.

$$\begin{aligned}\mathsf{run}_{p}\ d\ (\mathbf{not}\ R) &= \neg(\mathsf{run}_{p}\ d\ R) \\ \mathsf{run}_{p}\ d\ (R_1\ \mathbf{and}\ R_2) &= (\mathsf{run}_{p}\ d\ R_1) \wedge (\mathsf{run}_{p}\ d\ R_2) \\ \mathsf{run}_{p}\ d\ (R_1\ \mathbf{or}\ R_2) &= (\mathsf{run}_{p}\ d\ R_1) \vee (\mathsf{run}_{p}\ d\ R_2)\end{aligned}$$

Let's look at an example execution in the CT variational database, with the *pure - formulas* variational algorithm:

$$\begin{aligned}
& \mathsf{run}_{fp}\ d\ (\mathbf{some}\ \mathsf{Node} + \mathsf{Topic}) \\
\equiv\ & \ (26) \\
& \bigvee\{\phi \mid \exists t.\ (\mathsf{run}_{fp}\ d\ (\mathsf{Node} + \mathsf{Topic}))\ t = \phi\} \\
\equiv\ & \ (23) \\
& \bigvee\{\phi \mid \exists t.\ ((\mathsf{run}_{fp}\ d\ \mathsf{Node})\ \vec{\cup}\ (\mathsf{run}_{fp}\ d\ \mathsf{Topic}))\ t = \phi\} \\
\equiv\ & \ (19)\ \times 2 \\
& \bigvee\{\phi \mid \exists t.\ (\{\mathsf{CM} \mapsto C, \mathsf{RM} \mapsto R, \mathsf{S} \mapsto S, \mathsf{MP}_1 \mapsto (C \land R) \lor (C \land S) \lor (R \land S), \\
& \qquad \mathsf{MP}_2 \mapsto (C \land R \land S), \mathsf{SIM} \mapsto \mathbf{true}\}\ \vec{\cup}\ \{\mathsf{vel}_{\mathsf{CM}} \mapsto C, \mathsf{vel}_{\mathsf{RM}} \mapsto R, \mathsf{vel}_{\mathsf{S}} \mapsto S, \\
& \qquad \mathsf{vel}_{\mathsf{MP}_1} \mapsto (C \land R) \lor (C \land S) \lor (R \land S), \mathsf{vel}_{\mathsf{MP}_2} \mapsto (C \land R \land S), \mathsf{pose} \mapsto S\})\ t = \phi\} \\
\equiv\ & \ \textit{def. } \vec{\cup} \\
& \bigvee\{\phi \mid \exists t.\ (\{\mathsf{CM} \mapsto C, \mathsf{RM} \mapsto R, \mathsf{S} \mapsto S, \mathsf{MP}_1 \mapsto (C \land R) \lor (C \land S) \lor (R \land S), \\
& \qquad \mathsf{MP}_2 \mapsto (C \land R \land S), \mathsf{SIM} \mapsto \mathbf{true}, \mathsf{vel}_{\mathsf{CM}} \mapsto C, \mathsf{vel}_{\mathsf{RM}} \mapsto R, \mathsf{vel}_{\mathsf{S}} \mapsto S, \\
& \qquad \mathsf{vel}_{\mathsf{MP}_1} \mapsto (C \land R) \lor (C \land S) \lor (R \land S), \mathsf{vel}_{\mathsf{MP}_2} \mapsto (C \land R \land S), \mathsf{pose} \mapsto S\})\ t = \phi\} \\
\equiv\ & \ \textit{trivial} \\
& \bigvee\{C, R, S, (C \land R) \lor (C \land S) \lor (R \land S), C \land R \land S, \mathbf{true}\} \\
\equiv\ & \ \textit{def. } \bigvee \\
& C \lor R \lor S \lor (C \land R) \lor (C \land S) \lor (R \land S) \lor (C \land R \land S) \lor \mathbf{true} \\
\equiv\ & \ \textit{trivial} \\
& \mathbf{true}
\end{aligned}$$

This means that the property is valid in all configurations.

6

IMPLEMENTATION

In the previous chapters we introduced formal concepts to define and analyze variational databases. In this chapter, we put into to practice these concepts, creating a small framework that will allow us to test and evaluate the different approaches for checking properties on variational databases.

For this implementation, we used the Python language.

6.1 SIMPLE RELATIONS AND DATABASES

Let’s start with the simple concepts first.

6.1.1 *Simple Relations*

Just like before, we define a simple relation as a *set* of *tuples* with arbitrary size:

```
Relation = Set[Tuple[Any, ...]]
```

For instance,

```
Node = {(’n1’,), (’n2’,), (’n3’,)}
Topic = {(’t’,)}
subscribes = {(’n3’, ’t’)}
publishes = {(’n1’, ’t’), (’n2’, ’t’)}
```

are all examples of defined simple relations.

6.1.2 *Simple Databases*

A simple database is defined as a *dict*, with `str` values as keys and `Relation` values as values.

```
Database = Dict[Str, Relation]
```

With this, we can define a database, as it is shown in the following example:

```
db = {
    "Node" : {("n1",), ("n2",), ("n3",)},
    "Topic" : {("t",)},
    "subscribes" : {("n3", "t")},
    "publishes" :{("n1", "t"), ("n2", t)}
}
```

6.2 VARIATIONAL

Next, we have the variational concepts.

6.2.1 *Presence Conditions*

Let's start with presence conditions. Earlier, we defined presence conditions as formulas that represent the configurations in which a certain element is present. To represent these formulas, we use the Python Z3 library, which is a Python middleware to work with the high performance theorem prover Z3. For example,

$$\phi = A \wedge (\neg B \vee C)$$

can be defined as:

```
from z3 import *

// create the variables
A = Bool('A')
B = Bool('B')
C = Bool('C')

phi = And(A, Or(Not(B), C))
```

The Z3 library, besides having all these classes to represent formulas, also provides us with solving algorithms and formula simplification.

6.2.2 *Variational Relations*

A variational relation with formulas is defined as a *dict* with *tuples* as keys and *presence conditions* as values:

```
FVRelation = Dict[Tuple[Any, ...], BoolRef]
```

where `BoolRef` is the type of a formula in Z3, and it is used to represent a presence condition.

On the other hand, variational relations with sets of configurations are a *dict* with *tuples* as keys and *sets of configurations* as values:

```
SVRelation = Dict[Tuple[Any, ...], Set[SConfiguration]]
```

where

```
Configuration = Dict[BoolRef, Bool]
SConfiguration = FrozenSet[BoolRef]
```

Here are some examples of variational relations:

```
// With formulas
FNode = { (’n1’,): A, (’n2’,): B, (’n3’,): C }
FTopic = { (’t’,) : Or(A,B) }
Fsubscribes = { (’n3’,’t’): And(Or(A,B),C) }

// With sets
SNode = {
    (’n1’,): {{A},{A,B},{A,C},{A,B,C}},
    (’n2’,): {{B},{A,B},{B,C},{A,B,C}},
    (’n3’,): {{C},{A,C},{B,C},{A,B,C}}
}
...
```

6.2.3 *Variational Databases*

A variational database is defined like a simple database, only instead uses variational relations as values.

```
FVDatabase = Dict[Str, FVRelation]
SVDatabase = Dict[Str, SVRelation]
```

6.3 QUERY LANGUAGE AND ALGORITHMS

The query language was implemented using the python PLY library.

This is the implemented language grammar:

```
query -> compose1
compose1 -> compose1 OR compose2
compose1 -> compose2
compose2 -> compose2 AND atomic
compose2 -> atomic
atomic -> SOME expression
atomic -> NO expression
atomic -> expression IN expression
atomic -> NOT atomic
atomic -> ( compose1 )
expression -> expression + term
expression -> expression & term
expression -> expression - term
expression -> term
term -> term . term1
term -> term1
term1 -> ID
term1 -> ~ term1
term1 -> ( expression )
```

7

ANALYSIS AND EVALUATION OF THE VARIATIONAL ALGORITHMS

In chapter 5, we presented several algorithms for executing queries in variational databases. In this chapter, we make an evaluation of said algorithms, evaluating execution time and space, in order to try to find the most efficient one.

7.1 VARIATIONAL DATABASE GENERATION

In order to evaluate the algorithms, we first need to have some sample data. Until now, we have used the *Controlled Turtlesim* system as a “playground” to run and test the different algorithms for query execution. However, if we were to measure these executions in terms of time and space, we would see that the differences in the measurements would not be significant, as this system is just too small. Taking a real world example with enough size and model it with our variational databases would work, however, our evaluation would be constricted to that system and its characteristics. We do not know if what was good and efficient in this system, would also be in others, as different systems have different characteristics. So, we have decided to create a framework that allows us to generate custom synthetic databases that, although do not represent any real world system, contain information large enough for us to perform the evaluations.

In this framework, the generation of a database consists of two steps:

1. *Generation of a SPL architecture according to a set of configurations*.
   Specifying a set of configurations will allow us to generate a SPL architecture with specific characteristics.

2. *Conversion of the generated SPL architecture to each one of the variational databases types*.
   After the SPL architecture is generated, we want to convert it to the different variational database types. This way, we ensure that the information in the databases is consistent, i.e. all databases faithfully represent the same SPL architecture.

For efficiency reasons, in Step 1 we will represent the SPL architecture already in the format of the *Pure - Formula* variational database. We will denote this first database as $\mathcal{O}$. Then, in Step 2 we will convert it to the other two types of variational databases.

For generating $\mathcal{O}$, we have a couple of approaches: random generation and controlled generation.

7.1.1 *Random Generation*

First, we have the random generator. As the name implies, in this one we generate a random variational database, composed by variational sets and variational binary relations between the elements of the sets.

With this approach, we have two levels of configurations. The first one is on a structural level. If we strip the elements of a variational database of their presence conditions, we get a collection of non-variational sets and binary relations. This collection can be customized by the following variables:

- S - the variational sets. For example, $S = \{\text{Node}, \text{Topic}\}$.
- N_S - number of elements per set.
- R - the variational binary relations. For example, $R = \{\text{subscribes} : \text{Node} \rightarrow \text{Topic}, \text{publishes} : \text{Node} \rightarrow \text{Topic}\}$.
- $p : R \rightarrow [0, 1]$ - presence probability of a tuple from a relation R.

With this configurations, and based on the Erdos-Rényi model for generating random graphs, we generate the contents of the database with the following algorithms:

```
generate_sets(S, N_S):
    result = {}
    for s in S:
        result[s] = {}

        for i in range(N_S):
            element_name = s.lower() + str(i)
            element_pc = generate_PC(F, p_F, p_D)

            result[s][(element_name,)] = element_pc

    return result

generate_relations(R, p):
    result = {}
    for r in R:
        result[r.name] = {}
        for x in r.dom: # relation domain
            for y in r.cod: #relation codomain
                if rand(0,1) <= p(r):
                    result[r.name][(x,y)] = generate_PC(F, p_F, p_D)

    return result
```

The second level of configuration consists on the generation of presence conditions (function `generate_PC` invoked in the code above). We want to be able to customize this generation, in order to control the specificity of each presence condition, and also the distribution of features present in each presence condition. The presence conditions will be written in DNF (Disjunctive Normal Form) and, for generating them, we will use the following variables:

- F - the set of features. In this case we assume our $\mathcal{FM} = \mathbf{true}$, so there are $2^{|F|}$ possible configurations, and our set of configurations $\mathcal{C} = \mathcal{P}(F)$.
- p_F - the probability of a feature being present in a conjunction.
- p_D - the probability of a new disjunction.

A DNF formula is a disjunction of conjunctions. In this generation it is intended that the number of conjunctions follows an exponential distribution that depends on p_D. This way, we ensure that we will never have too many conjunctions in the presence condition. Also, a presence condition has, at least, one conjunction. This is because a disjunction with no conjunctions is a formula that evaluates to $\mathbf{false}$. And if a presence condition is $\mathbf{false}$, that means that the tuple to which it is associated does not exist in the database, which makes no sense.

On the other hand, the number of features present in each conjunction should follow a binomial distribution that depends on p_F.

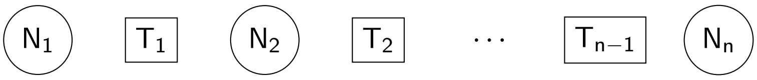
Having said that, the algorithm for generating a presence condition is the following:

```
generate_PC (F, p_F, p_D):
    result = generate_conjunction(F, p_F)
    while rand(0,1) <= p_D:
        result = result \/ generate_conjunction(F, p_F)

    return result


generate_conjunction (F, p_F):
    result = true
    for f in F:
        if rand(0,1) <= p_F:
            if rand(0,1) <= 0.5:
                result = result /\ f
            else:
                result = result /\ not f

    return result
```

7.1.2 *Controlled Generation*

Although the random generator is capable of generating almost an unlimited amount of variational databases, its generation is also too unpredictable, i.e. when we execute a query, we do not know which results we are going to get. With this in mind, we decided to create a different generator, that will let us create a special type of database in which we do know what results to expect.

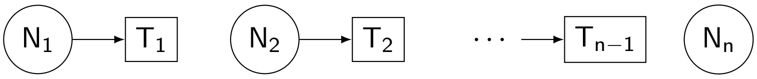
The generated databases will emulate a computation graph, consisting in a sequence of nodes and topics. In each generation we pass a value $\mathsf{n} > 0$ that indicates how many nodes there are in the computation graph. The number of topics shall be $\mathsf{n} - 1$.



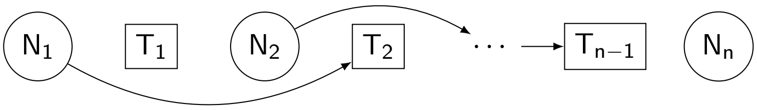
In this database, the features decide which nodes are subscribed/publish to which topics. Features of type P_x control the $\mathsf{publishes}$ relation, while features of type S_x control the $\mathsf{subscribes}$ relation, according to the following rules:

$$\forall\, 1 \leq x < \mathsf{n}.\ P_x \text{ is active} \Rightarrow \forall\, 1 \leq i \leq \mathsf{n} - x.\ (\mathsf{N_i}, \mathsf{T}_{i+x-1}) \in \mathsf{publishes}$$
$$\forall\, 1 \leq x < \mathsf{n}.\ S_x \text{ is active} \Rightarrow \forall\, 1 + x \leq i \leq \mathsf{n}.\ (\mathsf{N_i}, \mathsf{T}_{i-x}) \in \mathsf{subscribes}$$

If feature P_1 is active, this means that each node will publish to the topic directly after it:

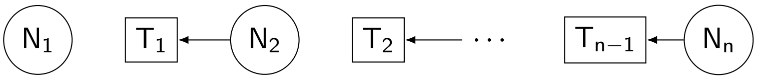


If P_2 is active, then each node will publish to the second topic after it:



And likewise for the remaining features of type P_x.

The S_n features have identical behavior. If S_1 is active, then each node is subscribed to the topic directly before it:



If S_2 is active, then each node is subscribed to the topic two steps before, and likewise for the remaining features of type S_x.

According to these definitions, for a given n, there are a total of $2 * (\mathsf{n} - 1)$ features, that equal to the following set $F = \{S_x \mid 1 \leq x < \mathsf{n}\} \cup \{P_x \mid 1 \leq x < \mathsf{n}\}$. Once again, we assume that the $\mathcal{FM}$ that is represented in this generations equals to $\mathbf{true}$, so the number of configurations of our generated database will be $2^{(2*(\mathsf{n}-1))}$. The set of configurations is defined by $\mathcal{C} = \mathcal{P}(F)$.

7.1.3 *Conversion*

Now that $\mathcal{O}$ is generated, we can convert it to the other types of databases.

Conversion to Non-variational

First, we have the *non-variational* variational database. If we recall, this variational database is a collection of simple databases, where each one has a configuration associated.

This conversion is achieved by projecting $\mathcal{O}$ for each configuration, creating in each projection a simple database, and then associating this simple database with the configuration that projected it.

$$\mathcal{O}_{nv} = \{c \mapsto d \mid d = \mathcal{O}|_c \wedge c \in \mathcal{C}\}$$

Conversion to Pure Variational - Sets

To convert to the *pure variational - sets* database, we just need to solve each presence condition in each variational relation.

$$\mathcal{O}_{ps} = \{id \mapsto sr \mid id \mapsto r \in \mathcal{O} \wedge sr = \{t \mapsto \mathsf{solve}_{\mathcal{FM}}\ \phi \mid t \mapsto \phi \in r\}\}$$

7.2 EVALUATION

In this section, we first present the settings for performing the comparison of the different querying algorithms for variational databases, and then present and discuss the results of said comparison.

7.2.1 *Evaluation settings*

In order to do a thorough comparison, we want to run the various algorithms taking into account different settings and which type of database was generated.

Settings for the random generated databases

In the random generated databases, we have a set of settings that are variable and will serve as metrics in the evaluations, and another set of settings that are fixed and will be common to all evaluations.

- Variable settings:

  - *number of features* - number of features of the database (depends on F);
  - *size of the database* - number of elements per set (depends on N_S);
  - *complexity of the presence conditions* - probability of each presence condition have several disjunctions (depends on p_D);

- Fixed settings:
  - $S = \{\text{Node}, \text{Topic}\}$
  - $R = \{\text{subscribes} : \text{Node} \rightarrow \text{Topic}\}$
  - $p(\text{subscribes}) = 0.3$
  - $p_F = 0.3$

As for the controlled generated databases, all settings are variable:

- *number of features* - number of features (depends on S_x and P_x);
- *size of the database* - number of elements per set (depends on n);

For each setting we will measure the performance time and memory consumption of running each algorithm for querying the variational databases with the following query:

$$\textbf{no } \text{Node.subscribes} - \text{Topic}$$

Which specifies the following property: "There can not be subscriptions to topics that do not exist."

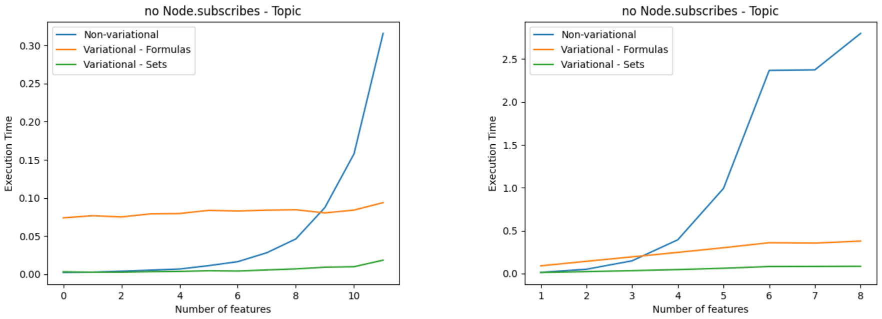
7.2.2 *Results and Discussion*

PERFORMANCE

- **Number of features** - Figures 21a and 21b shows the evolution of performance times of the execution of the different algorithms as the number of features of an SPL increases.

  In the random generation, the values of the variable settings were: $N_S = 50$ and $p_D = 0.3$. In the controlled generation, the values of the variable settings were: $\text{n} = 500$.

  As we can see, the *purely variational* approaches have more or less the same execution time throughout the analyses, with the *pure - sets* being the fastest. However, the *non-variational* algorithm, despite starting very quickly, has an exponential growth in terms of execution time as the number of features increases. This is to be expected because as the number of features increases, so does the number of possible configurations of the SPL, and consequently, the number of simple databases inside the *non-variational* variational database.



(a) Random database.

(b) Controlled database.

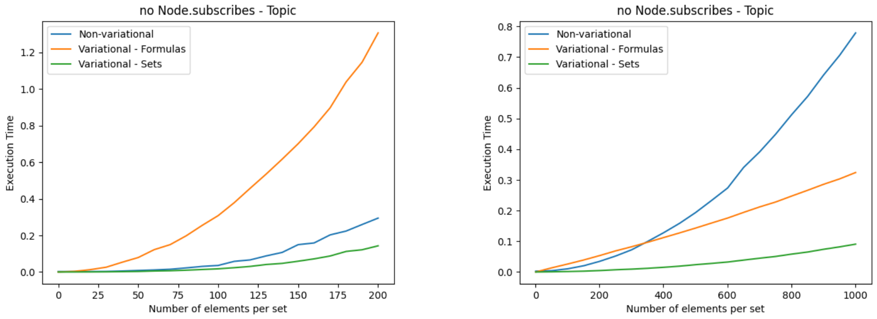
Figure 21: Number of features.

- **Size of the database** - In FigureS 22a and 22b, we can see the evolution of the execution time as the number of elements per set of the variational databases increases.

  In the random generation, the values of the variable settings were: $|F| = 4$ and $p_D = 0.3$. In the controlled generation, the values of the variable settings were: $S = \{S_1, S_2\}$ and $P = \{P_1, P_2\}$.

  In the random generation, all the curves appear to have more or less the same growth rate, with the *pure - formulas* algorithm being the slowest.

  In the controlled generation, the roles swap a little. Maybe because of the size of the database, we can see that with time, the *non-variational* algorithm becomes slower than the *pure - formulas* algorithm. Once again, the *pure - sets* approach is the fastest.



(a) Random database.

(b) Controlled database.

Figure 22: Number of elements per set.

- **Complexity of presence conditions** - Figure 23 shows the execution times of the algorithms in a random database when varying the probability of a new disjunction in a presence condition.

  In this generation, the variable settings were: $|F| = 6$ and $N_S = 100$.

We can see that, once again, the *pure - sets* approach takes the win in terms of efficiency. The *pure - formulas* approach is approximately constant. However, the *non-variational* approach execution times seem to increase exponentially as the probability of a new disjunction increases.

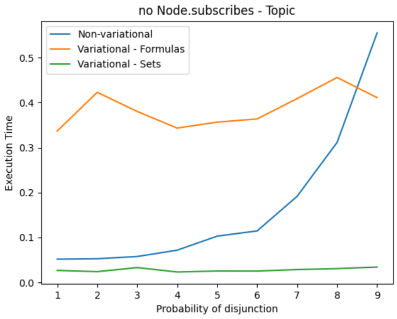
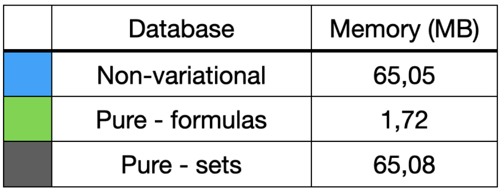


Figure 23: Plot showing the performance times of the different algorithms, varying the probability of a new disjunction in a presence condition.

MEMORY CONSUMPTION In terms of memory, the better approach is definitely the *pure - formulas*. In small databases, the difference is insignificant. However, if we start to grow our databases, we see that the *pure - formulas* variational database is significantly smaller than the others.

Let's look at two examples. In both of them, we measure the size of the Python objects that represent the variational databases at run-time. The first is a controlled generation, with 8 features and 1000 elements per set. In this generation, the sizes of the variational databases were:



	Database	Memory (MB)
🟦	Non-variational	65,05
🟩	Pure - formulas	1,72
⬛	Pure - sets	65,08

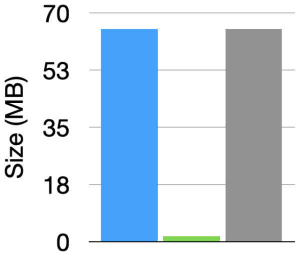


Figure 24: Memory consumption (MB) on controlled generation.

Here, the *non-variational* and the *pure - sets* variational databases are both about 37.8 times bigger than the *pure - formulas* variational database.

In another example, a random generated one, the sizes of the variational databases were the following:

	Database	Memory (MB)
🟦	Non-variational	34,02
🟩	Pure - formulas	5,74
⬛	Pure - sets	26,75

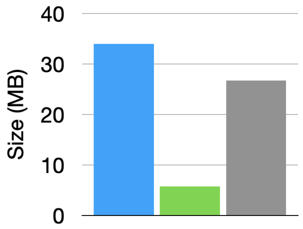


Figure 25: Memory consumption (MB) on random generation.

Which means that the *non-variational* and the *pure - sets* variational databases were, respectively, about 5.9 and 4.6 times bigger than the *pure - formulas* variational database.

In both cases we can see that the *pure - formulas* approach is the better approach in terms of space.

Overall, the *pure - sets* approach seems to be the fastest. Comparing to the *pure - formulas* approach, this difference must be related to the fact that, as calculations are carried out, the *pure -formulas* approach always tries simplifies its presence conditions. And formula simplification is not an easy process. However, this same simplification is also the responsible for the small space that the *pure - formulas* variational database occupies.

8

CONCLUSION

ROS is the most popular framework for developing robotic applications. A ROS system usually has several variants for supporting different configurations of a robot. The set of these configurations is called a *Software Product Line*, or SPL. HAROS is a static analysis framework used to analyse ROS systems. HAROS has a query engine that allows the specification and execution of architectural queries. Currently, this engine can only analyse one configuration at a time, which means that it will be unfeasible to analyse the whole SPL defined by a ROS application, in particular if it has a large number of features.

In this thesis, we presented three different approaches capable of analysing an SPL as a whole, instead of configuration by configuration. These approaches are: *non-variational*, *pure - formulas* and *pure - sets*. For each approach, we defined both a variational data structure (that we called *variational database*) for holding the data of the SPL, and a variational algorithm capable of executing queries on said data structure. We also created a small language, inspired by the Alloy specification language, that makes use of relations and first-order logic to specify the queries.

To compare the different approaches, we performed several analyses on synthetic databases with different characteristics, in order to measure the algorithms in terms of performance and memory consumption.

This experimental evaluation shows that overall, performance-wise, the *pure - sets* approach presents the better results. Between the *non-variational* and the *pure - formulas* approaches, it depends on the characteristics of the databases. The *non-variational* approach seems to be faster on almost every analysis that we made, except when the number of features is high and the presence conditions are more complex.

However, when we are talking about memory, then the *pure - formulas* is definitely the better option.

Another good advantage of the *pure - formulas* approach is that the query results are also formulas, which may simplify the interpretation of said results. For example, if a query returns $A \vee B$, we know that that query is valid in all configurations in which the features A or B are active. In the other approaches, the result would be a list of all configurations that have A or B active, making it harder to reach the same conclusion.

Future works concerns deeper analysis and implementations. An obvious one would be implementing this in the HAROS framework. However that would imply the support of a more complex language than the one we have. Other idea would be to try making a more detailed evaluation, with more databases, and see if the obtained results would be similar.

BIBLIOGRAPHY

Stephan Adelsberger, Stefan Sobernig, and Gustaf Neumann. Towards assessing the complexity of object migration in dynamic, feature-oriented software product lines. pages 17:1–17:8, 01 2013. ISBN 9781450325561. doi: 10.1145/2556624.2556645.

Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kastner. Detecting dependences and interactions in feature-oriented design. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*, pages 161–170, 2010. doi: 10.1109/ISSRE.2010.11.

Don Batory. Feature models, grammars, and propositional formulas. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 7–20, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32064-7.

Don Batory, J.N. Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *Software Engineering, IEEE Transactions on*, 30:355 – 371, 07 2004. doi: 10.1109/TSE.2004.23.

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Advanced Information Systems Engineering*, pages 491–503, Berlin, Heidelberg, 2005a. Springer Berlin Heidelberg.

David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using constraint programming to reason on feature models. In *IN THE SEVENTEENTH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING*, 2005b.

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35:615–636, 09 2010. doi: 10.1016/j.is.2010.01.001.

Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. pages 3–8, 07 2012. doi: 10.1145/2259051.2259052.

Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. 03 2012. doi: 10.1145/2162049.2162052.

Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, AOSD '02, page 10–18, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113469X. doi: 10.1145/508386.508389. URL https://doi.org/10.1145/508386.508389.

Andreas Classen, Quentin Boucher, and Patrick Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Science of Computer Programming, Special Issue on Software Evolution*, 76, 01 2010. doi: 10.1016/j.scico.2010.10.005.

Philippe Collet. Domain specific languages for managing feature models: Advances and challenges. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, pages 273–288, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-45234-9.

David Coppit, Robert Painter, and Meghan Revelle. Spotlight: A prototype tool for software plans. pages 754–757, 06 2007. ISBN 0-7695-2828-7. doi: 10.1109/ICSE.2007.79.

Maxime Cordy, Andreas Classen, Pierre Yves Schobbens, Patrick Heymans, and Axel Legay. Managing evolution in software product lines : A model-checking perspective. pages 183–191, 01 2012. doi: 10.1145/2110147.2110168.

Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10:7–29, 01 2005. doi: 10.1002/spip.213.

Arie Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10, 01 2002. doi: 10.2498/cit.2002.01.01.

Martin Erwig, Eric Walkingshaw, and Sheng Chen. An abstract representation of variational graphs. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, FOSD '13, page 25–32, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450321686. doi: 10.1145/2528265.2528270. URL https://doi.org/10.1145/2528265.2528270.

S. García, Daniel Strüber, D. Brugali, Alessandro Di Fava, Philipp Schillinger, P. Pelliccione, and T. Berger. Variability modeling of service robots: Experiences and challenges. *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*, 2019.

Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990. URL http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 311–320, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580791. doi: 10.1145/1368088.1368131. URL https://doi.org/10.1145/1368088.1368131.

Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. volume 3586, pages 195–213, 07 2005. ISBN 978-3-540-27992-1. doi: 10.1007/11531142_9.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. 09 2001. ISBN 978-3-540-42206-8. doi: 10.1007/3-540-45337-7_18.

Christian Kästner and Sven Apel. Integrating compositional and annotative approaches for product line engineering. 10 2008.

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. pages 81–91, 08 2013. doi: 10.1145/2491411.2491437.

Chong Liu, Nuno Macedo, and Alcino Cunha. Simplifying the analysis of software design variants with a colorful alloy. In Nan Guan, Joost-Pieter Katoen, and Jun Sun, editors, *Dependable Software Engineering. Theories, Tools, and Applications*, pages 38–55, Cham, 2019. Springer International Publishing. ISBN 978-3-030-35540-1.

Mike Mannion. Using first-order logic for product line model validation. pages 176–187, 08 2002. ISBN 978-3-540-43985-1. doi: 10.1007/3-540-45652-X_11.

Jan Midtgaard, Aleksandar Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Science of Computer Programming*, 105, 04 2015. doi: 10.1016/j.scico.2015.04.005.

Christian Prehofer. Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13:465–501, 05 2001. doi: 10.1002/cpe.583.

M. Quigley. ROS: an open-source robot operating system. In *ICRA 2009*, 2009.

John Reppy and Aaron Turon. Metaprogramming with traits. 07 2007. ISBN 978-3-540-73588-5. doi: 10.1007/978-3-540-73589-2_18.

Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent feature modularization. pages 11–18, 01 2010. doi: 10.1145/1869542.1869545.

Hamideh Sabouri and Ramtin Khosravi. Reducing the verification cost of evolving product families using static analysis techniques. *Sci. Comput. Program.*, 83:35–55, April 2014. ISSN 0167-6423. doi: 10.1016/j.scico.2013.06.009. URL https://doi.org/10.1016/j.scico.2013.06.009.

A. Santos, A. Cunha, N. Macedo, and C. Lourenço. A framework for quality assessment of ros repositories. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4491–4496, 2016. doi: 10.1109/IROS.2016.7759661.

A. Santos, A. Cunha, and N. Macedo. Static-time extraction and analysis of the ROS computation graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*, pages 62–69, 2019. doi: 10.1109/IRC.2019.00018.

Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. volume 35, page 705–754, USA, July 2005. John Wiley &amp; Sons, Inc.

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time - configurable system software: Facing the linux 10,000 feature problem. pages 47–60, 01 2011. doi: 10.1145/1966445.1966451.

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. Analysis strategies for software product lines: A classification and survey. 03 2015.

L. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583).