Vrije Universiteit Brussel

FACULTY OF SCIENCE AND BIO-ENGINEERING SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Rule Sharing for the Context Modelling Toolkit

Master thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Lars Van Holsbeeke

Promoter: Prof. Dr. Beat Signer
Advisor: Sandra Trullemans

Academic year 2015-2016

Vrije Universiteit Brussel

FACULTEIT WETENSCHAPPEN EN BIO-INGENIEURSWETENSCHAPPEN
VAKGROEP COMPUTERWETENSCHAPPEN

Rule Sharing for the Context Modelling Toolkit

Masterproef ingediend in gedeeltelijke vervulling van de eisen voor het behalen van de graad Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Lars Van Holsbeeke

Promotor: Prof. Dr. Beat Signer
Begeleider: Sandra Trullemans

Academiejaar 2015-2016

Abstract

About a decade ago, a revolution in mobile hardware unleashed itself. Nowadays, we all have a smartphone or tablet with various integrated sensors. However the revolution is not limited to mobile devices alone. Everyday physical objects surrounding us are connecting to each other and to the Internet in an immense grid: The Internet Of Things. Having such a large grid of sensors available, a device can collect a lots of information about its environment – the context in which it resides. Context awareness used to be limited to location awareness. However, context is a lot more than location only. Thanks to the increasing number of sensor types getting coming online everyday, the potential of doing something with that context is increasing rapidly. As such, a lot of tools for helping a user to use that context for their own needs exist. Multiple of these context modelling tools are discussed, each with their proper advantages and disadvantages. As a result of this discussion, a trade-off between the complexity of context rules and the required user skills will be identified in all of these frameworks. In order to tackle this problem, the Context Modelling Toolkit (CMT) has been developed at the WISE lab of the Vrije Universiteit Brussel. This framework allows users with different levels of expertise to model context for their own needs. Unfortunately, each user has their own system, which means that in case no expert user is available, low expertise users will still experience difficulties when trying to model complexer rules. That is the exact problem this thesis is addressing. By extending CMT with functionality to share rules, low expertise users can import more complex context rules from expert users, giving the former access to more complex features in context-awareness. Yet, sharing rules is not trivial, as each user has their own specific environment with its specific sensors. As such, context rules that have been created for one environment are not always compatible in another environment. This problem is solved by involving the user in the import process. To make this process as smooth as possible, intelligibility is of paramount importance.

In order to facilitate the above, this thesis makes three contributions. First, a study concerning different rule-sharing paradigms is conducted. Only one of these paradigms is eligible to use in combination with the design constraints imposed by CMT. Additionally, a string matching algorithm to match different types of building blocks of a context rule is needed. Therefore, we investigated multiple (fuzzy) string matching algorithms in order to identify the most appropriate one. Finally, the implementation of the extension and its connection with CMT is discussed.

Declaration of Originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited. I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Acknowledgements

In the first place, I would like to thank my promotor Prof. Dr. Beat Signer to provide me with an interesting thesis topic in the domain of home automation, as well as to give me the opportunity to write this thesis, physically, at the WISE lab and providing me with detailed feedback. I would also like to thank my advisor, Sandra Trullemans, who has, especially during the first months, been of great help to understand the Context Modelling Toolkit framework. I would also like to thank the other two assistants at the WISE lab, Reinout Roels and Audrey Sanctorum to help me through the difficult and sometimes frustrating moments I experienced during the implementation phase. Next, I want to thank my Erasmus and lunch companion Tim Witters for his advice on crucial moments in my thesis process. Together with Tim, I would also like to thank Youri Coppens for the many after lunch Blur racing games the three of us played together. Additionally, I would like to thank my parents for their patience and my brother to clear my mind when needed. Finally, I am in great debt to my best friend, Anastasia, from the USA. I have never seen a document being corrected with that much colour and precision as my thesis.

Contents

1 Introduction
1.1 The Birth of Ubiquitous Computing . . . 1
1.2 Problem Statement . . . 6
1.3 Contributions . . . 7
1.4 Thesis Outline . . . 8

2 Background
2.1 Context-Awareness and Intelligibility . . . 11
2.2 Context Rule Sharing . . . 14
2.2.1 Motivation . . . 14
2.2.2 Context Modelling approaches . . . 15
2.2.3 Rule Creation techniques . . . 17
2.2.4 Expressiveness vs Programming Skill Trade-Off . . . 21
2.2.5 Conclusion: Sharing Rules for more expressiveness . . . 22

3 The Context Modelling Toolkit from a User Perspective
3.1 Introduction to CMT . . . 23
3.1.1 Motivation for CMT . . . 23
3.1.2 Rule-based context modelling in CMT . . . 24
3.1.3 Multi-layered context modelling . . . 34
3.2 Sharing Rules as a User . . . 36

4 Extending CMT to Share Rules
4.1 CMT Implementation: overview . . . 41
4.2 Architecture . . . 44
4.3 Rule Sharing Mechanisms . . . 48
4.3.1 Naive method . . . 48
4.3.2 Type Matching Method . . . 49
4.3.3 Name-matching method . . . 53
4.4 String Matching Algorithms . . . 56
4.4.1 Sorensen-Dice index . . . 56

4.4.2 Levenshtein Distance 59
4.4.3 Jaro-Winkler Distance 61
4.4.4 Smith-Waterman 62
4.4.5 Overview and Technical Evaluation 64
4.4.6 Synonyms and Program Flow 66
4.5 Use Case: Exporting and Importing a Rule 69
4.5.1 Exporting a custom event 70
4.5.2 Importing an event 81

5 Conclusion and Future Work
5.1 Discussion 99
5.2 Future Work 102
5.3 Conclusion 104

A Appendix: Implementation Code

1 Introduction

1.1 The Birth of Ubiquitous Computing

Back in September 1991, Mark Weiser published his world-famous article *The Computer for the 21st Century* [1]. As can be implied from the title, Weiser made a prediction about how the computer will look like in the 21st century, how people will handle this "computer" and how it will integrate in *the new electronic world*. Weisers' vision is directed to the fact that the computer in the 21st century will be an integral part of human life. Yet, he states that this vision does not is not about following aspects:

- *Portability* of computers, such as e.g. laptops. These machines still demand attention of the users using them as one still has to concentrate on one single 'box', instead of on the world.
- *Multimedia Systems*: these require even more attention from the user, rather than disappearing into the background
- *Virtual Reality*: only simulates a world, rather than changing the existing world.

With his vision, Weiser wanted to make clear that, in comparison to the nineties, a complete new way of thinking about the computer itself and its

position in human society is needed. Computers need to fade seamlessly into the background, become part of that background.

> *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."* - Mark Weiser [1]

Weiser called this *Ubiquitous Computing* and was therefore the founder of that term in 1988. "Ubiquitous Computing" itself means that the idea of interacting with a computer does not longer exists. Weiser compares it with how we are nowadays acting towards an ordinary piece of paper: completely unconsciously. Today, such devices already exist: think for example about a microwave, dishwasher, thermostat and stereo.

In order to accomplish the vision of having computers seamlessly fade away into the background, Weiser defined some criteria, involving both software and hardware. Considering hardware, he presented three types of computers (Tabs, Pads and Boards), with the main difference being the size of these devices. In the 1990s it was already possible to produce them. According to Moore's law, this should become even better over the years.

As good as the hardware might be(come), the software side of the vision is a whole different story. To make his vision come true, Weiser enumerated three major limiting software problems:

1. *Operating Systems*: Especially in the 1990s, operating systems assumed a relatively fixed configuration of hardware and software in the core. This restricts ubiquitous computing as devices come and go. Hence hardware is added and removed constantly and new devices need new software at any time. Nowadays, thanks to more flexible operating systems, this specific issue has been significantly reduced (e.g. USB).

2. *Window Systems*: These were in the beginning of the 1990s rather limited in their capabilities. In particular, they are quite bad in handling applications that move from one place, e.g. a computer screen, to another, e.g. a mobile phone screen. Because of performance reasons, these systems assume a fixed screen and input mode. Nowadays, this issue has mostly been solved too. We all know one or more applications, such as e.g. browsers and map applications, smoothly moving between multiple screens and devices. Also in the WISE lab, research is going on to find even better solutions to this problem [2].

3. *Network*: As two of the three devices Weiser proposes to reach ubiquitous computing are mobile (e.g. they constantly move from one place to another), the only acceptable way to connect these devices is via wireless links. Even these wireless links themselves need to be mobile. Think for example about two or more mobile devices getting into each other's range, they should be able to connect to each other without external infrastructure (e.g. WiFi direct [3]). However, even today, coordinating these connections is not trivial. Some attempts have successfully been made. For example, a programming language called AmbientTalk [4] has been developed especially to solve this problem. Yet, even after 25 years, interconnecting ubiquitous devices still is a hot research topic.

After elaborating on these problems, Weiser illustrates what he means with ubiquitous computing by presenting an extensive scenario on how it can affect our daily lives. He presents a day in the life of a professional single mother called Sal, giving the reader insight in how the world evolves around Sals' assumed needs. Devices such as computers, cameras and sensors seamlessly fade into her world, making her life smoother and more efficient. In the following we present a few snippets from the scenario about a day in Sal's life:

> *Sal awakens: she smells coffee. A few minutes ago her alarm clock, alerted by her restless rolling before waking, had quietly asked "coffee?", and she had mumbled "yes." "Yes" and "no" are the only words it knows.*
>
> *On the way to work Sal glances in the foreview mirror to check the traffic. She spots a slowdown ahead,...*
>
> *As she walks into the building the machines in her office prepare to log her in, but don't complete the sequence until she actually enters her office.*
>
> *The telltale by the door that **Sal programmed** her first day on the job is blinking: fresh coffee. She heads for the coffee machine.* [1]

In all of these snippets, one can clearly see what Weiser means with the notion of ubiquitous computing. When Sal wakes up, coffee will be ready for her. When she looks into the foreview mirror, she is informed about slowdowns ahead. When she walks into her office building, the login sequence

of her machine(s) is started. These are all precise concepts Weiser proposes himself to happen when a specific situation occurs. He does not mention any possibilities how users themselves could define what should happen in particular situations, i.e. depending on the context. A main reason for this is that the term (as well as the start of research on the topic) "context aware" has been developed by his colleagues at Xerox PARC 3 years after Weiser published *The Computer for the 21st Century. Context-Aware computing* is a result of the research performed on *Ubiquitous Computing* [5]. Nevertheless, Weiser mentioned that ubiquitous computing makes a whole new range of applications possible [6], and that these applications "*are of course the whole point of ubiquitous computing*" [7].

After a quarter of a century, context-ware computing, nowadays called *context awareness*, still is a hot research topic. This thesis therefore wants to contribute to existing research concerning context awareness, with the ultimate goal to make the vision of Mark Weiser come true.

Today, we know context awareness mostly as location awareness. Depending on the location of a user, the behaviour of one or multiple devices changes. This is for example typically used in home automation [8] settings: *when I'm in a certain room, the light should turn on*, but also in mobile apps, e.g. *find me the nearest restaurant.* Yet, context-awareness is more than only location awareness.

> *Context is any information that can be used to characterize the interaction of a user with a software system (and vice-versa), as well as the environment where such interaction occurs.* [9]

Supplementary to location data, there is lot of other, not necessarily 'physical', sensor data available. This can for example be preferences or properties of the user involved, mode of interaction, place/time of interaction, properties of the environment, history of interaction or the calendar of a user

As context awareness is strongly influenced by the person 'requesting' it, we should focus our attention more on that person together with their preferences. People typically have their own individual interpretation of context and implicitly want to apply their own preferences on it [10]. One person for example interprets bright sunshine as a motivation to open doors and windows to let the warm summer air come in, while someone else will keep everything closed to prevent pollen to come in. In order for a computer system to know these preferences, we need an interface allowing the end user to model these context preferences. In Weiser's paper, *Sal* programmed her

preferences into the system. However, not all potential users have enough programming expertise to do this. A lot of context-modelling systems, both academic and commercial, using different approaches already exist [11]. The Context Modelling Toolkit (CMT) [12] is such a system. It allows one to model context using a GUI based on specific building blocks. Because CMT uses a rule-based way to model context, each of these blocks represents a specific rule. The main target of CMT is to be easily adoptable for any type of user. This is accomplished by, first of all, enabling them to model context in a straightforward way. As every user is different, CMT also differentiates on the level of expertise of each user. But the most powerful feature is the fact that CMT can be used to model any possible type of context.

Unfortunately, to model any possible type of context, CMT needs to provide a high level of generalisation to the user, which inevitably comes with specific issues. The most significant being the fact that it is quite difficult to introduce a (hard) structure into an open system. For example, take a home automation system with a very low level of abstraction, i.e. it can only be used for home automation. In general, every device in such a system is a priori known. Assume the system knows that a coffee machine or fridge is available and what specific functions and parameters can be used/are needed to communicate with these devices to let them perform specific tasks. Due to the high level of abstraction, this is not the case for CMT. CMT allows any possible device, or better, any possible concept in context awareness to be modelled and reasoned about. A consequence of this high level of abstraction are the limited possibilities to structure the context building blocks inside CMT. Therefore, it is not trivial either to develop a protocol that enables to share rules and concepts based on those context building blocks between multiple instances of the CMT system. There is no way to predict how other users defined their concepts and rules and whether or not these can be mapped to the environment of another user. For example, one can create a rule to turn on the coffee maker when they enter the kitchen before 8 o'clock in the morning. Time itself is easy to map, i.e. it can be structured in a trivial way. Coffee machine is already a bit more difficult. What if a user has two coffee machines? Determining when our user enters the kitchen is the hardest part. One user might for example have an indoor localisation system based on beacons [13], whereas another user has their location being detected by means of e.g. triggering an RFID sensor [14].

How can we map these two ways of localising to each other? How can we generalise this for any possible concept involving context awareness?

1.2 Problem Statement

Many context modelling systems, both academic and commercial, already exist. Each of them has its advantages and disadvantages. Nevertheless, these systems mainly focus on one specific use case of context awareness, such as find activities nearby based on the weather and location or home automation [15]. One of the most well-known context modelling systems is IFTTT[1] [16] (IF This Then That). According to a variety of sources[2], IFTTT became very popular because of various reasons. One of the most important ones is the simplicity of rules and the possibility to use them directly without almost no configuration. An example of such a rule is given in Figure 1.1. Even to a user with limited skills in context modelling, it is immediately clear what this rule does. When a user then wants to apply this rule, they only have to connect their phone and Facebook account to IFTTT using familiar interfaces. The strength of IFTTT is the fact that it has a library with hundreds rules available, involving all kinds of popular services such as Facebook, Twitter, DropBox, Google Maps or even Github. As such, users are not required to have any programming skills. Downloading a rule from the library and applying it on the user's proper situation is all that needs to be done. However, creating a rule using this platform is trickier, especially if there is no matching "IF" or "THEN" block available in the IFTTT library. One cannot expect from a general user to enter API or file urls to create a rule. Using multiple IF blocks, such as e.g. *IF it is hot inside AND someone is home THEN turn on the airconditioning*, in one rule is not possible either. This also implies that one cannot make their own reusable abstractions to deal with complexity. IFTTT can hence be seen as a very simple system to model context with and gained a lot of popularity due to the extensive library with simple rules that can easily be applied to one's environment. Yet, this simplicity comes with a major drawback: it is very difficult for users to create their own rules, which might require more complexity to be more powerful.

For that reason, CMT has been developed, supporting more powerful rules. In this thesis, the functionality of CMT has been extended to enable users to share rules between different CMT instances. For example, if Alice made a very useful rule on her own CMT system, she will be able to share it with Bob. Enabling rule sharing also means enabling less experienced users to use more complex, powerful rules in their system that they would not be able

[1] http://www.ifttt.com

[2] `http://www.pcworld.com/`, `http://www.pcmag.com/`, `http://www.cnet.com/`, `http://www.news18.com/news/tech/`

Figure 1.1: Example rule on IFTTT

to create themselves. The next section will explain what contributions have been made to solve the problem stated.

1.3 Contributions

The main contribution of this thesis is the reduction of the complexity when creating rules for the context modelling tools. To get more insights in the possibilities of sharing rules, a research study has been conducted which focusses on how existing context modelling frameworks (can potentially) share rules. In this study both the technical (e.g. How to encode rules on the sharing side to make sure the importing side can import it) as well as the human perspective (i.e. what does a user need to do in order to import a particular rule) are involved. In the latter, the role of intelligibility will be a major aspect.

Next, rule sharing paradigms that can be used with CMT are studied. This is not as trivial as it might first look like, because rules and other structures in CMT are not based on pre-defined ontologies. This is because CMT wants to enable its users to model any type of context existing. CMT can, for example, be used to model more than only home-automation related context. Yet, this makes interchanging rules between two different CMT instances more challenging.

The rule sharing paradigm used requires a fuzzy string matching algorithm, as the reader will see later on. Therefore, an extensive study about different fuzzy string matching algorithms is conducted to determine which one has

the best performance considering rule sharing with CMT. In the end, it appears that using only such an algorithm is not sufficient. Adding an online service returning synonyms for a specific input term to the importing process of the paradigm appeared to be the best solution.

Last, there is the rule sharing module itself, enabling users to export and import rules and templates defined for the CMT. This extensible module maintains a clear separation from the internal implementation of CMT by interacting with only four public abstract components. Even with this constraint, the module is still able to enable rule sharing with CMT, independent of the specific user or environment in which a rule has been created. This is not trivial as CMT wants to allow its users to model any type of context and is therefore not based on a (fixed) ontology. Yet, this makes rule sharing a lot more complex, as different users have different sensors and artefacts they can use in a rule. The user they are then sharing that rule to might not have those. Hence, the sharing algorithm is designed on a high level and does require some user involvement, in particular when receiving a rule from another user. One of the key aspects of CMT is to maintain intelligibility throughout the system. As such, this key aspects has been kept into account when designing the sharing module, in particular at those moments in the sharing process where user involvement is required.

1.4 Thesis Outline

In this section, a quick tour concerning the structure of this thesis is given. First, related work concerning different context modelling frameworks is explored. The reader will get an insight in the different types of context modelling there exist nowadays, as well as how context is visually presented to the user. During the exploration of the literature, the focus will mainly be on the possibilities or potential a certain approach has to share rules. A trade-off between expressiveness and required user skill will be identified as well as the need for user intelligibility when sharing rules. The background chapter will then be concluded by a conclusion why sharing rules with CMT can be seen as an advantage for the user.

The next chapter aims to give the user insight about how CMT works from a user's perspective. Multiple illustrated examples will be given to understand the design choices of CMT and how these will affect the way rules can be shared between different users using different CMT instances. Throughout this and the next chapter, the examples given will be mainly based on mul-

tiple home automation use cases. Another section in the same chapter will then explain a possible graphical user interface, considering rule sharing, for the client side of CMT. Different sketches will give the reader a better insight in what a user is required to do in order to share a particular rule.

Having elaborated about the general functionality of the extension, the next chapter goes into full detail on how the rule sharing extension exactly works. First, some basic principles about the implementation of CMT are given in order for the reader to understand how the extension is attached to CMT and what constraints had to be kept into account when designing it. Hence, the next section goes into more detail about the actual connection between CMT and the rule-sharing extension, proving that there is an almost perfect abstraction layer between both. Knowing the CMT data model and how it is implemented, a study is conducted to check which of three discussed rule sharing mechanisms can be used with CMT. After this section, the reader will understand why only one rule sharing mechanism can be used with CMT. As this rule sharing mechanism is based on matching the names of different building block types (see later), one needs to find a good (fuzzy) string matching algorithm. Hence, five different string matching algorithms are studied in the next section, of which two make a chance to be used in the final algorithm. To end this chapter, a use case about Alice sharing a rule to Bob is given. During every step in this use case, the implementation of the extension is explained in full detail. Instead of showing the actual (more complex) code, each significant method is explained by means of one or more flowcharts to comfort the reader as much as possible in understanding this rather complex matter. In case the reader would be interested, the code matching these flowcharts can be found in the appendix

The final chapter summarises this thesis with a critical discussion about what has been presented and how it could be improved. The latter is presented in a separate section concerning future work possibilities. Finally, this thesis ends with some conclusions about the presented work.

2 Background

In this chapter, research in the field of context modelling is explored and discussed. The reader will first get an introduction to the terms "Context-Awareness" and "Intelligibility" to understand the design choices of CMT, the system on which the rule sharing extension is built. Hence, the latter needs to be designed according to the same principles as CMT. This chapter ends with a discussion of the rule creation and rule sharing mechanisms of different context modelling systems. A trade-off between the expressiveness of the rules versus programming skill of the users in a context modelling system will be identified.

2.1 Context-Awareness and Intelligibility

The first and major point of context modelling tool, is to present that context in a clear and understandable way to its target user(s). According to a study of Chen and Kotz, there exist two types of context awareness: *active* and *passive* [17]:

> ***Active Context Awareness****: an application automatically adapts to the context discovered, by changing the application's behaviour.*

> ***Passive Context Awareness****:an application presents the new or updated context to an interested user or makes the context persistent for the user to retrieve later.*

An example of an application using active context awareness would be a smartphone app changing its weather report when you travel to another city. The corresponding passive context awareness app would first prompt the user to confirm that the weather report can be changed to his new location. Hence, the reader can already infer that we will have to make a decision in CMT about inferencing actions from (user) defined rules: do we have to ask permission from the user to commit any action, potentially exposing unnecessary complexity of the system and frustrating the user? Or can we just do everything fully automatic, with the risk that we might not perform the actions a user wants at that specific time.

Some further research on this issue was has been done by Belotti and Edwards [18]. One of the insights they provided is that not everything can be sensed. At this time for example, it is not always possible to detect a users' mood, what they are wearing or derive specific preferences. Even if we would stuff the environment with sensors, the people interacting with it cannot be fully sensed, nor modelled. The only factor in an interactive context-aware system who can take the right decision, is the user himself. But therefore, he needs to understand the nature of the context-aware app: what are the capabilities of the app and what actions will it take under what circumstances? Nevertheless, we cannot expect feedback from the user for every possible situation. It is for example not feasible asking the user to update his mood and emotions into the system. Also for very complex systems (like p.e. auto-pilots of airplanes) we cannot expect users to know every specific detail about their behaviour.

Hence, we need to find a balance between autonomy and user-control as, according to Hardian et. al "*user's sense of control decreases when the autonomy of the system increases*" [19]. A system must therefore be understandable or, as Belotti and Edwards define it: *intelligible*, in order for the user to reason about it. Only then, the user can perform the right action. For example, assume a sunny day in July. In order to keep the living room at the right temperature the air conditioning is running. However, it does not succeed to keep the desired temperature. A user needs to have some knowledge on how the system works in order to solve potential issues. It might, for example, be that a window is still open. If the user would not know that the system cannot detect this, they could assume the system to

be broken instead of knowing that an environment parameter on which the system has no effect is set wrong, e.g. the window is open.

To bring this intelligibility to the user, Belotti and Edwards [18] propose some principles. First of all, the user should be informed about the system capabilities and understandings. This can be supported by providing the user feedback by means of feedforward, i.e. *What will happen if I do this?*, as well as confirmation, i.e. *What am I doing and what have I done.* This can also be in combination with what others are doing/have done to the system. But perhaps the most important principle, is to give control to the user, enabling them to override the system. This is particularly crucial in case of conflicts of interest.

CMT is meant to be used by users with different levels of expertise, meaning that its developers these principles into account when designing the system. Hence, as we are developing an extension to CMT, we also need to keep this principles into account and find a balance between a fully autonomous or user-driven decision making process. As will become clear later on, a rule originating from one CMT instance being imported into another CMT instance needs to be integrated in the latter. However this rule might consist of building blocks already available in the importing instance. Therefore a matching of these blocks needs to be performed, which might sometimes be fuzzy. In such cases, user intervention is required to determine which building blocks can be matched/merged with what existing building block. Does an existing building block need to be extended, or do we introduce an new type of building block in our system. To what level do we ask for user interaction? Can we only automatically merge when the names of two rules exactly match? Importing a rule would then demand a lot of user attention as most of the building blocks will require manual matching. This also includes blocks with only very subtle differences in their names, e.g. one is written in capitals while the other is not. Another way to tackle this problem, would be only requesting user intervention when the name of a building block differs that much with the one's in the database that it is impossible to find a match. However, assume we would now have two (location) building blocks named *bar* and *car*. As the names of these blocks only differ by one character, our system will automatically match them, although they are semantically completely different.

Finding a good balance between autonomy and user interaction is therefore critical. One solution to improve this balance could be giving more control to

the user. Let them decide on a certain threshold determining to what level the system should ask their intervention. Still, we could further improve the level of intelligibility by finding out what kind of information the user really needs in order to make CMT intelligible for them. An assessment of Lim and Dey [20] on the demand of intelligibility in context-aware applications states that different types of intelligibility in these applications exists. Lim and Dey identified 13 types of intelligibility in total of which three should be made available for all context-aware applications.

1. *Why*: Answering the 'why' question: *Why does a system act as it does?*
2. *Certainty*: How certain is the system in its decision or action?
3. *Control*: Allow users themselves to be in control, but warn them about the danger of doing so

2.2 Context Rule Sharing

As the core part of this thesis is about sharing rules between different CMT instances, this section explains the need to share rules by comparing different context modelling approaches and frameworks.

2.2.1 Motivation

From the late 1990s on, a revolution in telecommunication technology has changed our lives forever. Today, nobody can imagine a system that is not interconnected with one or more other systems in order to perform its tasks. Having for example a look to a home automation system: who would today buy a system that cannot be controlled remotely, e.g. from a smartphone. Most likely nobody. Having systems communicating with each other, whether or not via the Internet, significantly expands their possibilities. A smart thermostat could for example retrieve the weather forecast for tomorrow and already adapt the heather and/or AC in order to save energy.

More and more devices will get (better) connectivity in the near future, ultimately converging into the concept of *The Internet of Things* [21]. In this momentum, we need to extend CMT to also allow communication with other systems, more specifically other CMT instances. One important aspect is integrating a feature that allows users to share rules between their CMT instances. First of all, it opens the eyes of the end users about the possibilities the system really offers. If every user would just create their own rules

and never get in touch with those made by others, a lot of potential is not used as users will be biased by their own train of thoughts (cognitive bias). A second motivation to include rule sharing comes from a usability perspective. Creating rules demands energy from the user. As a lot of rules will be useful for most users, e.g. *turn off all light when I leave home*, it would not be very efficient to have those rules being created again and again by each user that wants to use them.

2.2.2 Context Modelling approaches

In this section, multiple context modelling tools will be discussed. As such, the reader should get some insight in the design choices of CMT presented later on. During the discussion of each approach, the advantages and disadvantages concerning context rule sharing will also be discussed. All of the context modelling tools discussed can be subdivided in following categories [22].

Ontology-based approach

In this category, developers define an ontology with possible context situations that can occur, e.g. `kitchen door opens`. The ontology is then used on the lower level to reason over contextual data entering the system. However, using an ontology-based system is not feasible in dynamic context modelling. One of the main requirements of CMT is enabling the user to change the context environment at runtime which implies that the ontology must be changed at runtime too. Hence, this might introduce problems with towards the integrity of the ontology on multiple instances of the same. An example of such ontology based system is SOCAM [23]. Concerning rule sharing, this approach look ideal. Assuming that every instance of context system has the same ontology installed, one can just encode a rule with respect to the ontology and then export it to another system. The latter will not have any problems decoding the rule and applying it, as it knows the same ontology. However, as already mentioned, the environment changes constantly, new types of sensors will be introduced. Such a system would be condemned to dead after a few years, as the ontology cannot be updated quickly enough to cope with technology changes.

Object-Oriented-based approach

A first OO-based system is JCAF[24], which works with so-called *context services*. New rules can then be created by composition of those context services, which is a major advantage as those context services can be added

and/or removed at runtime. Hence, JCAF can easily update according to technological changes. However, all the reasoning is pushed to the application layer, having the maleficent effect that client applications need to be redeployed when a user changes the behaviour of a rule. To handle this issue, another solution called JCOOLS [25] has been proposed. Essentially, JCOOLS is a combination of JCAF and DROOLS [26], the latter used to perform the reasoning. Client applications can use an application-specific XML schema to create and subsequently insert new facts and events. At first sight, JCOOLS looks to be the ideal solution. However it lacks support for end and expert users which is an essential requirement for CMT as every user, independently of their level of expertise, should be able to model context. Regarding rule sharing, this concept is a lot more complex. New context services can be added to the system at any point in time. Hence, when a rule is shared requiring such a specific context service, the system importing the rule needs to make sure that it also has access to that specific context service. E.g. Alice shares a rule requiring a coffee machine to Bob. Then Bob must have a coffee machine in order for that rule to work. It can be even worse as one cannot know beforehand what context services rules might require.

Component-based approach

Component-based systems, such as the *Context Toolkit* [27], work based on components called widgets. Each of these widgets processes low-level information, e.g. *the front door opens*, and maps it to a higher level in order to detect particular situation, e.g. *Bob got home*. Unfortunately the *Context Toolkit*, has its reasoning mechanism on the application level, just like JCAF. A second disadvantage is the fact that situations can only take widgets as input, implying that situations themselves cannot be reused. Focussing on the rule sharing potential of this system, the same issues as with the object-oriented approach can be identified. One cannot guarantee to have all widgets a shared rule needs on the importing system.

Apart from the context modelling tools discussed, there are also more commercial system available which are especially targeted on usability and user adoption. Though, these systems do not have significant contributions to this particular thesis [11].

Summarized, this section focussed mostly on the technical aspect of sharing rules concerning the different categories of context modelling tools currently available. The next section will focus more on the human side of the whole

picture. What is the easiest way for user to create rules and what implications does this have on the level of intelligibility? This section will also try to answer whether or not there is an actual need to share context rules.

2.2.3 Rule Creation techniques

To get a better idea on how we could share rules between multiple CMT instances, let us have a look on how already existing systems do this and what could eventually be improved. However, before sharing rules, we first need to investigate why it takes so much energy to create them.

In the interest of CMT, we need to find out how we can easily have end users, who are assumed to have no experience with programming whatsoever, program their own system, e.g. create their own rules, to have that system meet their personal preferences. As of today, multiple programming language metaphors [28] for non-programmers exist.

Visual Programming

A first very popular category in context modelling user interfaces is *Visual Programming*. One of the best-known ways of programming visually is the Jigsaw metaphor. It is for example used in programming environments for kids such as e.g. Scratch [29] and Puzzle [30]. Here a user just needs to drag and drop puzzle blocks representing statements, function calls or variable declarations of a normal programming language. But is not limited to pure programming only as even whole context-based home-automation systems can be controlled with it [31]. But there is more as iCAP [32], another visual context modelling tool, has even more expressiveness. iCAP allows a user to connect context entities such as `Person` *`"Jane"`* with `Room` *`bedroom`* and `Temperature` between *`15`* `and` *`20`* degrees. Hence rules such as for example *If Jane is in bedroom and the temperature is between 15 and 20 degrees, turn on the heater* are no problem for this tool as shown in 2.1. A user can choose a building block from the leftmost column and put it in the middle or rightmost column. The middle column defines `inputs`: if all the blocks in an input sheet, in Figure 2.1 two of these input sheets exist, the blocks in the output field will be executed. Unfortunately, iCAP does not allow these input sheets to be reused.

To conclude, one can state that *Visual Programming* has a very high degree of expressiveness. Nevertheless, this high expressiveness might get

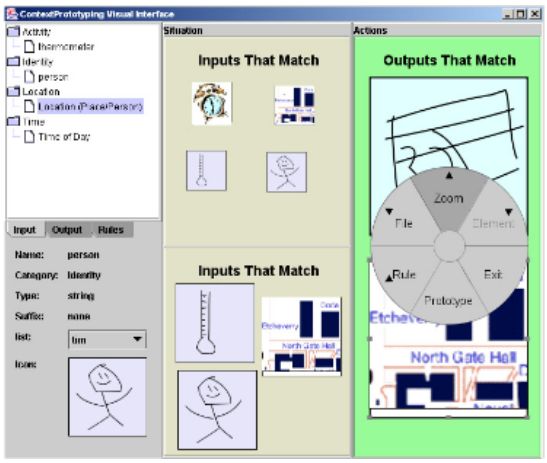

Figure 2.1: iCAP user interface with example rule using two input sheets Image from [32]

confusing for the user. Having 'too many' blocks to choose from, the end user might end up not knowing which one to choose to reach their goal. An option would then be to have the developer limit the amount of statements a user can choose from, but then again reducing the expressiveness.
A second way of Visual Programming, is the so-called recipe-based way used by e.g. IFTTT and WigWag[1]. As previously mentioned, these recipe-based ways of modelling context are very accessible for people with no programming experience at all. However their expressiveness is extremely limited due to the simplistic character of the rules that can be created.

Inductive Programming

Another category is *Inductive Programming*. A good example here are the macros one can use to automate tasks in Microsoft Office programs such as e.g. Excel and PowerPoint. A major advantage of this approach is that any user can create rules using a general rule creation environment, e.g. IFTTT. Based on these rules and the available modalities, e.g. for home automation: a smart fridge, light bulb, etc. the system can recommend new (similar) rules to the user. Take for example a rule, made by the user, that turns on the light when they arrives home. The system could then easily suggest to also turn on the heating when our user arrives home. Hence, the user does not need to create/program themselves to acquire new rules.

Nevertheless, the rules that can be recommended by the system will re-

[1] `http://www.wigwag.com/index.html`

main rather simple and based on already existing ones. Having the system recommend more complex rules will most likely result in (for the end-user) meaningless rules. Assume for example a system containing following rules: *"If the dishwasher is ready and someone is home, then open the dishwasher and ring a bell"* and *"If it is after 9pm and there is no movement and children are in their room, then children are sleeping"*. Potential rules that could be inferred are: *If it is after 9pm, then the dishwasher is ready*, *If the children are sleeping and the dishwasher is ready, then ring a bell* or *If the dishwasher is ready and the children are sleeping, then open the dishwasher*. Of these three inferred rules, only the last one makes sense.

When using simple inference, the user can still easily understand why the system has inferred particular rules. They only differ from an existing rule by one or two of the available modalities. However, when more complex rules are inferred, it gets more difficult to understand why the system inferred such rules. Additionally, these newly inferred rules might not cover specific side cases the user is not aware off, resulting in a complete loss of intelligibility. One solution to prevent this is having the user programming specific constraints into the rules that have been inferred.

To finish, we can conclude that Inductive Programming can be used to create new rules. Still, it is only viable when inferring simple rules and has therefore a rather low expressiveness. In order to increase that expressiveness, one could encourage the end-user to program parts of the rules they wants to use. Unfortunately, not every user might have the necessary programming skills to do so.

Tangible Programming

A next paradigm to create rules, is *Tangible Programming*, more specifically *Programming by Example*. Here, a user is encouraged to input different scenarios into the system by physically performing them. For example, if I arrive at my office, I want my desk lamp to turn on. This can be put into the system by making it 'listen' to actions that are performed. In this particular case opening the office door and switching on the desk lamp. Such systems already exist, for example GALLAG Strip [33] and the interactive mode of HomeRules [34] can be used to program a context-aware system by demonstration. A major advantage of this type of systems is that they allow any user, without any programming skill, to create complex rules.

But is it really that simple? Having a closer look to the two existing systems

mentioned before, one can observe that the actions performed by the user are transformed into building blocks, e.g. door-sensor or desk-lamp. The user still needs to combine these blocks in order to produce a trigger-action rule. The combination of these blocks is performed in a visual, recipe based way as described before. As it is now, programming by example has similar disadvantages as those coming with visual programming. Yet, the former is one step ahead, as the user does not need to find the right blocks any longer, they appear as the user is performing their actions. Though, it might become exhausting for the user to perform every (simple) rule they want to enter into the system.

Natural Language

A last way of creating rules is by means of *Natural Language*. By using this approach, a user can just talk to the system and tell it what to do. Two main possibilities exist First of all the user can act as a trigger to have the system perform a certain action. E.g. tell the system "*It's too bright inside*", making the system dimming the light. The other, more interesting possibility concerning this thesis, is to create rules with it. One can compare this with the nowadays popular personal assistants such as Siri[2] (Apple) or Cortana[3] (Microsoft). For example, telling these assistants "*When I go home, remind me to buy some milk*" will internally create a rule similar to, depending on the available modalities, `IF` *`Office door locked`* `AND` *`After 5pm`* `THEN` *`Show notification "Buy Milk"`*. These systems could easily be mapped to the home automation, or the whole context modelling use case. Frameworks such as Kitt AI [35] and or Google Now in combination with IFTTT [36] specifically developed for home automation already exist.

However, there are some drawbacks. In general, natural language processing gained enough accuracy to process simple queries, and therefore simple rules (e.g. *"turn off the heating when I leave home"* should be perfectly interpretable). Unfortunately, when one wants to create more complex rules, there is a risk the system might interpret the user input differently than intended. This can happen in two different ways. First of all, the longer the sentence a user dictates, the higher the chance that one of the words is wrongly mapped to text. Second, for longer rules, users might construct their sentences differently. For simple rules, most inputs will correspond to the following skeleton `IF...THEN...` (e.g. *IF I leave home THEN turn off*

[2] `http://www.apple.com/ios/siri/`

[3] `https://www.microsoft.com/en-us/mobile/experiences/cortana/`

the heating. A more complex rule such as *"Turn on the air conditioning when all doors are closed, the temperature is too high and the children are not sleeping and someone is home.* Here a user might first of all dictate a situation the system does not know yet. E.g. if *children are sleeping* is not yet known by the system, the rule the user wants to insert will not work. Second the system cannot always deduce where to insert logical operators (`AND, OR, NOT`) if the user does not explicitly mention them: do the doors need to be closed `AND` the temperature be too high, or is the clause true when one of the two is satisfied. Another issue is that the system does not know how to align the 'brackets' either: `(...AND(...OR...))` versus `((...AND...)OR...)`. Therefore we can conclude that natural language processing to create rules has only limited expressiveness.

2.2.4 Expressiveness vs Programming Skill Trade-Off

In the previous section, four different ways to create rules have been treated. Assuming an average user with less or no programming skill, these paradigms are ordered based on their expressiveness as shown in Figure 2.2. Please note that this comparison is made on very high level.

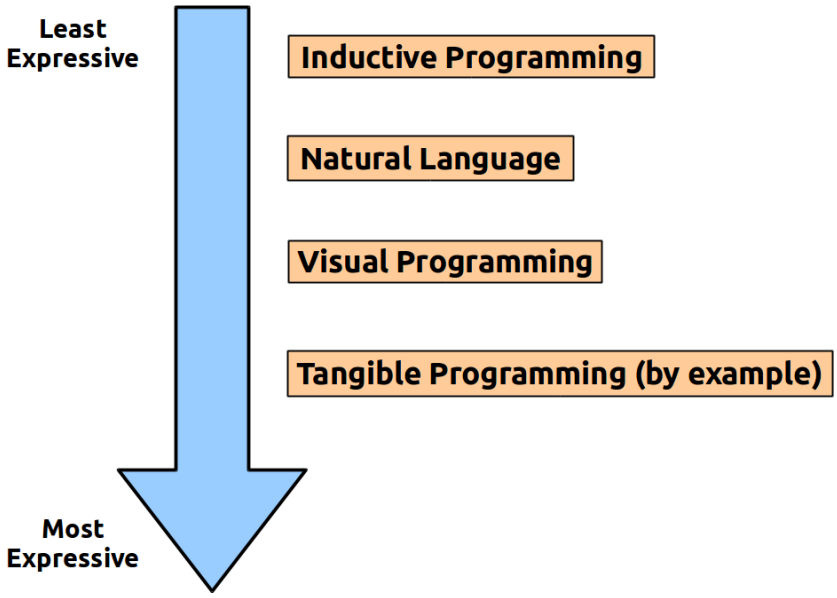

Figure 2.2: Rule creation mechanisms ordered by expressiveness

At the moment of writing, *Inductive Programming* can be seen as the least expressive way to create rules. A system can recommend new rules to the user by inferring from existing rules. However, to prevent the recommen-

dation of meaningless rules, only simple ones can be inferred. Next comes *Natural Language*. Here, a user can put rules into the system by 'talking' to it. Nevertheless, they needs to make sure the system interprets their input correctly and know how the system does this in order not too lose intelligibility. Hereafter, we have *Visual Programming* where a user can easily create rules by dragging blocks into one another, or create recipes with them. Though to make more complex rules, the user needs to have a clear understanding of what these blocks mean and how the logical operator between them act. Finally, there is *Tangible programming (by example)*. Using this way, users can create rules by emulating the situation(s) and performing the necessary action(s) themselves. In the end, this is very similar to visual programming, as the user still needs to combine the blocks resulting from his actions. The big difference however, is that the user does not need to choose the correct modality building block from a library, thereby resulting in more expressiveness than pure visual programming.

2.2.5 Conclusion: Sharing Rules for more expressiveness

Overall, we can conclude that none of the previously mentioned techniques gives users enough 'expressiveness' in the rules they can create. In order to create more complex ones, we have to look to the users themselves. As described in the CMT paper [12], CMT assumes three different types of users, being end users with little or no programming experience at all, expert users with some some, but still little, programming experience but eager to get the most out of the system and, finally, programmers. Hence, the solution is quite obvious. Expert users and programmers have the skills to create more complex rules. If we can make these accessible to regular end users, the problem is as good as solved without major loss of intelligibility concerning the latter.

Sharing rules does not only make more complex rules available to the end user, it also prevents unnecessary 'redundancy' during rule creation. A typical rule such as *Turn off the heating when I leave home* will be used by a lot of users. Why would users create such a rule themselves if it already exists?

3

The Context Modelling Toolkit from a User Perspective

This chapter gives the reader more insight into how CMT is perceived by the user. The user interface used to create rules and templates with is discussed, as well as why certain functionality is hidden from certain types of users. Next, we focus further on the rule sharing extension, more specifically on how it can be integrated into CMT purely based on user interface. As this thesis has been prioritized on finding a back-end solution to share rules between different CMT instances, some sketches about a possible user interface on one particular type of client are given.

3.1 Introduction to CMT

This first section is about why CMT has been developed and how it can be used for context modelling.

3.1.1 Motivation for CMT

In the first place, CMT is developed to offer its users the possibility to model context in any possible way. This can be at home, in the office or even on the go. There already exist a lot of context modelling systems, both

academic and commercial which are treated later on. According to Belotti and Edwards [18], the user adoption of such a context modelling system depends on two major factors:

- Does the system bring major enhancements into a user's life?
- Can the user easily and efficiently get insight into the internal workings of the system? (*Intelligibility*)

Giving the user insights into the internal workings of a system means that these internal workings need to be 'uncovered' in one way or another. A good way to do this is by giving the user, from time to time, feedback and/or asking them targeted questions, helping them to stay up-to-date with the reasoning of the system.

However, how much and what type of feedback can we give to a user? There is no general rule determining the balance between how much control the user should get versus software autonomy, i.e. full automation [19]. Initially, context-aware systems were mostly location-only based [22], but as complexity in these systems will only further increase, new ways to make interaction with the user easier need to be found. On the other hand, existing systems such as the well-known IFTTT[1] only provide very simple rules with a typical *if* `this` *then* `that` skeleton. Although these are very easy to use for the average user, they cannot cover all potential 'smart' behaviour [37]. Context awareness can, cognitively, be sensed in different ways depending on the user. For example, a person 'A' can define a meeting as *"having a talk with another person"*, while a person 'B' defines a meeting as *"being in a room with a projector"*. This 'personal interpretation of context' is therefore a tough issue to deal with specifically for this thesis as we want to enable users to share context rules with each other and reuse them in different events [38].

3.1.2 Rule-based context modelling in CMT

CMT is a rule-based context modelling framework, as one can derive from the subsection above. To cope with the different levels of expertise, CMT introduces a three-layered interface providing a seamless transition between the different levels of user expertise (further details will be provided later on in this section).

1. *End User*: Users without any programming experience. These users can only create rules based on templates made by expert users.

[1] `http://www.ifttt.com`

2. *Expert User*: Users that are interested in having some extra advanced functionality without requiring programming experience. These users can make rule templates (see later).

3. *Programmer*: The low-level programmer, having the knowledge and experience to completely configure the systems to their own and other users' needs.

Apart from the rule-based approach CMT uses, other ways to model context have also been proposed, as discussed in Chapter 2. By means of the three-layered interface, CMT implements the following requirements:

- A seamless transition between *different levels of experience* of end users should be provided.
- One should be able to *create events at runtime.*
- events, whether or not custom-made, should be *reusable as input of other events* .
- CMT is implemented using a *rule-based* client-server architecture, as shown in Figure 3.1. Data about state changes of facts as well as events that are happening are sent to the CMT server to reason about. This reasoning is based on event and context rules created by human clients and inserted into the system. Depending on the outcome of the reasoning, actions will be sent to the client, e.g. turn a light on or off.

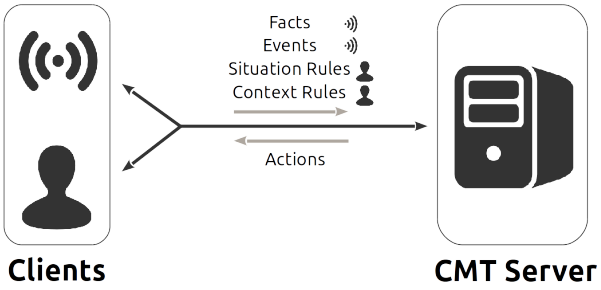

Figure 3.1: CMT client-server architecture. Clients can be humans and/or machines

In total, CMT supports three types of rules that users can use to model context: context rules, event rules and templates. Templates are used to create a skeleton for event rules. As these templates need to be instantiated, i.e. be converted to an event rule before one can reason about it, they are only used on the client. The next few subsections elaborate on these three types of rules. Every rule, independent of its type, has an *if ... then* structure, as shown in Figure 3.2. The `IF` side can take one or more input blocks. Depending on the type of rule, these can be facts or events. The `THEN` side takes one output block which can be, also depending on the type of rule, an event or action. In general, when the conjunction of all the input blocks equals 'true', the output block will be executed.

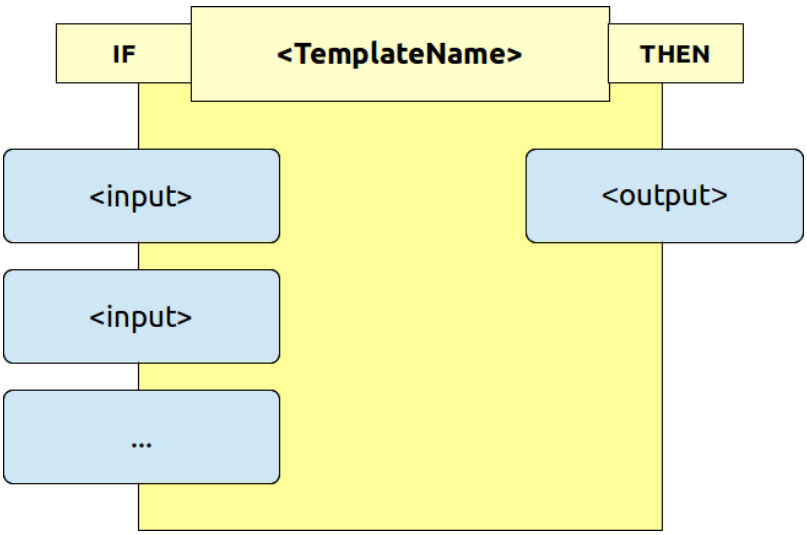

Figure 3.2: Basic structure of a CMT rule

As one can already derive from Figure 3.2, the CMT offers the user a drag and drop GUI where users can take blocks of various types out of a library on the screen and drag them onto fitting placeholders, i.e. of the same type.

In the following subsections, multiple examples will be given about the different types of rules CMT implements. Every time an example has been explained, we will give an update about which blocks are added/removed from the library. Initially, it contains the following blocks, subdivided by category[2] with syntax {("NameOfBlock",`<LowLevelTypeOfBlock>`);...} depending on the category of the block:

[2]Please mind that in real life situations, the block library will contain a lot more blocks. In the examples, it has deliberately been kept small in order to prevent confusion.

- *Template blocks*: {*"PersInLocation"*}
- *Custom Event blocks*: {"PersonInGarden"; "WaterTapOpen"; "TimeBetween"}
- *Event blocks*: {"Time"}
- *Fact blocks*: {("Alice", `<Person>`); ("Garden", `<Location>`); ("OutsideTap", `<WaterTap>`)}

Event Rules

The first rule category discussed is event rules. In general they are used to, as their name suggests, create new events. One of the main requirements of CMT prescribes that (custom-made) events should be reusable, therefore the input blocks of an event rule can contain pre-programmed events or self-created (i.e custom) events.

Take for example an event rule in the home automation domain depicted in Figure 3.3. The *PersInLocation* template[3] is used to create events where a specific person resides in a specific location. In this particular case, the user is expected to drag exactly two input blocks to the IF side, i.e. one block of type `Person`, the other of type `Location`. As such, an output block of type `PersInLocation` will be generated, which can then be reused in other event rules. Yet, before this block can be used, the user must give it a name, such as *AliceIsInGarden*.

After generating *AliceIsInGarden*, the block library is extended by one extra event block. *AliceIsInGarden* will be true if the current `Person` is indeed Alice and the current `Location` is indeed the garden. Hence, as already mentioned, once the conjunction on the `IF` side is true, the event defined on the `THEN` side will become true.

As already mentioned, another major functionality of CMT is that it allows the reuse of events, whether or not these are custom-made. In Figure 3.4, the reader can see how this works in practice. Assume an expert user created a template *GardenSpraying* in our CMT, shown in Figure 3.4, after the user created the event shown in Figure 3.3. It takes two input-blocks: one of type *AliceIsInGarden*, the other of type `WaterTap`. As the template is based on the *AliceIsInGarden* custom-made event, it is already filled in, which is

[3]More detail about templates will be given later on.

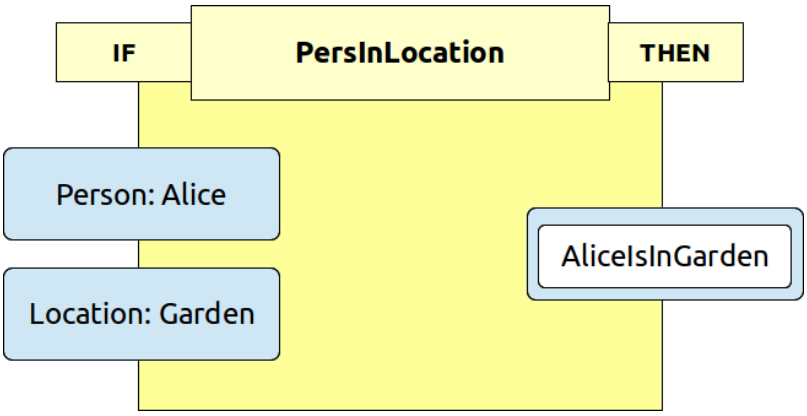

Figure 3.3: Example event rule: *AliceIsInGarden*, `true` if a specific person, here *Alice*, is in a specific location, here *Garden*

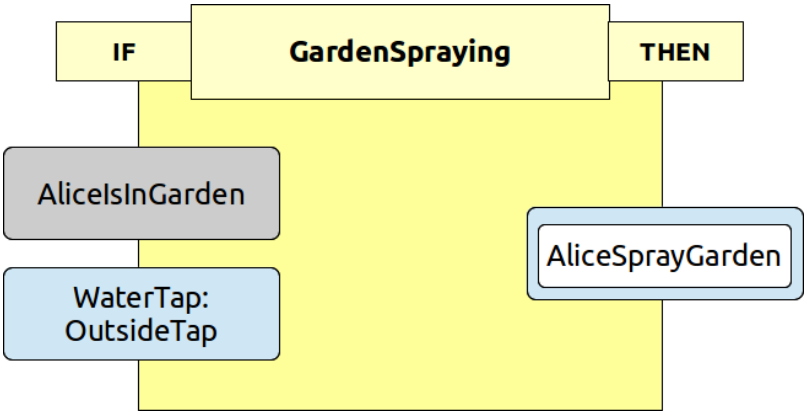

Figure 3.4: GardenSpraying template: reusing *AliceIsInGarden* from Figure 3.3

represented by a grey colour in Figure 3.4). In this way, a custom user-made event can be reused when creating new rules. Adding one final context event *OutsideTap* of type `WaterTap` and giving the resulting event a name, here *AliceSprayGarden*, makes the new event rule ready to use. Once Alice is in the garden and the outside water tap is opened, event *AliceSprayGarden* will become true. After saving, category *event blocks* of the block library will be extended with this new rule.

Template Rules

Event rules are created by dragging the right input blocks on the right placeholders of a template. Hence, until now, a user is limited to the templates

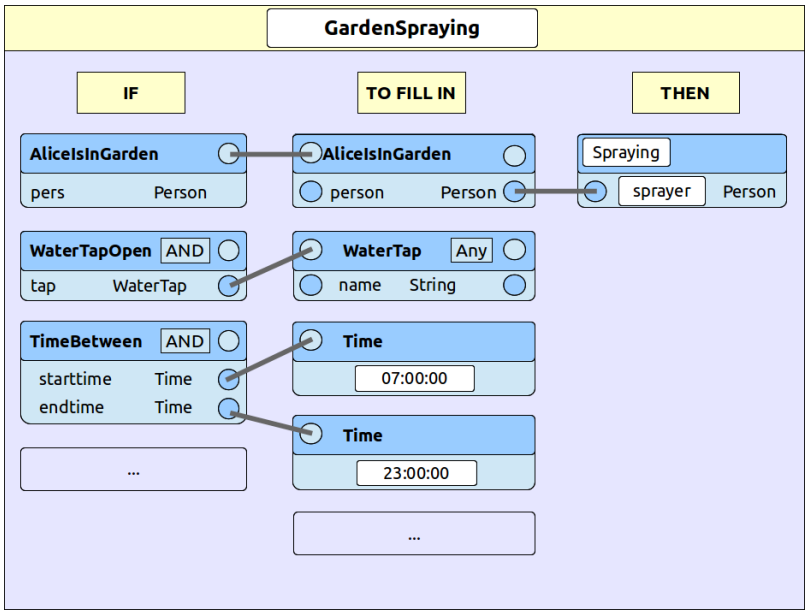

Figure 3.5: Interface to create rule templates. Example: *GardenSpraying* template

already provided by the system. In order to give more advanced users some extra challenge, CMT contains an interface to edit templates, which will be the main focus of this subsection. To prevent confusion, we will now talk about the *end user* as the user using pre-made templates to create events rules, and the *expert user* as the user creating templates.

In general, templates serve as a skeleton for both event and context rules. Invisible to the end user filling it in, a template takes functions and events creating the logic to determine if the event/action which the end user creates/wants to be performed, using the template, is true. However these functions/events need parameters, i.e. the input blocks the end user has to drop on the template. Therefore the template interface is subdivided into three main parts, as shown in Figure 3.5, where the *GardenSpraying* templates is being used to create the event shown in Figure 3.4.

1. *IF*: Takes functions and events that need to be evaluated in order to determine if the template is true. Chronologically, an expert user can add these as follows:

   (a) Drag a function or event from the block library onto an IF placeholder. In case of an event, the type does not matter.

(b) Once a first block, in the example function *PersonInGarden*, is dragged onto such a placeholder, the block itself expands to show the fields the function/event takes, using `<Fieldname, FieldType>` as syntax. In this specific case, *pers* as field name and `Person` as field type. The expert user can then associate these fields with values the end user has to fill, which will determine whether or not the function/event will evaluate to `true`.

(c) After having dragged and dropped a first block to the IF-side, the expert user can add as many blocks as they want. Yet, from the second block on, a drop down list will appear on it giving the user following options:

- `AND`: Means that this function/event AND the previous one should evaluate to `true` in order to make the IF side true.
- `OR`: Analogue to AND, this block OR the previous one need to be `true` to make the IF side `true`.
- `NOT`: trivial.

Considering the template shown in Figure 3.5, *TimeBetween, WaterTapOpen, PersonInGarden* all need to evaluate to `true` in order for the IF-side to evaluate to `true`.

2. *To Fill In*: The second 'column' of the template editor allows the expert user to bind actual parameters, i.e values, to the blocks on the IF-side. Hence, when the template is used, these will be the blocks the end user needs to assign a value to. As such, the *IF-side* determines the template's semantics, whereas the *To Fill In* side provides values that are applied to those semantics. Yet, the expert user can still decide what exact value is assigned to what formal parameters. As long as a value is assigned to every IF-block in the end, the resulting rule will work fine.

   Again, take Figure 3.5 as an example. Custom event *PersonInGarden* has one field called *pers* of type `Person`. However, by design of CMT, an expert user is not allowed to change this parameter[4]. Hence, the end user can only assign the *AliceIsInGarden* block to it, which the system therefore already fills in automatically as shown in Figure 3.4). Next let us have a look to the non-custom event IF-block *WaterTapOpen* having a field *tap* of type `WaterTap` to which the end user can assign a value. To make this possible, the expert user needs to perform the following actions:

[4]The exact reason for this is explained later

(a) Look for a block in the block library that fits the type of the parameter, here `WaterTap`. Hence, as an expert user, one can take the block representing the type `WaterTap` and drop it on a ToFillIn placeholder.

(b) Similar to the IF side, once a block is dropped on a placeholder, it expands to show its fields. Here `WaterTap` only holds one field, i.e. *name* of type `String`.

(c) From now on, the user can connect fields on the IF-side to fields on the ToFillIn side. In this specific case, *WaterTapOpen* needs a block of type `WaterTap`. Hence, the most straightforward way to do this is by connecting the *tap* directly to the block representing type `WaterTap`.

(d) In case the expert user wants to assign a specific instance of `Watertap` to *tap*, the interface provides a small menu on top of the `WaterTap` block to choose that specific instance. In this particular case, shown in Figure 3.5, the expert user did not change the default value, i.e. `Any`. This means that the end user, using the template to create a rule, should choose a specific instance.

(e) The expert user can drag as many blocks as they want to the ToFillIn area in order to assign actual values to the fields on the IF-side. Depending on the type of block, it is possible to already fill in values at template design time: e.g. `Any` as already discussed, or in case of a `Time` block: directly into the block itself.

3. *Then*: In the template interface, the THEN side allows the expert user to assign variable fields to events end users create by using the template. In this specific case, the expert user chose to make the *person* field of the *AliceIsInGarden* block variable and called it *sprayer*. This has no effect on the end user whatsoever, they can just fill in the template as if the change would not have been made. Yet, when the expert user designs a new template a (custom) event made by the end user, the variable fields of the template used to create that event will be displayed. In the specific case shown in Figure 3.5, when an end user uses this template to define a event, e.g. *AliceSprayGarden* as shown in Figure 3.4, and this event is being reused by an expert user in the template interface, the variable fields defined by the *GardenSpraying* template will be accessible for that expert user. In this particular example, only one field is involved, i.e. <*sprayer*, `Person`>, as show in Figure 3.6.

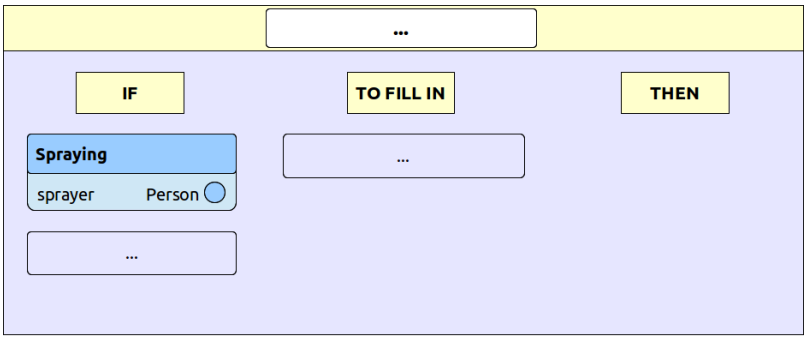

Figure 3.6: Reusing a event defined by a custom template in the template editor

Hence, after applying all these operations, the library of rules now looks as follows:

- *Template blocks*: {<PersInLocation>; <GardenSpraying>}
- *Custom Event blocks*: {"PersonInGarden"; "WaterTapOpen"; "TimeBetween"; "AliceSprayGarden}
- *Event blocks*: {"Time"}
- *Fact blocks*: {("Alice", <Person>); ("Garden", <Location>); ("OutsideTap", <WaterTap>)}

To summarise, it can be stated that the template functionality offered by CMT is a very powerful feature. It enables the reuse of custom events in different settings and generates new custom, events out of them. Technically, a template serves as a skeleton for rules. It takes (custom) events and functions containing several fields. It is independent of specific rules, but does determine what type of blocks an end user needs to fill in. To keep track of these types of blocks, a template stores bindings, i.e. what data of the blocks the end user dragged are needed to fill in the fields of the events and functions on the IF-side. The latter determining the semantics of the template and, therefore, the rules created with it. Finally, the end user only needs to drag specific instances of the right type on the template in order to create a new rule.

Context Rules

A final type of rules are the so-called *Context Rules*. In brief, these rules can be compared to rules one can make with other existing systems such as IFTTT. In CMT, a context rule looks like the one shown in Figure 3.7.

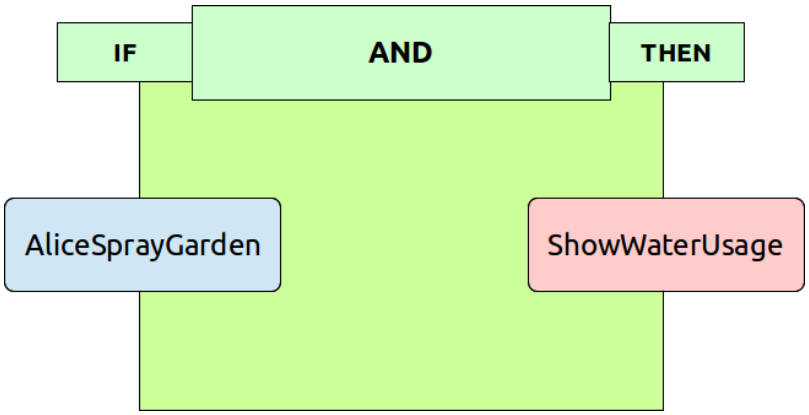

Figure 3.8: Example of a context rule: IF Alice is spraying the garden, THEN the water usage should be shown

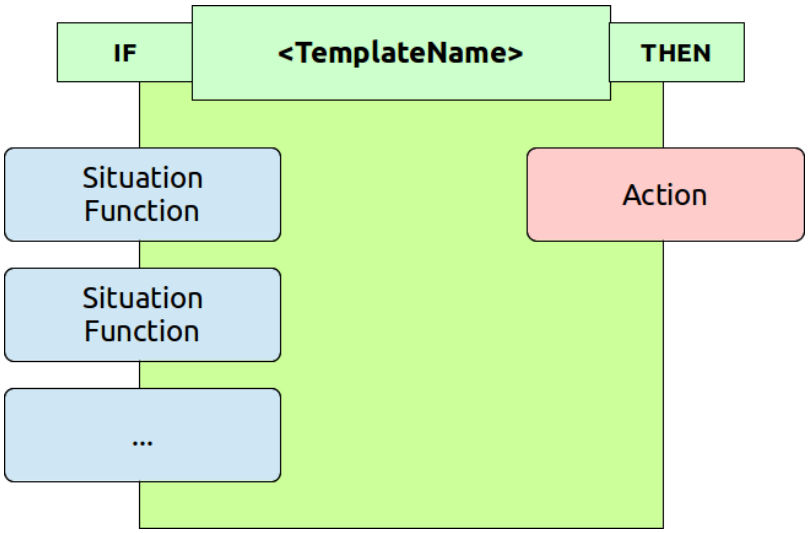

Figure 3.7: Basic structure of context rule

Context rules take one or more functions or events as input. These need to evaluate to true in order for the action on the *then*-side to be invoked. Whether or not all the events and functions on the IF-side need to be true depends on the template being used. For example, in Figure 3.8, the `AND` template is used, implying that all input blocks need to evaluate to true. However, in this particular case, only *AliceSprayGarden* needs to evaluate to true in order for the action, *ShowWaterUsage*, to be invoked.

3.1.3 Multi-layered context modelling

In previous sections, the notion of end and expert users have been used. These actually represent the multi-layered context modelling approach CMT uses.

CMT is meant to enable all kinds of users to model context. Yet, different people have different levels of expertise in the field of context modelling [37]. Therefore, to make CMT easier to adopt, different user skill based 'layers' are introduced. Existing systems also know this 'layering' of users, but only on a limited level. Mostly, they only have two layers. One for the regular end user to enable them to create and use simple rules, and one for the programmer connecting the building blocks of those rules with the available hardware. As a consequence, these systems have very little expressiveness. A good example is the previously mentioned IFTTT website providing such rules. Having a look at Figures 3.9 and 3.10, one can clearly see that these rules are pre-created by a programmer and can only be applied if the user has the appropriate hardware or user account to use these services. Hence, the end user has almost no control over these rules, e.g. in Figure 3.9, a Philips Hue[5] light(s) will turn on when it's 18:00. However, the end user cannot easily change the time when the lights should turn on, let alone reuse the rule to work with another brand of lights. The only way an end user can 'simulate' changing a rule, is by recreating it from scratch. In case of IFTTT, this is possible by selecting exactly two blocks, change the accessible parameters, e.g. in case of *time* the time itself, and applying the rule. Thus, one can state that the IFTTT approach of context modelling is based on a two-layered architecture. The end user has little or no input on how rules behave, e.g. the rules in the IFTTT library are just 'as-is', i.e immutable. Users can create rules themselves, but only with very little expressiveness, e.g. only pre-existing blocks can be used. Hence, IFTTT assigns most of the expressiveness to the programmers. Only they have full access to the power of the system and therefore the power to create blocks. As such, to increase expressiveness and control at the users' side, CMT introduces an extra layer between the end user and the programmer, i.e. the expert user. In this way, more advanced users get more access to the inside of a rule.

To conclude, one can state that the CMT is composed out of three seamless layers, described in Figure 3.11.

[5] `http://www2.meethue.com/`

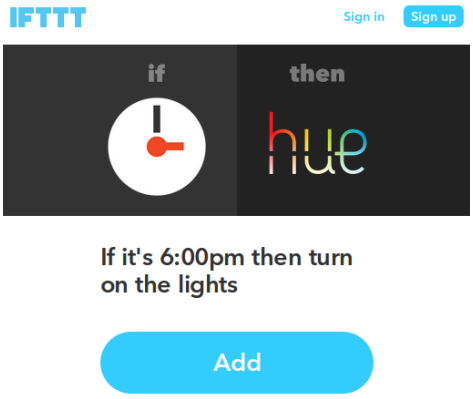

Figure 3.9

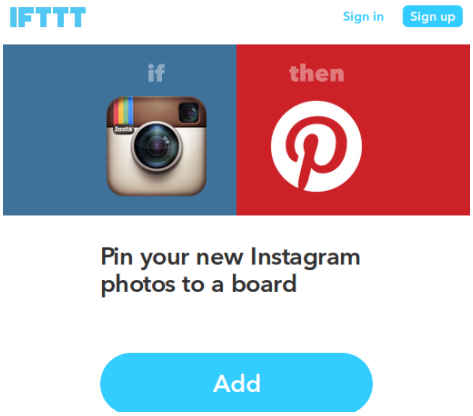

Figure 3.10

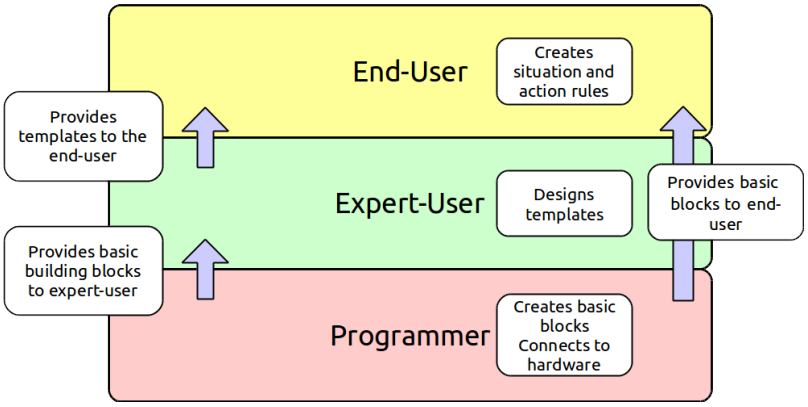

Figure 3.11: Three seamless layers of CMT

3.2 Sharing Rules as a User

In this thesis, the focus is mostly on the back-end functionality providing a REST API to any possible front-end client application. As such, a developer can still choose how to implement such a front-end client, as this requires some analysis on its own. For example, older people most likely need more help to export and import rules. Therefore, a wizard-based user interface might be more appropriate than for a twenty-five year old with a lot of interest in home automation, for example. Developers might also want to design different user interfaces for different devices, e.g. mobile phones, tablets or even general personal computers. Nevertheless, to give the reader an idea of what such a rule sharing interface could look like, six user interface sketches designed for an average tablet have been created. Each of them represents a step in the process of exporting and importing a rule.

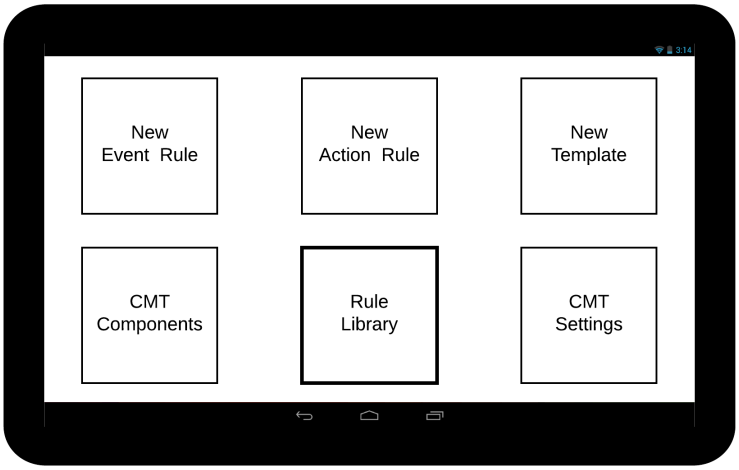

Figure 3.12: CMT Client: Mainscreen

When a user launches the CMT application, they first see the home screen which gives access to all of the functionality as shown in the sketch in Figure 3.12. From here, the user can create new rules and templates as well as add or remove components to CMT, i.e. sensors and devices such as a door sensor or a coffee machine. Next, it also allows changes in general CMT settings, e.g. IP address, and exploration of the rule library. In order to share a rule, the user must first navigate to it by browsing the rule library. Touching the corresponding tile opens the screen shown in Figure 3.13.

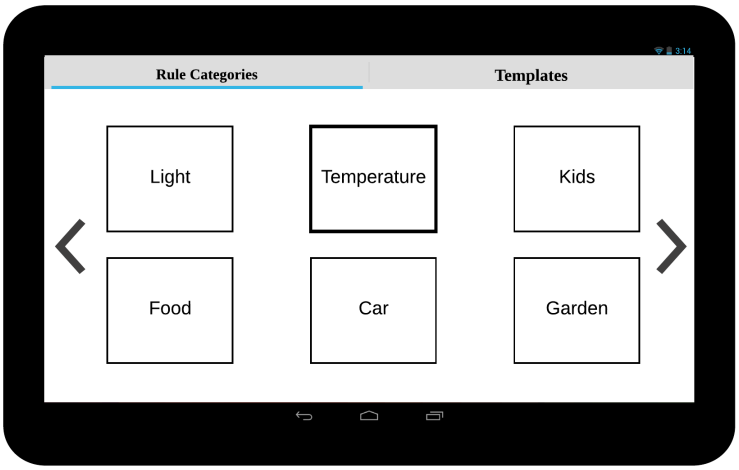

Figure 3.13: CMT Client: Rule Library

In this example of a user interface, the rule library consists of two parts depicted by the tabs on top of the screen. One tab gives access to the rules, grouped by a category users can choose themselves. The other tab allows template browsing. Assume a user, called Alice, wants to share a rule about the temperature of her swimming pool with another user, called Bob. She first opens the temperature category by ticking the corresponding tile shown on Figure 3.13. Next, all rules of category *Temperature* are shown, as being sketched in Figure 3.14.

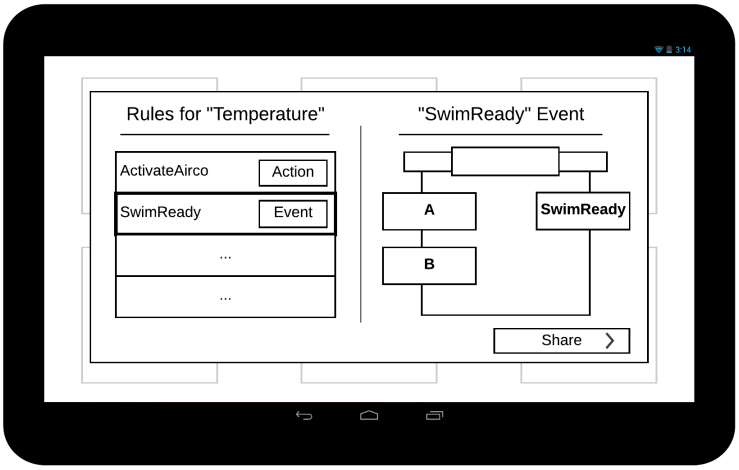

Figure 3.14: CMT Client: Rules of category *Temperature*

Only two rules are shown as an example in Figure 3.14. Each of them has a little box indicating whether it is an action rule, i.e. a rule that invokes an action such as activating the air conditioning, or a rule that defines an event. In the example, Alice selects the *SwimReady* event rule of which the event will be fired if all parameters of her swimming pool are ideal to go for a swim. Selecting a rule shows its visual representation on the right-hand side of the screen. As such, Alice knows immediately she selected the correct one and presses the share button, which brings her to the share interface shown on Figure 3.15.

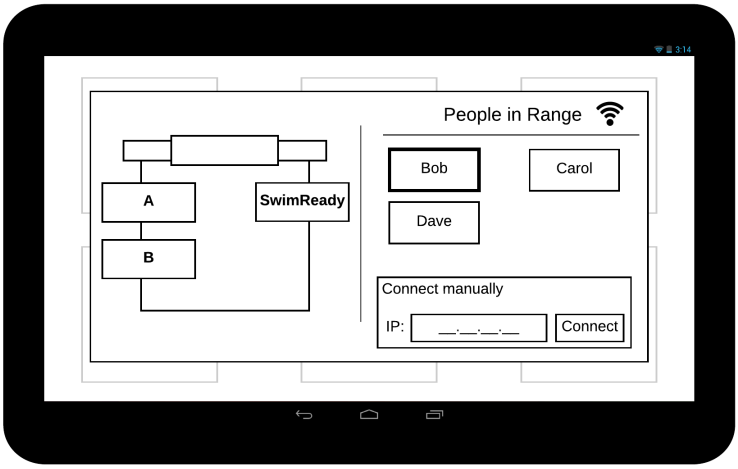

Figure 3.15: CMT Client: Sharing a rule with a specific user

The share interface first of all shows the rule about to be shared another time. This is to make sure that the user indeed wants to share that specific rule. But it also has a second advantage. Before sharing a rule, the two users involved will most likely sit together and discuss it. Hence, a visual representation of the rule can help during that discussion. On the right-hand side, the share interface shows an overview of all devices within wireless range, i.e. by scanning for other devices using WPAN (Wireless Personal Area Network) technologies such as WiFi-Direct or Bluetooth. In case a device would not appear in the overview, e.g. the device did not made itself visible, the user can still enter its IP address to connect manually. In Figure 3.15, Bob's client device appears immediately in Alice's overview. She can therefore easily tick his name, invoking a call to her CMT server to export the *SwimReady* rule to Bob.

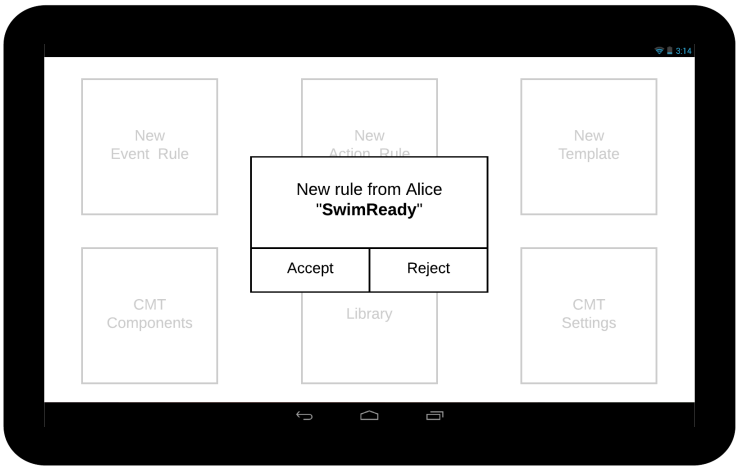

Figure 3.16: CMT Client: Shared rule arrives at the receiver's side

Once the rule arrives, Bob's tablet notifies him that a rule has been shared with him and asks whether or not to import, as shown in Figure 3.16. Bob knows, of course, that Alice just shared *SwimReady* and ticks the accept button which starts the import process on his CMT server.

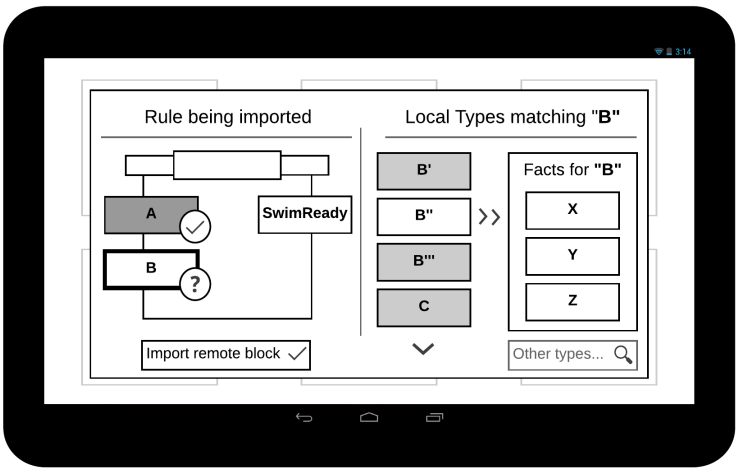

Figure 3.17: CMT Client: Matching local with remote building blocks

After a second or two, Bob's server responds, providing him with suggestions for each of the input blocks the rule is composed of, as shown in Figure 3.17. First of all, the interface shows a visual representation of the rule that has

been imported. From here on, the user is invited to solve the types of each of the input blocks, i.e. the blocks shown on the left-hand side of the rule. Assume Bob already solved the block of type `A`, visually represented by a check mark in Figure 3.17. Therefore, only the second block of type `B` still needs to be solved. When Bob ticks the `B` block on the left-hand side, a bunch of locally known block types appears on the right-hand side of the interface. These blocks are ordered by matching score: the better the score, the higher they will end up in the list of suggestions. Considering the specific case shown in Figure 3.17, three different types, i.e. `B'`,`B''` and `B'''`, are closely matching with `B` and hence end up on top of the list. According to Bob, `B''` can replace `B` in the original rule. By ticking type `B''`, a list of locally available facts of type `B''` appear, of which Bob can choose one to assign to the rule. In case Bob cannot find a matching type in the list given, he has two options. One is to manually search a type using the search box in the lower right corner. The other is to import the original `B` type block of the original rule. More detail about this second option will be given in the implementation section.

Once Bob solves all the blocks of the rule defining the *SwimReady* event, his server adds it to the local database and inserts it into the reasoning engine. From here on, the new rule is in use. However, the rule imported by Bob defines an event, i.e. *SwimReady*. Bob still needs to make at least one (action) rule in order for this event to become really effective. Such a rule can, for example, be *IF ItIsSunny AND SwimReady THEN OpenSwimmingPoolCover*.

4

Extending CMT to Share Rules

In this chapter, the extension built to allow the Context Modelling Toolkit to share rules will be discussed in full detail. First, an overview of CMT itself is presented in order to give the reader a better understanding why certain design and implementation choices considering the extension have been made. Next, the architecture of the extension will be discussed, which also includes further detail about how it is connected to CMT and its extensibility potential. Before going into the implementation details, three different ways to share rules are investigated of which only one will be used. Next, a study with 5 fuzzy string algorithms is being carried out to determine which algorithm is most feasible to be used for rule sharing. Next, the actual implementation of the extension is discussed. To prevent the reader from getting lost in this complex matter, a use-case is presented along. One is not required to read code either, as a flowchart is provided for each of the most significant methods in the implementation.

4.1 CMT Implementation: overview

This section is written to get a better understanding of the implementation of CMT, in particular those parts that are vital to understand the design and implementation choices of the rule sharing extension. CMT, a toolkit to model context, has been developed to tackle the problem of different users

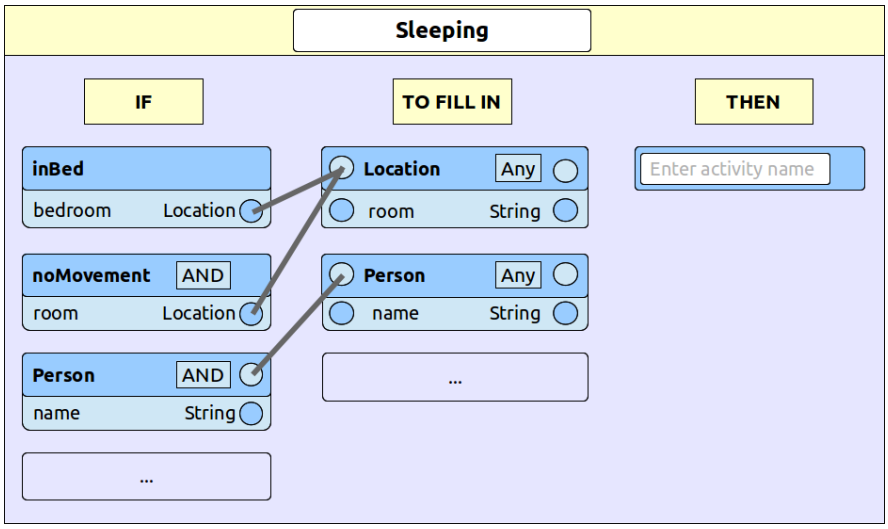

Figure 4.1: Example of a CMT Template

having a different level of expertise in the domain of context modelling. Additionally, it also enable a user to reuse the output of one rule in another rule. In the following, the example of context modelling in home automation will be used.

In essence, CMT assumes two basic types: facts and events. A fact can be seen as a fixed artefact in a particular environment that can have a specific state for a longer period of time. E.g. a kitchen lamp, a person named Alice, a heather, a door and so on. The state of kitchen lamp can then be for example `ON` or `OFF`. An event can be seen as a stream of information being sent if a certain event is true at a particular point in time. For example someone passing through a door or a light switch being pressed. In order to create relations between different facts and events, CMT supports functions. These can for example be used to check whether or not one or more facts or events are true at the same time, hence making another event true. For example, if a pressure sensor mounted under a bed detects someone is lying in bed and the person currently in the room is Alice, then event `AliceSleeps` is marked true.

As already mentioned before, to allow users with different levels of expertise to model context, CMT introduces templates. These templates can be seen as skeletons for rules and are only available to so-called 'expert users' to prevent general 'end users' getting confused. Such a template, as shown in Figure 4.1, consists of three types of building blocks. From left to right:

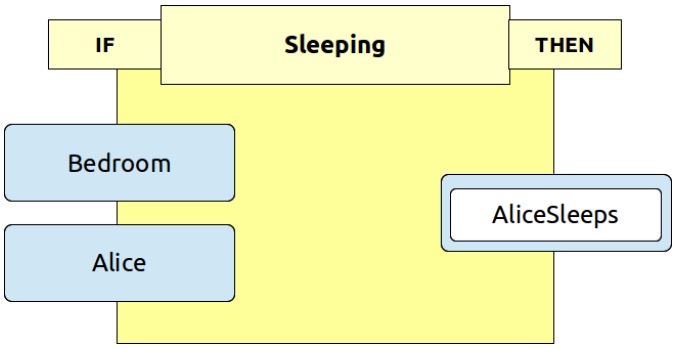

Figure 4.2: Event created with template `Sleeping`

IF-blocks representing the semantics of the template and therefore the rules made with it, *input-blocks* representing the values an end user has to fill in when making a rule based on the template and finally the *output-block* representing the custom event a rule based on the template creates. As such, an event being generated by this template might be `AliceSleeps`, as shown in Figure 4.2. Please mind that a custom event is always created using a template, whereas a standard event is preprogrammed in CMT.

Having a look back to the `Sleeping` template, one can observe lines drawn between IF-blocks and input-blocks. These are so-called *Bindings* and assign a value the end user filled in to the parameters of, or the IF-blocks themselves. In the `Sleeping` template, functions *inBed* and *noMovement* get an actual parameter bound to their formal parameter of type `Location`. In the implementation of the extension, there will be a lot of activity around the input blocks. Yet, to 'programatically' get to a particular input block, an algorithm needs to 'travel' through a lot of data structures. In order to get a better understanding of this, the internal structure of a template will now be discussed. Templates can create rules that generate events or rules, represented by `TemplateHA` and `TemplateActions` respectively. Yet, both classes implement the `Template` interface containing the template's name, IF-blocks and operators, as shown in Figure 4.3. The list of operators represent the `AND`, `OR` and `NOT` operators an expert user can set as relation between the IF-blocks of the template. E.g. in Figure 4.1, all the IF-blocks need to evaluate to true in order for the event generated by the template to be true. More interesting are the IF-blocks, represented by the `IFBlock` class. An IF-block has a string encoded type, which can be "function" or "activity" in case it hold a function or event respectively. Depending on the type the IF-block holds, one of both *function* or *event* fields will be filled in, i.e. not equal `null`. Functions such as *inBed* are represented by class `Function` with their param-

eters held by `CMTParameter`. Fields of these classes are trivial to understand.

Back to `IFBlock`: each IF-block has a list of bindings starting at that IF-block. Hence, a binding, represented by class `Binding`, has a start and an end point, represented by fields *startBinding* and *endBinding* respectively. For example, IF-block `Person` has one binding with start point itself and end point input-block `Person`. Both of these fields are abstracted by the `BindingParameter` interface. Downcasting *startBinding* results in the `BindingIF` subtype implementing `BindingParameter`[1]. Downcasting *endBinding* first results in the `BindingInputBlock` interface, in case a binding between an IF-block and an input-block is involved. Downcasting to an even lower subtype depends on both how the binding ends on an input-block. Considering the `Person` to `Person` binding in Figure 4.1, the binding ends on the block itself instead of ending on the field (*name*). As such, `BindingInputBlock` will be downcasted to `BindingInputFact` instead of `BindingInputField`. The difference between these two types only being what *field* the binding ends on. Finally, a programmer can retrieve the input-block from here on, encoded in the *inputObject* field. Input-blocks can be `FactType`s, `Fact`s or `EventInput`s (events), explaining why *inputObject* is of type `IFactType`, an interface all of these three types implement.

Before ending this section, a short explanation of the relations between `FactType`, `Fact` and `EventInput` is given. All of these three types are meta-types of types used in rules and templates. E.g type `Person` in Figure 4.3 has meta-type `FactType`. Whereas a specific person, e.g. Bob, has meta-type `Fact`. In the template editor, this difference is made by choosing a specific `Fact` from those available in the system for a specific `FactType`. In case of the template in Figure 4.3, the box behind `FactType Person` is set to *Any*. As such this input-block is a `FactType`, meaning that the end user has to fill-in a specific `Fact`, e.g. Bob, when creating a rule. Later on, the reader will get insight why this subtle difference plays a major role in sharing rules between multiple CMT owners.

4.2 Architecture

To situate the extension within the existing CMT implementation, let us have a look at the architecture. As shown in Figure 4.4, one can identify three layers of abstraction. In the middle, the CMT interfaces can be found, abstracting the extension from lower-level components. Four classes can be

[1]Please mind that `BindingIF` is not shown in Figure 4.3

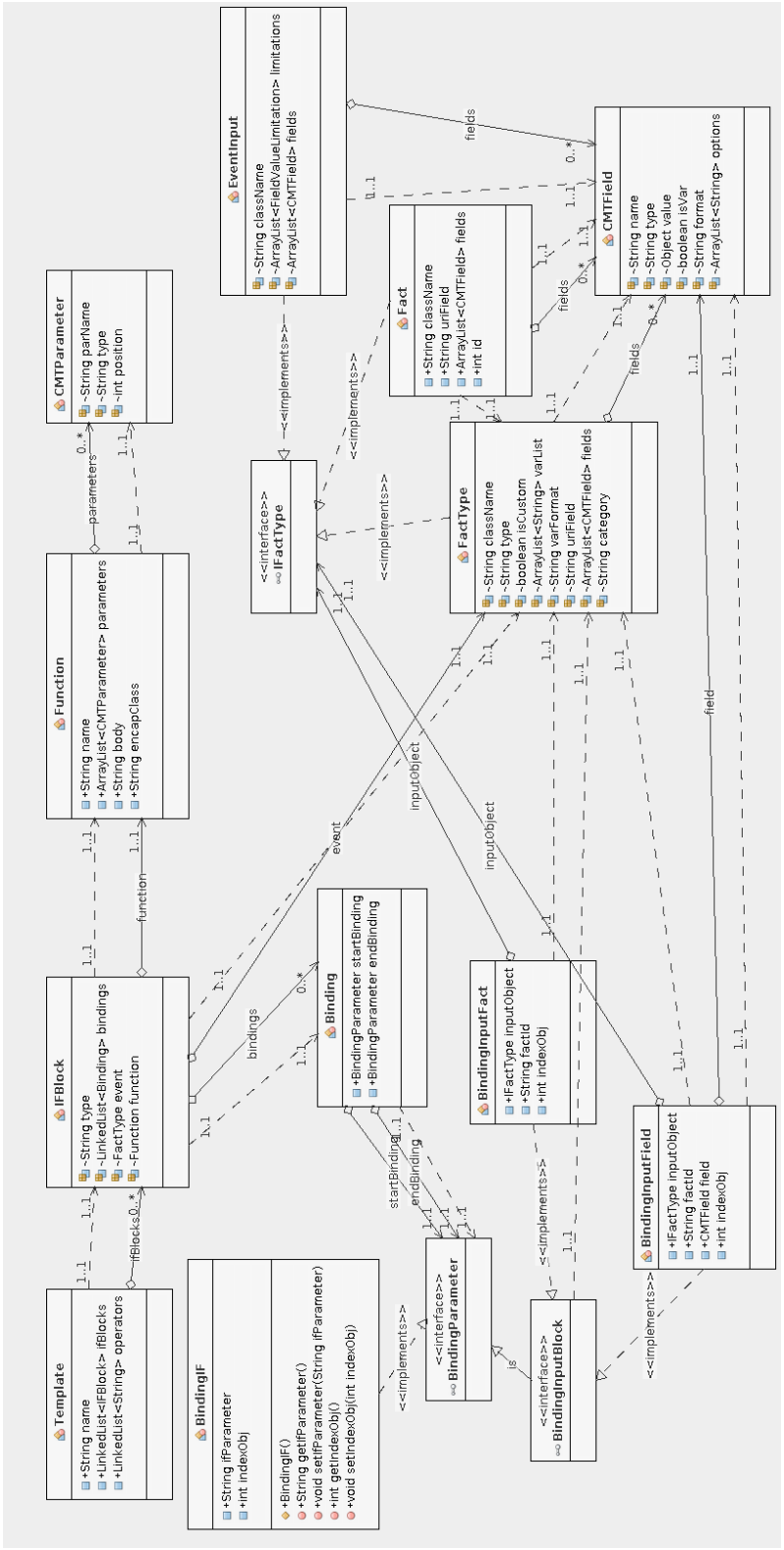

Figure 4.3: CMT Data Model, simplified

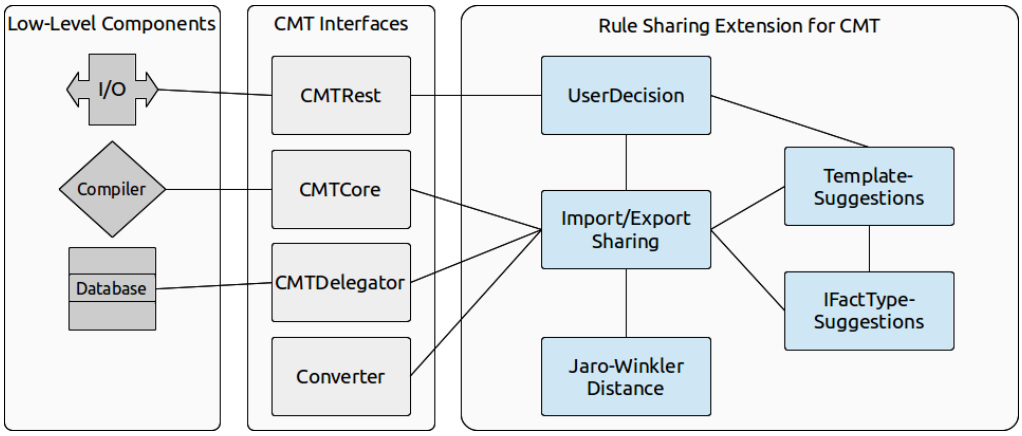

Figure 4.4: Architecture of the extension, relative to CMT

identified. First there is `CMTRest`, controlling the REST interface by which CMT communicates with its clients. The second interface is `CMTCore` which provides access to the application-level logic of CMT. `CMTCore` allows one to, for example, register new `FactTypes` in the reasoning engine[2] as well as modifying existing ones. Then comes `CMTDelegator`, providing access to the lower-level database. Adding, changing or modifying facts, events or templates can be done by using this interface. Yet, one should not forget to update the reasoning engine too if necessary, using `CMTCore`. The last class shown is the `Converter`, allowing a programmer to convert any CMT (meta) type, such as e.g. `FactType`, `Fact`, `Template` or `Binding`, to its corresponding JSON representation and back.

Having a closer look to the Rule Sharing Extension layer, one can identify the `Import/Export Sharing` component, containing all the core logic of the extension. First of all, it has connections to both the `Converter` of CMT, allowing the extension to convert CMT objects to a 'transportable' format. Originally the `Converter` class was meant to convert CMT types to JSON in order to send them back and forth between server and client. Yet, after some refactoring and additions, it can now be reused to convert CMT types in order to make them 'transportable' between different CMT instances, i.e. CMT instances owned by different users. Next there is the connection to `CMTDelegator`, which is, just like the `Converter` used during both importing and exporting. Considering the exporting process, it is called to retrieve templates and rules that need to be converted to an exportable format. During the import process, `CMTDelegator` is called a lot. In the first

[2]At the moment of writing implemented using Drools [26]

place, for each `FactType` of an input-block in a template, all `FactType`s of the database are collected to compute a matching score with that `FactType`. As explained later, this score will be used to provide the user closely matching blocks they can decide to reuse in the rule he is importing. Once all modifications by the user have been done, which also implies calls to the database, the import algorithm uses `CMTDelegator` a last time to write the imported rule into the local database. Importing a new rule might include introducing new `FactType`s or modifications to existing ones. In order to update the reasoning engine about these changes, `CMTCore` is used. Another connection of the `Import/Export Sharing` component leads to `UserDecision`, acting as an intermediary component that can be used to save state between retrieving a rule from a remote CMT instance and afterwards retrieving instructions from the user on how a rule should actually be introduced into the local system. `UserDecision` is also connected to the `CMTRest` interface already provided by CMT to handle the communication with the remote CMT instance from which the rule is retrieved as well as the client. Before finishing this section, there are another two components: `TemplateSuggestions` and `IFactTypeSuggestions`. The former is used to keep suggestions considering the building blocks of a particular template together, as well some extra information about the template itself, together and send it as a whole to the client where a user can then make decisions about how that template should be inserted into the system. The latter keeps suggestions for a particular building block together and hence, is always contained by a `TemplateSuggestions` object. Last there is the `Jaro-WinklerDistance` component, containing the fuzzy string matching algorithm being used to compute a matching score between a building block of a rule to be imported and local building blocks.

To conclude this section, the Rule Sharing Extension for CMT only makes use of the four already built-in CMT interfaces shown in Figure 4.4. Yet, some modification and additions needed to be made to all of the four components in order to have the extension work in a less complex way. Yet, these modifications and addition have no effect on the behaviour of CMT itself, nor do they affect its functionality. The extension itself is also extendible and components can easily be changed. In case a better matching algorithm is found, one just needs to replace the `Jaro-WinklerDistance` component with the new algorithm. In case REST would no longer be an appropriate technology to communicate, one does not need to replace the `UserDecision` component as only the `CMTRest` component depends on the underlying technology being used.

4.3 Rule Sharing Mechanisms

In following subsections, three different ways of sharing rules will be treated. Only one of them will remain as being viable for further use and will therefore be discussed in more detail in terms of its design as well as practical implementation.

4.3.1 Naive method

For someone not familiar with the internal implementation of CMT, the most straightforward method to share a rule or event would be just taking a rule from a CMT instance A, convert it to an intermediary mutually agreed format (e.g. using JSON) and insert into another CMT instance B. The first and second step, taking a rule converting and de-converting it, are not really a problem. However the third step, inserting the rule in CMT instance B, is not as easy as it looks like as shown in Figure 4.5. As action and event rules are roughly the same considering their structure, we will explain the event case. The action rule case is analogue.

Just importing a rule without having a closer look to the internal structure of it will most likely not work for a majority of rules. This is because a rule can only be applied if all or some of its IF-side blocks, so-called *IF-blocks*, evaluate to true. Yet, one cannot expect that the CMT instance importing a rule can map all of the IF-blocks of that rule onto e.g. sensors or functions. it already knows. For example, let us assume a simple case: Alice, using CMT instance A to control her home, shown by Figure 4.5, wants to share a rule to Bob, owning CMT instance B. The rule considered controls the heating of Alice's swimming pool depending on the water temperature measured by a *swimming pool temperature sensor*. Yet, Bob only has a simple swimming pool with a simple heater connected to a timer. If Bob does not have a *swimming pool temperature sensor*, Alice's rule cannot be applied to his system. This example is rather obvious, meaning that any user will immediately understand why such a rule will not work. However, rules can get more complex, containing more IF-blocks. If then one of the (required) IF-blocks cannot be mapped to a sensor or function, it will be way more difficult for a user to understand why a rule does not work. It can get even more complex. Assume for example that Alice shares a rule that turns her television off when she enters her kitchen. This can e.g. be detected checking whether or not the television is turned on and the kitchen door is opened. In contrast to Alice's home, Bob has two kitchen doors, one leading to the hallway and one to the living room. Although Bob's system has all neces-

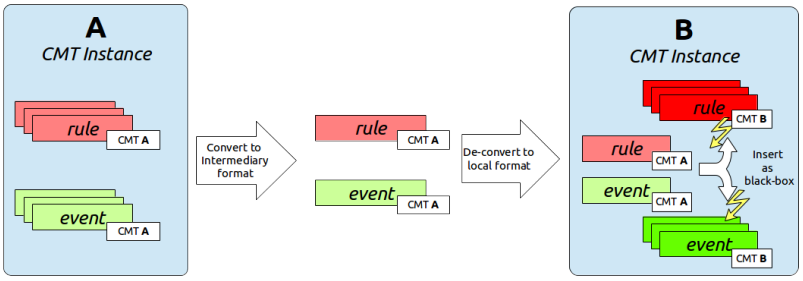

Figure 4.5: Sharing rules according to the 'naive' method

sary sensors, it cannot know which door sensor should be mapped to Alice's kitchen door IF-block in the rule he got from here.

As such, one can conclude that we need to look deeper into the structure and semantics of rules. Importing rules as a black box will most likely not work for most rules as shown above. As it seems now, we most likely need to expose the user to the importing process of the system. In this way, we can get help with the mapping when importing a rule. Yet, this means that we might loose intelligibility as some users might for example not understand what they exactly need to do when importing a rule. It is, to a certain level, required to know how a rule works, what logic it uses in order to get a specific result in order to know how the system treats new rules. Nevertheless, we can reject the naive, black box method to import rules.

4.3.2 Type Matching Method

Using the naive method is not an option to import our rules due to potential incompatibilities it generates on the importing system. As concluded in the previous subsection, we need to find a way how the IF-blocks of a rule from a CMT instance A can be mapped to the sensors available at another instance B. A classic way to do is, is by introducing types to every IF-block and sensor out in the field. For example, a temperature sensor, heater or AC might then be of type `Temperature`. To ensure compatibility on all CMT instances, every sensor then needs to obey a specific protocol for the type it has been assigned to. For example, if the protocol of type `Temperature` defines that temperature measurements have to be communicated in Celsius, every sensor of type `Temperature` needs to send its measurements in Celsius. If a temperature sensor of a CMT instance C provides its measurements in Fahrenheit, a rule involved with temperature imported into that CMT in-

stance C, will most likely turn on the air conditioning when it receives a temperature value of 65 degrees (Fahrenheit).

Assigning types and working with general protocols has some particular advantages. One of them being that a user importing a specific rule can immediately see whether or not that rule will be compatible with his current configuration and the reason why. Assume, for example, that we again have two users, Alice and Bob, owning their own CMT instances. As illustrated by Figure 4.6, assume that Alice owns CMT instance A and Bob CMT instance B. Alice's system is connected to, and supports, three different sensor types. In Figure 4.6, each of these types is illustrated using a coloured square where the colour determines the type of sensor. Alice's system supports types yellow (e.g. Temperature), purple (e.g. Light), orange (e.g. Door) and red (e.g. Bed). Bob's case is analogue. Both now want to import three rules, shown on the left in Figure 4.6. Assuming that each of the IF-blocks in each of these three rules need to be true in order for it to be executed, Alice can import the first and the last rule. The first rule requires sensors of type purple, orange and red (e.g. dim the light when user goes to bed), which are all supported by Alice's CMT and will therefore be compatible. This implies that the last rule will also be compatible, as the types of the IF-blocks are a subset of the first rule. Bob cannot import the first rule, as his system does not support purple typed sensors (e.g. Bob's lighting system is not connected to his CMT). For the same reason, the second rule cannot be imported by both Alice and Bob (e.g. they both live on an apartment and do not have a lawn, represented by the green type).

At first sight, this might seem to be a good solution for our matching problem. Yet, there are some pitfalls that must be taken into account. First of all, what happens if a new, high level, type is launched and rules become available for it? The answer to this is actually quite trivial. A new type can be seen as an already existing type that is being imported into a CMT instance having no support for it. To enable this support, a user must introduce an artefact corresponding to that new type into their system. Being a bit futuristic, presume that tomorrow spaceships are introduced, having their own type `SpaceShip`. If Alice wants to use a rule that turns on the heater when she arrives home with her spaceship, she first needs to buy a spaceship and connect it to her CMT. Only then, the rule will have an effect. Hence, adding new types over time does not seem to be that much of an issue. Yet, this is not the case when having a closer look at it. Assigning just one particular type to an artefact is not that easy. Many artefacts can have (very) specific

functionalities that cannot be covered by their general type as this need to 'summarise' the functionality of all artefacts. Assume for example the type `Switch`, only supporting general switches that can only have two states: `on` or `off`. In order to introduce switches with dimming functionality, we need to make a new type for this as type `Switch` does not support intermediary states in between `on` and `off`, i.e. 20%, 50%, etc. are not supported, only 0% and 100%. Unfortunately, this might cause a proliferation in the amount of types being used among CMT instances in an unstructured way. For example `Switch` and `Dimmer` are related types, but no relation is established between them upon creation of `Dimmer`. In essence, this is not such a major problem, but it will only become more difficult for sensor developers to find the right type for their product in the 'unorganised wilderness' of available types. The most straightforward solution is to introduce structure in that proliferation of different types. Just like in programming languages, we can opt to create a certain hierarchy between types, e.g. by means of subtyping. The `Dimmer` type would then be a subtype of `Switch`. Though, this approach also has its own drawbacks. First of all, it demands strict discipline from the creators of types to maintain the hierarchy and find the right way to add a new type into the hierarchy. For example, should a sensor measuring the temperature of a swimming pool be subtyped under `SwimmingPool` or `Temperature` or both. Designers of these types will need to find an agreement that fits everybody. Secondly, every time a new (sub)type is introduced, we need to update the hierarchy on all CMT instances that might get in touch with that new subtype. One way to do this is by adding a description to each new type describing where exactly in the hierarchy it should be integrated. Once a CMT instance encounters such a new type, it can then use that description to fit the new type in its local hierarchy. However, it might be the case that when a CMT instance gets in touch with such a new subtype, it does not know the supertype. Assume, for example, the existence of the type `FlatScreenMonitor` having subtypes `LCD` and `Plasma`, where `LCD` is further subtyped into `StandardBacklight` and `LEDBacklight` as illustrated in Figure 4.7. Assume now that Alice has a CMT instance with a rule that turns off all her flat screen monitors when she leaves home. At a certain point in time, she decides to buy a more ecologically friendly LCD monitor with LED Backlight. With the current approach, her CMT instance will not accept the new monitor as it does not know the LCD type. We now have two options: the first one is to include the whole hierarchy up to a specific point in time in each sensor. The second one is to provide a centralised service where every CMT can then request the types it needs. In our specific case, Alice's CMT could then obtain the missing `LCD` type, necessary to install her new LED

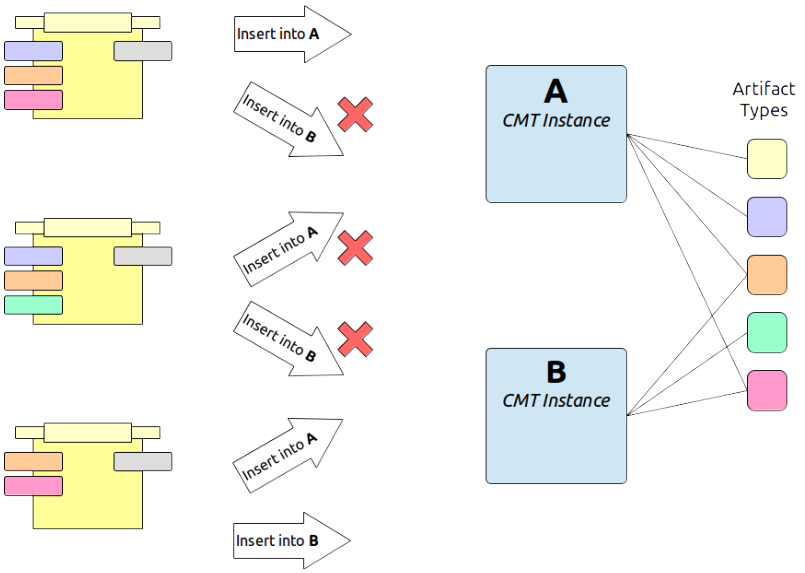

Figure 4.6: Sharing rules using high level types

monitor and allow rules to work with it.

Still, this approach does not solve all problems described when elaborating about the naive method to import rules (see previous subsection). Having a look back to the "two doors problem", this mechanism cannot offer a fully automated way of importing rules either. Remember that Alice wants to share a rule with Bob, a rule that turns off her television when she enters her kitchen, detected by a sensor on her kitchen door. Bob's home has two kitchen doors: one leading to the hallway and one to the living room. Except for the case where Alice's kitchen-to-living room door has been subtyped 'deep' enough, e.g. all the way to the type `KitchenToLivingRoomDoor`, for Bob's system to know that that door is meant, Bob can import the rule without any further issues. But what if Alice now wants to share this rule with Carol, whose kitchen door is connected to her lounge. Carol will not be able to import the rule, as she does not have a type `KitchenToLivingRoomDoor` in her system. Therefore, we can conclude that subtyping until this low level, i.e. describing where a specific door leads to, will make it impossible for a lot of rules to be shared. On the other hand, if we do not do this, the system might not know on what door the rule should be mapped. Hence, this method is rejected.

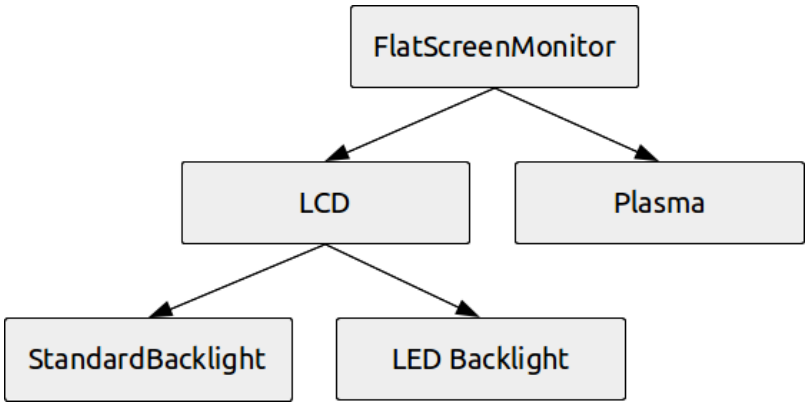

Figure 4.7: Subtyping example

4.3.3 Name-matching method

As has become clear from the previous subsections, we need to involve the user into the process of mapping the IF-blocks of a rule on the sensors types of the importing CMT-instance. A second advantage of involving the user in the process, is that it increases the users' level of intelligibility about the system and the import process itself. Still, we need find a way allowing the user to do the mapping process without requiring too much cognitive effort.

Independently of how the user will be exactly involved into the mapping process, we need to analyse the IF-blocks of the rule to be imported and expose the user to it. The most straightforward way to map the IF-blocks on known sensors, is asking the user to do it for us. However, mapping a local building block, which represents a `FactType`, `Fact` or `EventInput`, to every IF-block of a rule by hand is not feasible without any additional help. Assume Bob wants to import a rule from Alice that turns on his alarm clock when he lays in bed and the light in his bedroom is switched off (shown in Figure 4.8). Having Bob doing the mapping process without help, he has to find a mapping for both `PersonInBed` and `BedRoomLight` by searching through all the building blocks his system has. This can be 10 blocks, but as Bob's system will most likely grow over time, this might increase to 20, 50 or even 100rds of building blocks [3]. This simply requires too much cognitive effort from a user, especially for a system designed to make his life easier to

[3]Please remind that CMT is not limited to be used in a home automation environment. It can be used to model any type of context. All rules (home automation related, phone preferences related, etc.) a user might have can be managed by a single CMT instance. This implies that the amount of rules might become very large over time depending on the use case(s).

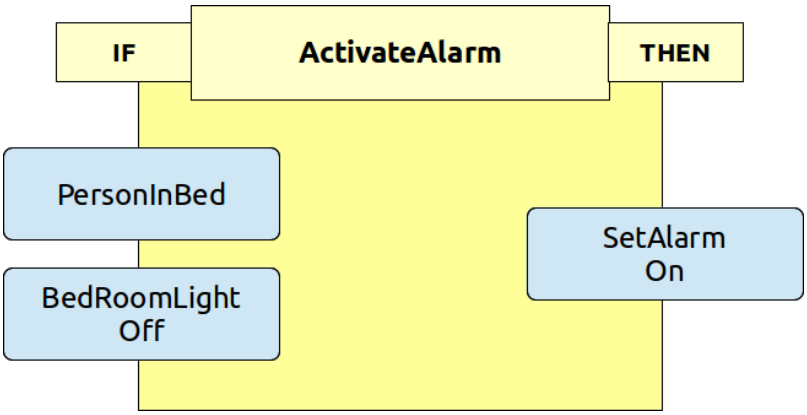

Figure 4.8: Rule to turn on the alarm when Bob is in bed and the bedroom light is switched off

manage by automating certain parts of it.

The most straightforward way to help a user in the process of finding the right local building block, is by suggesting some. This could be done in two ways: type-based or name-based suggestions. In the type-based way, the system checks whether or not there are some similarities in the fields of the block to be mapped with a local block. Blocks having a similarity above a certain threshold are then suggested to the user as potential matching blocks. Yet, this might introduce some strange results. Assume for example a building block representing the status of a drill and one representing the status of a hard disk drive. Both have a temperature and vibration sensor. Hence, their types (e.g. `Drill` and `HDD`) will both contain a field of FactType `Temperature` and one of FactType `Vibration`. As such, if Bob has to map a rule with an IF-block representing a drill, the system will most likely suggest building blocks of FactType `Drill` and `HDD`. From a user's perspective, it might be very confusing not to know why the `HDD` suggestion is offered, hence harming the level of intelligibility we want a user to have about the system. As such, it is more useful to focus on a building block as a whole, rather than analysing its internal structure. Yet, there is only one property that represents the overall (abstract) semantical value of a building block: its type name. Considering the rule Bob wants to import shown in figure 4.8, the type names of the two IF-blocks are *PersonInBed* and *BedRoomLight*.

Practically, matching building blocks based on their type name is not as trivial as it might look like. First of all, we cannot just match using regular string matching algorithms that look for equality between two strings, as

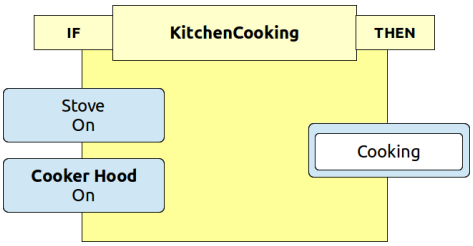

Figure 4.9: Imported definition for *Cooking*

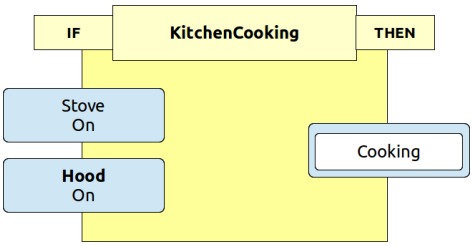

Figure 4.10: Own definition for *Cooking*

users might name their own custom-made events differently, even when they are semantically the same. Assume Bob wants to import the event declaration for the event `cooking` shown in Figure 4.9. As Bob owns a CMT system for many years now, he does not remember that he already has a similar, semantically equal, declaration of the cooking event. The only difference is that he did not call the hood above the stove `cooker hood`, but just `hood`, as illustrated in Figure 4.10. Hence, the string matching algorithm being used to create suggestions for Bob must be compatible with this kind of similarities, e.g. `cooker hood` must match `hood`. The next section provides more insights considering the choice of a fitting fuzzy string matching algorithm, as well as their behaviour when comparing abbreviations of a term with that term itself. However, there is one last aspect we cannot solve with (fuzzy) string matching algorithms: synonyms.

Assume that Alice uses `Place` as type for an IF-block in a rule she wants to share with Bob. However, Bob's system only knows the type `Location`, which is semantically equal to Alice's `Place`. As this is a pure linguistic problem, the best way to solve it is by using linguistic resources. As such, the current implementation of the extension consults an online service using REST [39] to retrieve synonyms for a specific term. More specific information about how synonyms are exactly handled can be found in the next section. One might wonder why the current system does not use an external service to lookup abbreviations. At the moment of writing, there exist lots of online services returning a list of potential abbreviations for a specific term (and vice versa). Yet, users can still use their own arbitrary abbreviations that might not appear in the list of official ones retrieved from the online service(s). In the next section, it will become clear that this is not strictly necessary as the fuzzy string matching algorithm being used shows tolerant behaviour towards most abbreviations.

4.4 String Matching Algorithms

As already mentioned, exact string matching algorithms are not feasible to use when matching type names of building blocks in CMT. Therefore we will have to focus our attention on *Approximate String Matching Algorithms*, more specifically to *Fuzzy String Matching Algorithms*. Having a look to the entity names matching problem [40], we can already find following potentially suitable matching algorithms: *Sorensen-Dice, Levenshtein Distance, Jaro-Winkler Distance and Smith-Waterman*. As one of the most popular fuzzy string matching algorithms, *Hamming-Distance* should also be considered. In order to check which of these (or a combination of) algorithms perform best in our case, a small Java program has been written showing the return (match) value of each algorithm given two strings (aka. needles) as input. In Figure 4.11, one can see this comparison program in action. On the left-hand side, a small bunch of home automation terms can be found, mostly taken from the CMT paper [12]. These terms will then be used as needles to compare another needle using each of the algorithms specified in the column headers. Selecting a needle on the left will fill it into the `ToMatch` text field to compare it with the other needles in the leftmost column of the table. If needed, a user can also enter a custom needle in the same field. In this case, *Kitch* is compared against 13 other needles using five different algorithms. Using the slider below the table, a user can set a threshold (between 0.0 and 1.0), which will colour scores in the table greater than it in green. This should make it easier to get an overview of which algorithms match best on what needles. We will now cover each of these algorithms and determine why they are (not) suitable for our semantic matching problem.

4.4.1 Sorensen-Dice index

The Sorensen-Dice index[4] has been developed independently by Thorvald Sorensen [41] and Lee Raymond Dice [42]. Given two needles `X` and `Y` respectively split into bi-grams[5] $\mathtt{N}_1$ and $\mathtt{N}_2$,, the Sorensen-Dice index can be computed as follows:

$$QS = \frac{2|N_1 \cap N_2|}{|N_1| + |N_2|}$$

[4] `https://en.wikipedia.org/wiki/Sorensen-Dice_coefficient`

[5] Wikipedia definition for bi-grams: A bigram or digram is a sequence of two adjacent elements from a string of tokens, which are typically letters, syllables, or words. From `https://en.wikipedia.org/wiki/Bigram`, August 6th, 2016. In this particular case, a bi-gram of a word is a collection where each element contains 2 consecutive characters of that word. In case the word has an odd amount of characters, the last element is allowed to contain only one character: the last character of the word.

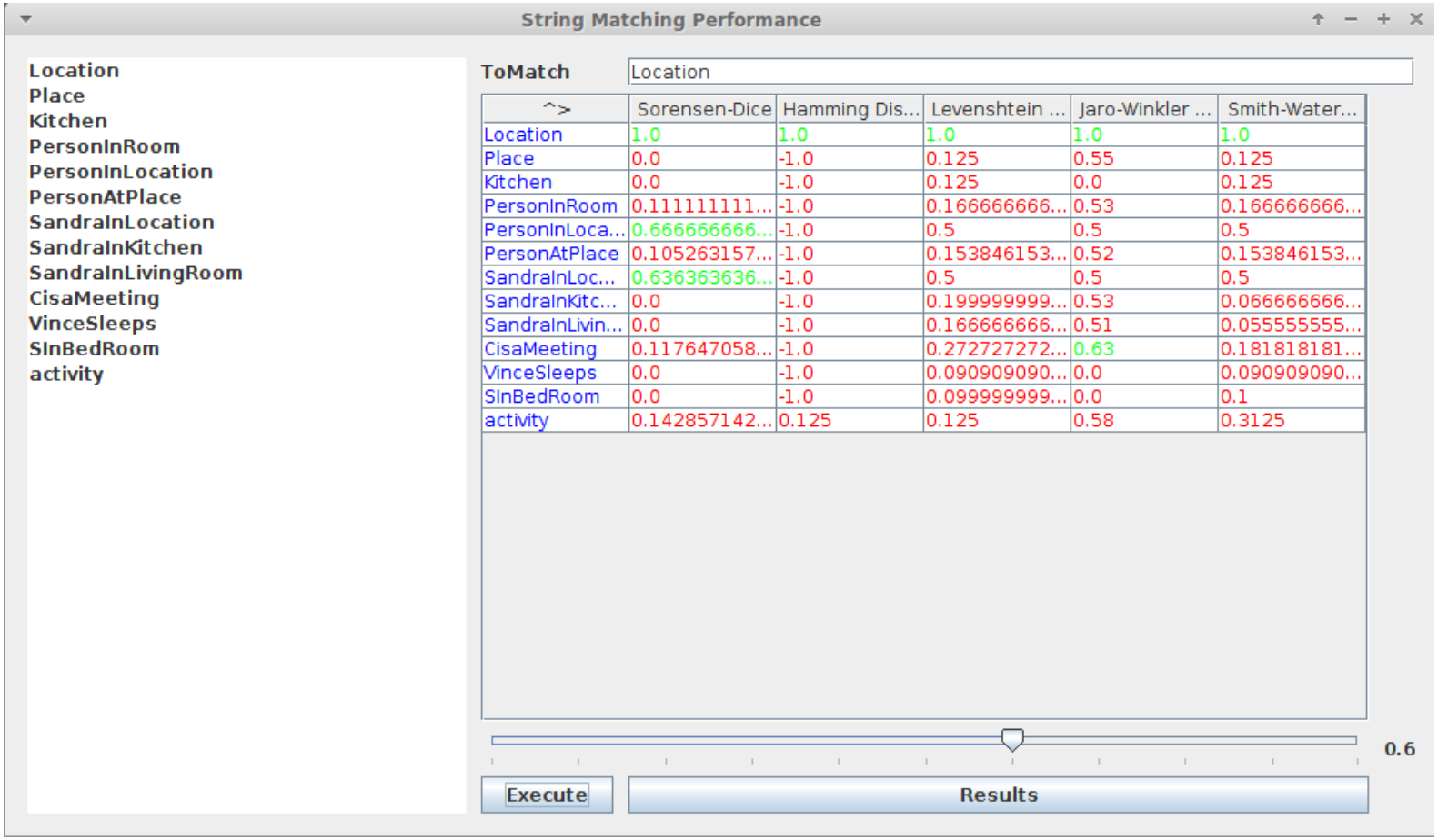

Figure 4.11: String Matching Algorithms Performance

Assigning *Kitch* to `X` and *Kitchen* to `Y`, we can split `X` and `Y` in bi-grams as follows: `ki|tc|h` (N_1) and `ki|tc|he|n` (N_2). Then `QS` equals:

$$\{\texttt{ki|tc|h}\}$$
$$\{\texttt{ki|tc|he|n}\}$$

$$QS = \frac{2x2}{2+3} = 0.8$$

This can also be seen in the table in Figure 4.11 on row `Kitchen` and column `Sorensen-Dice`. Hence, it looks like the Sorensen-Dice index is a suitable matching algorithm. Yet, it fails if we abbreviate by for example taking away all vowels from a word:

$$\{\texttt{k|tc|hn}\}$$
$$\{\texttt{ki|tc|he|n}\}$$

$$QS = \frac{2*1}{2+3} = 0.4$$

As such this algorithm might not be as feasible as it first looks like: we cannot ignore the case were users would abbreviate by removing vowels. Table 4.1 shows the results of each algorithm when matching needle *Ktchn* against each needle in the left column.

Hamming Distance

Another algorithm that looks interesting is the Hamming Distance between two character sequences, found by Richard W. Hamming [43]. This distance was initially used to compare error correction codes, but can be of help in comparing strings too. Simply explained, the Hamming Distance between two character sequences is the amount of substitutions that need to be performed in the first needle to obtain the second one. The lower the distance, the more similar the two needles are. Assume for example needle 1 equals *tower* and needle 2 *towel*. We only need to substitute the r with an l to obtain needle 2.

Although this looks a very useful algorithm, there are some significant issues with it for our particular case. Due to its nature, computing the Hamming Distance between two needles requires those needles to be of the same length. This is why needles with different lengths get a *-1* score in figures and tables showing the results of the string matching algorithms. Even with implementations not requiring both needles to be of equal length, Hamming

	Sorensen Dice	Hamming Distance	Levensht. Distance	Jaro Winkler	Smith Waterman
Location	0.0	-1.0	0.25	0.0	0.125
Place	0.0	0.0	0.0	0.47	0.2
Kitchen	0.4	-1.0	0.714	0.91	0.571
PersonInR...	0.0	-1.0	0.083	0.43	0.083
PersonInLoc...	0.0	-1.0	0.125	0.42	0.063
PersonAtP...	0.0	-1.0	0.076	0.43	0.077
SandraInLoc...	0.0	-1.0	0.125	0.42	0.063
SandraInKit...	0.222	-1.0	0.333	0.42	0.267
SandraInLiv...	0.0	-1.0	0.055	0.42	0.056
CisaMeeting	0.0	-1.0	0.090	0.43	0.091
VinceSleeps	0.0	-1.0	0.090	0.36	0.091
SInBedRoom	0.0	-1.0	0.0	0.43	0.1
activity	0.0	-1.0	0.125	0.38	0.125

Table 4.1: String matching algorithms, abbreviation by removing vowels Matching *"Ktchn"* with threshold 0.6

Distance is not useful because of the same reason as Sorensen-Dice. Specific types of abbreviations cause matching characters not to be aligned any longer, therefore implying a large Hamming Distance.

4.4.2 Levenshtein Distance

Looking for algorithms not depending on exact alignment of matching characters, we find the *Levenshtein Distance* between two inputs. In practice, this distance is equal to the amount of edits needed to get from needle 1 to needle 2. The difference to the Hamming Distance is that the input character sequences do not need to be aligned [44]. Hence, a difference in length between the input sequences will not cause any errors. The shorter sequence will be appended with white characters until the length of the longer sequence is reached. This does however mean that every alignment of a white character with a non-white one counts as an edit, hence increasing the total distance with 1.

To further clarify this, let us have a look at some examples:

$$\left.\begin{matrix}\texttt{tower}\\ \texttt{towel}\end{matrix}\right\} \implies LevenshteinDistance = 1$$

The Levenshtein distance between *tower* and *towel* is still 1 (equal to the Hamming Distance) as only one edit needs to be performed, which is substituting the r with an l.

$$\left.\begin{matrix}\texttt{loc}\\ \texttt{location}\end{matrix}\right\} \implies LevenshteinDistance = 5$$

Here, five whitespaces of *loc* need to be filled up with *ation* to become *location*, hence resulting in a distance of five.

As shown, the Levenshtein distance depends on the different lengths of the input character sequences. Hence, when longer sequences are abbreviated the distance with their abbreviation will be larger, assuming that short abbreviation(s) are used. This is a factor that must be kept into account. For example, the Levenshtein distance between *Loc* and *Location* is 5 as shown above. If we now take a longer term like for example *PersonInLocation* which can be abbreviated to *PersInLoc*, the Levenshtein distance equals:

$$\left.\begin{matrix}\texttt{persinloc}\\ \texttt{personinlocation}\end{matrix}\right\} \implies LevenshteinDistance = 7$$

The distance equals 7, as we need to add 7 characters, *on* and *ation*, to the abbreviation in order to reconstruct the original term. Differences in length between a term and its abbreviation should not make a difference in the final result the algorithm returns.

Having a closer look at Table 4.2 showing the results of the string matching performance program when matching *Loc*. One can observe a score of `0.375` at row `Location` and column `Levenshtein Distance`, resembling the equivalence between *Loc* and *Location* using the Levenshtein Distance algorithm. Results are on a scale between 0.0 and 1.0 and are computed as follows:

$$LevEquality = 1 - (LevDist/max(length(Needle1), length(Needle2))$$

In our specific case of *"Loc"* and *"Location"*, this implies:

$$LevEquality = 1 - (5/max(3, 8) = 1 - 0.625 = 0.375$$

In order to make the comparison fair, this same correction has been committed on results on other (similar) algorithms. Nevertheless, the difference in length between two strings has too much effect on the result, leaving the Levenshtein Distance algorithm infeasible for our case.

	Sorensen Dice	Hamming Distance	Levensht. Distance	Jaro Winkler	Smith Waterman
Location	0.444	-1.0	0.375	0.85	0.375
Place	0.0	-1.0	0.199	0.69	0.2
Kitchen	0.0	-1.0	0.143	0.49	0.143
PersonInR...	0.0	-1.0	0.083	0.0	0.083
PersonInLoc...	0.25	-1.0	0.188	0.0	0.188
PersonAtPl...	0.0	-1.0	0.153	0.0	0.077
SandraInLoc...	0.236	-1.0	0.188	0.0	0.188
SandraInK...	0.0	-1.0	0.066	0.0	0.067
SandraInLiv...	0.0	-1.0	0.111	0.0	0.056
CisaMeeting	0.0	-1.0	0.0	0.0	0.091
VinceSleeps	0.0	-1.0	0.090	0.47	0.091
SInBedRoom	0.0	-1.0	0.099	0.0	0.1
activity	0.0	-1.0	0.0	0.49	0.125

Table 4.2: String matching algorithms, Levenshtein distance
Matching *"Loc"* with threshold 0.6

4.4.3 Jaro-Winkler Distance

A more recent algorithm to compare similarity between two strings, is the Jaro-Winkler Distance [45], developed by Winkler based on the Jaro distance metric. The Jaro distance (d_j) of two needles is defined as follows[6]:

$$d_j = \begin{cases} 0 & \text{if } m \text{ is } 0 \\ \frac{1}{3}\left(\frac{m}{n_1} + \frac{m}{n_2} + \frac{m-t}{m}\right) & \text{else} \end{cases}$$

With m the number of matching characters
With t half the number of transpositions

The total amount of transpositions is given by the total amount of matching characters with different in-string positions divided by two. For example, in *Place* and *Plaec* all characters of the first string match those of the second However two letters, *c* and *e*, are on different positions, leading to two transpositions. Hence t equals 1.

Based on the Jaro distance (d_j), we can now define the Jaro-Winkler distance (d_w):

$$d_w = d_j + (lp(1 - d_j))$$

[6] `https://en.wikipedia.org/wiki/Jaro-Winkler_distance`

With d_j the Jaro distance between n_1 and n_2
With l a prefix n_1 and n_2 have, up to 4 characters, in common
With p a constant scaling factor, by default0.1

Going through some examples, this should become more clear to the reader. Having already computed `t`, we can now compute the Jaro Distance for *Place* and *Plaec*

$$m \neq 0 \implies d_j = \frac{1}{3}(\frac{5}{5} + \frac{5}{5} + \frac{5-1}{5}) = 0.93\ldots$$

Hence, the Jaro-Winkler distance equals:

$$d_w = 0.9333\ldots + (3x0.1(1 - 0.9333\ldots)) = 0.95$$

Please note that, in the case of Jaro-Winkler, the higher the result, the more similar the input strings are.

As one can observe, Jaro-Winkler is very tolerant when two characters change position. Yet, a better property is the fact that the algorithm looks for an equal prefix in both needles, the longer this prefix, the higher the final score. For our case of matching abbreviations, this is quite a beneficial feature as a lot of abbreviations are actually a prefix of the word they abbreviate.

Having a look back to the issues of the Levenshtein distance, more specifically the fact that the difference in length between two input strings had a significant effect on the end score, let us now check how Jaro-Winkler performs on the same example: *Loc* versus *Location*.

$$t = 0$$
$$m \neq 0 \implies d_j = \tfrac{1}{3}(\tfrac{3}{3} + \tfrac{3}{8} + \tfrac{3-0}{3}) = 0.79$$
$$d_w = 0.792 + 3x0.1(1 - 0.792) = 0.85$$

Having a look back at Table 4.2, the result of our manual computations are confirmed. As we still have a high score (0.85), Jaro-Winkler looks like an excellent algorithm to be used in our case. Though, as the reader might have spotted in that same Table 4.2, the score of *Loc* versus *Place* is quite high, even above our threshold. Although these are synonyms, they should not match using the Jaro-Winkler Distance alone, as the latter does not have any notion of linguistic equivalence. Therefore, this algorithm should be carefully used, especially when determining a threshold.

4.4.4 Smith-Waterman

Ending our search to an optimal algorithm, we have Smith-Waterman [46], originally developed to find an optimal alignment between two sequences, in

particular DNA-sequences.

Simply explained[7], Smith-Waterman builds a 2D matrix with each of the columns representing one of the matching sequences, the rows the other. Using a specific similarity function, the matrix is then filled with scores for each matching or mismatching row/columns character. This score also depends on already filled in neighbouring cells. When this initial matrix is constructed, the algorithm then loops through it from the highest until a null score is encountered (row 1 and column 1 are null rows), by following the path to the next highest score lower than the current score. In the end, this path then shows the optimal alignment of characters of the two input sequences. The output of the algorithm represents the amount of characters matching in both input strings using optimal alignment, minus the amount of mismatches. For example, matching *Kitchen* with *Ktchn* will result in :

$$\left.\begin{matrix}\texttt{Kitchen}\\ \texttt{K-tch-n}\end{matrix}\right\} = 2 - 1 + 6 - 1 + 2 = 8$$

In Table 4.1, the scores of Smith-Waterman have been brought to a scale of 0.0–1.0 by dividing the score by two times the length of the longest needle. In the case above, this would be:

$$\frac{Score}{2 * (Max(needle1.length, needle2.length))} = \frac{8}{2 * 7} = 0.571$$

Another example with *Location* and *Loc* results in the following:

$$\left.\begin{matrix}\texttt{Location}\\ \texttt{Loc-----}\end{matrix}\right\} = 6 - 0 = 6$$

The resulting score equals 6, not 1, as mismatches in a prefix or suffix are not kept into account. Scaling this result equals 0.375

Hence, the Smith-Waterman algorithm does not seem to be a good algorithm for our specific case. First of all, it is rather complex to understand. Second, the scores returned for matching `<term,abbreviation>` pairs are rather low. The most likely reason for this is that Smith-Waterman is designed to match DNA sequences, which do require exact alignment. As a result the algorithm is not really forgiving for abbreviations. Especially in the case where vowels are removed to abbreviate, the scores are really low. Therefore, in next sections and final implementations, the Jaro-Winkler distance will be used for string matching.

[7] `https://en.wikipedia.org/wiki/Smith-Waterman_algorithm`

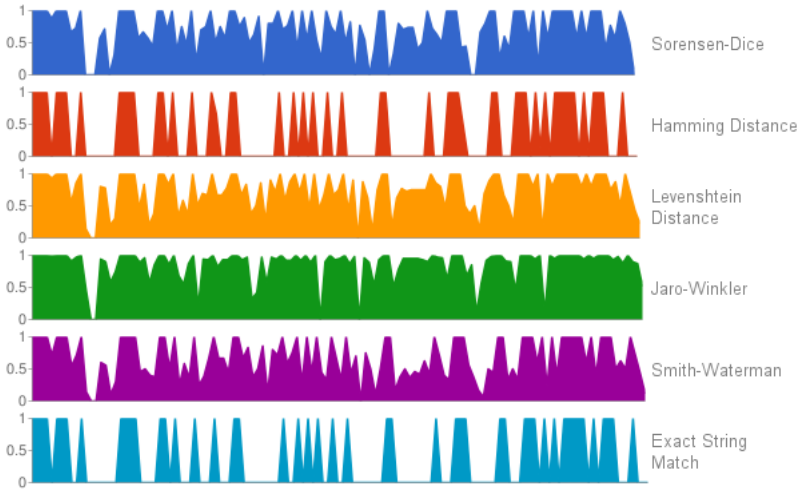

Figure 4.12: String Matching Algorithms: performance per pair

4.4.5 Overview and Technical Evaluation

Below in Table 4.3, a brief overview concerning each of the algorithms discussed is given.

Algorithm	Issues
Sorensen-Dice	Bad with infix removals
Hamming Distance	Inputs must be same length
Levenshtein Distance	Effect of different lengths
Jaro-Winkler Distance	-
Smit-Waterman	Strict (DNA sequencing)

Table 4.3: String Matching algorithms: overview

Previous sections have shown that the Jaro-Winkler distance algorithm seems to be the most optimal to be used in our case. However, only a limited amount of sample pairs, i.e. a term and one of its possible abbreviations, have been used to check how each of the algorithms computes a matching score. To make it possible to compare these algorithms against each other, a small technical evaluation has been committed using 126 sample pairs. Each of these pairs represents a home automation term, such as light and temperature, together with one of its possible abbreviations[8]. Although, not for

[8]These abbreviations have been retrieved by means of the online REST-based abbreviations service of *Stands4*. The latter can be found on `http://www.abbreviations.com/`

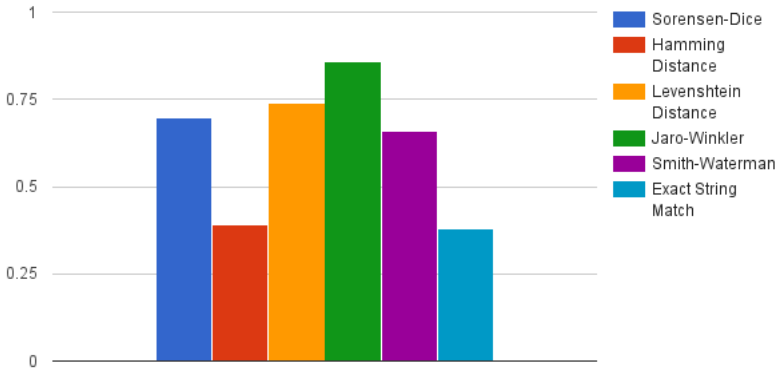

Figure 4.13: String Matching Algorithms: average score per algorithm

every term an abbreviation could be obtained. If this is the case, the term itself is used twice in a pair. For example, no abbreviation could be found for *rinse*, hence the resulting pair is: <*rinse*,*rinse*>. In case multiple abbreviations exist, only the one with the highest rating is considered. To optimize the matching process, special characters such as e.g. '.' in abbreviations have been removed and all input strings have been made case insensitive.

Having a look at Figure 4.12, one can observe 6 different graphs, one for each algorithm being discussed. The lowest graph represents the exact string matching algorithm, which returns 0 in case there is no match and 1 when there is a perfect match. Its performance will be used as a baseline to make a comparison with the others. Just like in the previous sections, the higher the matching score, on a scale from 0 to 1, the better. The ideal algorithm should return a score of 1.0 for each of the input pairs. Therefore, the more the graph is filled, the better the algorithm.

Focussing on each of the algorithms in Figure 4.12, one can immediately observe the strong similarity between the exact string matching algorithm and the Hamming Distance. This can be attributed to the fact that the Hamming Distance only returns a score different from 0 if both the term and its abbreviation are of the same length. This is almost only the case for sample pairs where term and abbreviation are equal, i.e. those terms for which no abbreviation was found, hence declaring the strong similarity with the exact string matching algorithm. Next, the Sorensen-Dice and Smith-Waterman algorithms appear to perform better on the test set as their graphs

'cover more area' compared to the Hamming-Distance. This is confirmed by Figure 4.13, showing the average matching score per algorithm on the same test set. Yet, both Sorensen-Dice and Smith-Waterman perform average compared to the other algorithms, which is most likely related to some particular issues discussed in previous subsections. The only two algorithms left are Levenshtein and Jaro-Winkler Distance. According to the average score shown on Figure 4.13, the Jaro-Winkler distance seems to score way better than the Levenshtein Distance. However, one has to keep in mind that these are only averages. Looking back to Figure 4.12, it looks like the Jaro-Winkler and Levenshtein distance agree quite well on what sample pairs match or do not match. Although, the Jaro-Winkler graph seems to reduce the difference between perfect and 'almost perfect' matches. Therefore, it might seem better to use the Levenshtein distance, as the effect changing the threshold above which a pair is considered to be a match is much bigger, returning more accurate results. Yet, as already discussed, the most significant reason why the score of 'almost perfect' matches using Jaro-Winkler is higher, is because the difference in length of a term does affect the Levenshtein Distance. Considering abbreviations, this is rather disadvantageous. As a result, the Jaro-Winkler distance is used to match building block type names (see later) in the implementation of the extension.

4.4.6 Synonyms and Program Flow

When matching building blocks by name, we have to keep potential synonyms for those names into account. Currently, the string matching program does this by calling a REST-API, provided by Big-Huge Labs[9], returning synonyms for a specific word in JSON format. On a high level, the program flow looks as shown in Figure 4.14. Initially we have a needle `n1`, i.e. the name of a building block to import, that needs to be matched against a list of needles $\texttt{n2}_\texttt{i}$, i.e. the names of already known building blocks. Hence, for each needle $\texttt{n2}_\texttt{i}$ the program encounters, the Jaro-Winkler Distance algorithm is called to return a score representing the distance between `n1` and $\texttt{n2}_\texttt{i}$. The resulting scores are then filtered. Each $\texttt{n2}_\texttt{i}$ with a score lower than the user-defined threshold is removed. The $\texttt{n2}_\texttt{i}$ needles with a score above the threshold are then sent (*collect1*) to be shown in the results. Concerning the synonyms processing part of the program, the REST-API of Big Huge Labs is called with `n1` as an argument. From the returned JSON, the synonyms for `n1` are retrieved. After this, the intersection between the retrieved synonyms and the list of needles `n2` is taken. We are only interested in synonyms with

[9] `http://words.bighugelabs.com`

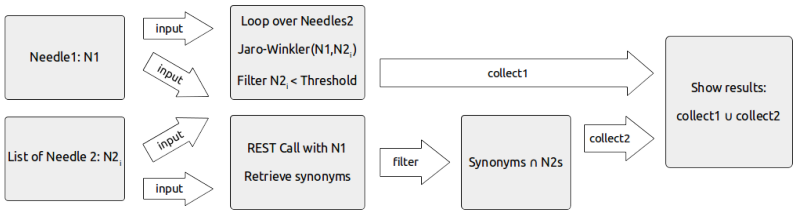

Figure 4.14: String matching with synonyms flow

	Sorensen Dice	Hamming Distance	Levensh... Distance	Jaro Winkler	Smith Waterman
Location	0.0	-1.0	0.125	0.55	0.125
Place	1.0	1.0	1.0	1.0	1.0
Kitchen	0.0	-1.0	0.285	0.56	0.214
PersonInR...	0.0	-1.0	0.083	0.1	0.083
PersonInLoc...	0.0	-1.0	0.125	0.1	0.063
PersonAtPl...	0.5	-1.0	0.385	0.1	0.385
SandraInLoc...	0.0	-1.0	0.125	0.42	0.063
SandraInK...	0.0	-1.0	0.199	0.42	0.1
SandraInLiv...	0.0	-1.0	0.055	0.42	0.056
CisaMeeting	0.0	-1.0	0.181	0.0	0.091
VinceSleeps	0.143	-1.0	0.181	0.53	0.181
SInBedRoom	0.0	-1.0	0.099	0.43	0.1
activity	0.181	-1.0	0.0	0.55	0.25
position	0.0	-1.0	0.0	0.44	0.125

Table 4.4: Synonym-based String Matching. Matching *Place* with threshold equal to 0.5

the same name of building blocks we already know. These results are then collected (*collect2*) and sent to be shown in the results, which is the union of *collect1* and *collect2* in Figure 4.14. To further clarify this, an example is given. Restarting our string matching performance algorithm, we added a new `n2` to the list: *"position"* and lowered the threshold to 0.5. When now entering *"Place"* in the `ToMatch` field, the input field for `n1` and pressing the `Execute` button, the resulting scores shown are depicted in Table 4.4.
One can observe that five `n2` string are above the threshold of 0.5 (coloured green). However, *position* is not a part of it. When now pressing the `Results` button, we get the matching `n2` needles resulting from the Jaro-Winkler algorithm, as well as those matching one of the synonyms in the retrieved

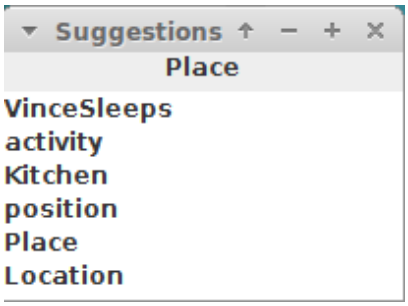

Figure 4.15: Matching needles including synonyms

synonyms list. In case of `n1` equal to *Place*, this synonym list contains following synonyms (limited to 9 elements):

spot property stead position lieu shoes home post berth

As *position* is part of the list of synonyms and the list of `n2` needles, it should be added to the final list of matching needles, as shown in Figure 4.15. Please mind that we lowered the threshold to 0.5, which might result in non-usable matches.

4.5 Use Case: Exporting and Importing a Rule

Having discussed how we can semantically match building blocks of rules and events, let us now have a look on how we can extend CMT to export and import rules and events, with user involvement to prevent losing intelligibility. As the implementation for context rules and custom events is completely analogue, we will only discuss the case where custom events can be made, as it contains some interesting side cases. In order to make this rather complex part more understandable for the reader, an example use case will be given along the more detailed explanation. The implementation code can be found in Appendix A.

Assume that Alice is invited at Bob's home for a cup of tea. When sitting in the sofa, they discuss the latest amazing trends in home automation technology. For an unknown reason to Alice, Bob does not seem to be in a good mood. Alice spots bags under his eyes and asks him if everything is all right. Bob answers that he has some difficulties sleeping, mostly caused by a variety of noises disturbing him. Bob says that he already created some rules to turn off noisy devices, roll down the shutters to prevent neighbouring noise from coming into his bedroom. Yet, he cannot find a way to tell is his system disturbing noise is around during his sleep so that it can take proper action. Alice smiles and tells Bob that she used to have the same problem once, but she resolved it by installing a noise sensor in her room and creating a custom event rule for it. She offers to help Bob installing it too and Bob of course agrees. He is already dreaming of some good nights sleep. No sooner said than done, Alice and Bob went to a local do-it-yourself store and bought an ambient noise sensor, compatible with CMT, such as the one shown in Figure 4.16. Once back home, Bob installs the sensor discretely behind the armature of his bedroom light and adds it to his CMT instance[10]. Alice then shares the rule, returning an event called *DisturbingNoise*, to Bob. Bob can then later integrate in the rules he made to turn off selected devices and/or roll down the shutters.

Below, a detailed step-by-step explanation is given on how the rule sharing mechanism exactly works. The explanation is limited to rules declaring custom events. The case for rules declaring actions is completely analogue.

[10]More information about how to add new hardware to a CMT instance can be found by the original developer of it [12]

Figure 4.16: Ambient noise sensor

4.5.1 Exporting a custom event

As mentioned before, Alice wants to share a rule declaring the custom made event *DisturbingNoise*. In order to do this, she opens a compatible front-end CMT app on a client device (such as e.g. a tablet or smartphone) connected to her own CMT instance. In this case the CMT server that controls her smart home. In the app, she can then navigate to the sharing environment allowing to share her rules with others. As shown in Figure 4.17, Alice selects the rule she wants to share and shares it to Bob. Depending on how the client-side app is implemented, different strategies can be used on how to detect and select users to share a rule to. One such strategy can, for example, be that the client-side app connects to every CMT it has in (wireless) range and shows the clients owning it/being connected to it, e.g. via WiFi Direct [3]. Another option is to use a WPAN (Wireless Personal Area Network) approach, such as for example the popular Zigbee [47] and Z-wave [48], but even more approaches with different architectures exist [49].

Once Alice presses the share button, a whole process starts, as shown in Figure 4.18. The first step is to send the rule Alice wants to share to her own CMT in order to convert it to a format enabling Bob's system to import it.

Let us have a look to the technical aspects of this first step. Sharing a rule with another CMT instance means that we need to agree on a general format by which we will exchange data, i.e. rules, templates and their internal structures. At first, this might seem to be quite complex a task as we need to encode every specific type declared in CMT to an intermediary format. Fortunately, there already exist numerous methods converting CMT types to JSON, as this is needed to communicate with the client. For example, if

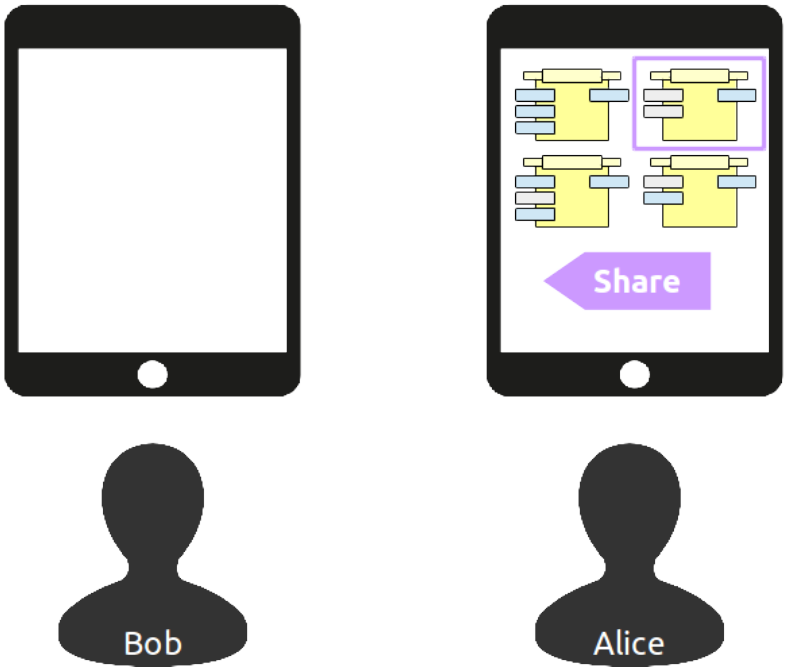

Figure 4.17: Alice selects a rule to share with Bob

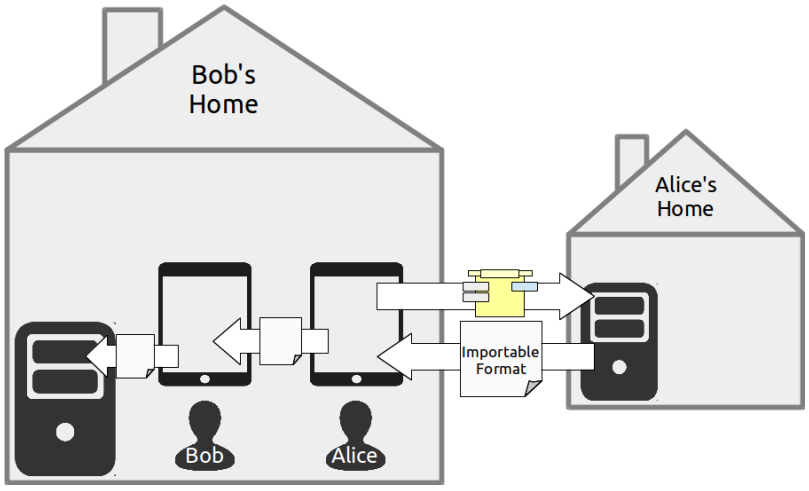

Figure 4.18: Converting the chosen rule to an importable format and sending it to Bob

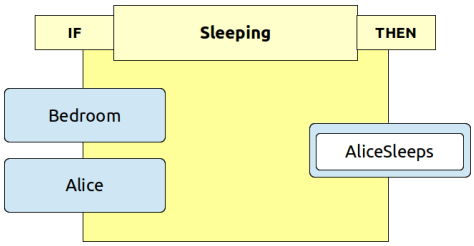

Figure 4.19: Rule definition of the `AliceSleeps` custom event

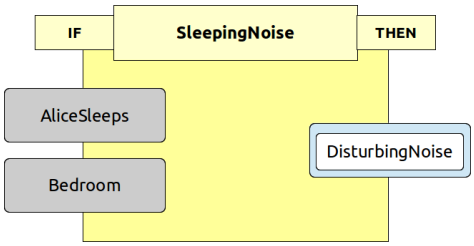

Figure 4.20: Rule definition of the *DisturbingNoise* custom event

Alice creates a rule on her tablet, that rule need to be inserted into her CMT server. The communication with that server is done by means of JSON. As such, we can reuse these methods to convert rules and templates to JSON and then send the resulting JSON file over to the CMT instance we want to share our event with, in this particular case Bob's server.

Yet, this is not as simple as it may look like. One of the main features of CMT is the fact that a custom event can be reused in a rule creating a new custom event. This is also the case for the rule Alice wants to share with Bob. One can observe that the rule shown in Figure 4.20 takes two input blocks, `AliceSleeps` and `Bedroom`[11]. Of these two input blocks, `AliceSleeps` is a reused custom event declared in Figure 4.19. As such, if we would naively use the built-in conversion functionality, only `DisturbingNoise` will be converted to JSON. When Bob's CMT server then starts analysing the input-blocks of `DisturbingNoise`, encountering `AliceSleeps`, an error will occur as it does not know when `AliceSleeps` is supposed to evaluate to true. Therefore, it is required to include `AliceSleeps` in the conversion process of `DisturbingNoise`. In case `AliceSleeps` would also contain any custom input-blocks (not the case in this particular example), those should also be converted, hence creating a recursive algorithm looping through the declarations of custom events.

Converting the Template: Alice's CMT server

Let us have a look to what the conversion process of a custom event is about. If Alice's server would only consider the rule declaring `DisturbingNoise`, it would not be able to encode under what specific circumstances `DisturbingNoise` is true. This is because every rule is based on a template. Such a template defines the real semantic value behind a rule. Consider for example the

[11] Further explanations about how these blocks are assigned will be given later on

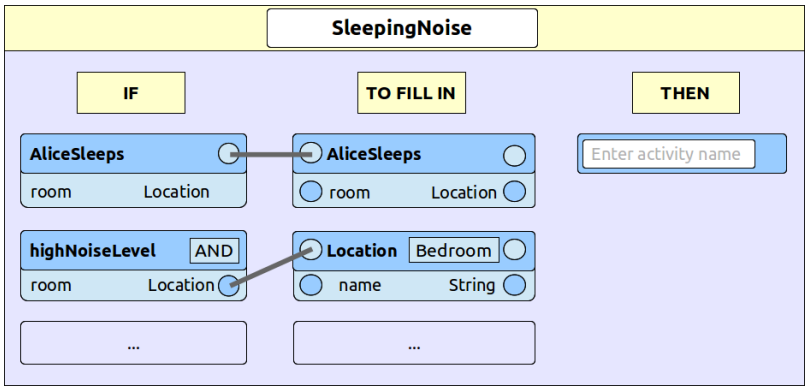

Figure 4.21: Template of rule `DisturbingNoise`

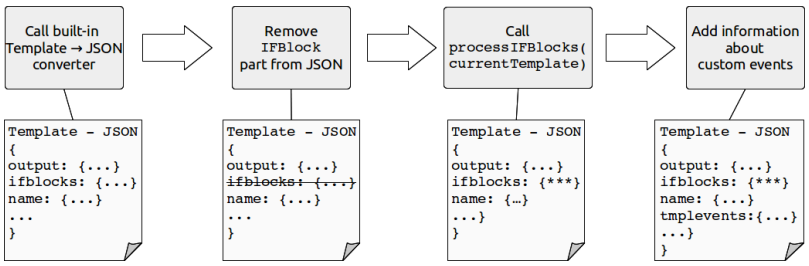

Figure 4.22: Overview of the `prepareTemplateSkeletonJSON` function

template of `DisturbingNoise` shown in Figure 4.21. It takes two IF-blocks. One is the custom event `AliceSleeps`, which is, by design of CMT, automatically connected to its corresponding input block (2nd column). The second IF-block is a function called `highNoiseLevel`, detecting whether or not the noise level in a particular location is too high in case someone wants to sleep. Its only parameter *room* of type `Location` is then connected to a `Location` typed input block. Yet, Alice defined that this location should always be her bedroom. As a result, when creating a rule based on this template, the user is not required to assign input blocks to it, represented by the grey colour of the input blocks in Figure 4.20. For every rule encountered, the template on which it is built needs to be converted too. Converting a template can easily be done using the built-in `fromTemplateToJSON(Template template)` converter method. Yet, we need to keep custom events into account. In code, these can be found in the `IFBlock` section of the template. Therefore, first the template to JSON converter is called of which the IF-block sections is then removed out of the JSON allowing us to fill it with our own custom IF-block format.

From now on, we can manually process the IF-blocks of the template, check them on custom events and add extra attributes to allowing Bob's CMT server to import the template. This is done by the `processIFBlocks` method. After this, some extra information about IF-blocks containing event building blocks is added to the JSON. The reason behind this will become clear to the reader when discussing the import algorithm. In the end, the resulting JSON is ready to be sent to another CMT-instance, here Bob's CMT server. The flow of this first function explained is shown in Figure 4.22.

Processing the IF-blocks of a template

The next step is to process the actual IF-blocks. Two IF-blocks can be identified in Figure 4.21: the `AliceSleeps` custom event and `highNoiseLevel` function. The JSON to be sent to Bob, being partially constructed in the calling function, `prepareTemplateSkeletonJSON` (see previous), is further completed here. After having retrieved the IF-blocks of the template in conversion, `processIFBlocks` starts to loop over the IF-blocks. The general flow can be seen in Figure 4.23. First the index, representing the relative 'position' of the IF-block in the templates is encoded, just like the original converter would do. In this particular case `AliceSleeps` would have 0 as index, `highNoiseLevel` 1. Then, depending on the type of IF-block, the corresponding built-in converters are called: `fromFactTypeToJSON` for events, `fromFunctionToJSON` for functions. Until now, we have mostly been copying the functionality of the original converter. From this point on, this changes. After doing the general conversion of the current IF-block, the bindings will be checked. Based only on the bindings of an IF-block, one can determine whether or not a custom event is involved and a recursive process should therefore be started. The function `processBindings` takes care of this and is therefore called with each of the IF-blocks as an argument.

Processing the Bindings of an IF-block

Before diving into the implementation on how the bindings of an IF-block are processed, let us first try to reason when we should provide extra information concerning a custom event enabling the importing CMT instance to import that custom event. Looking back to the template of the rule `DisturbingNoise`, Alice wants to share with Bob as shown in Figure 4.21. In total, there exist three types of IF-blocks (leftmost column in Figure 4.21) a template can take. First, there are functions, second and third are custom and non-custom events. Every (formal) parameter of a function in its

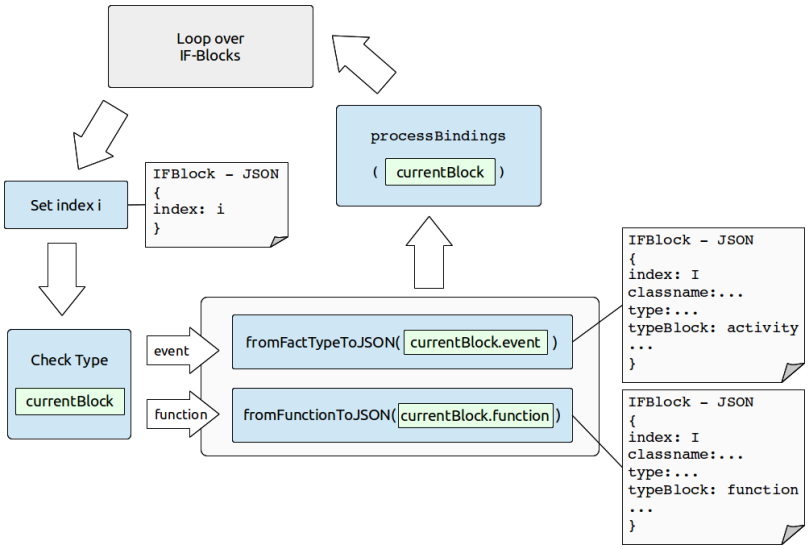

Figure 4.23: Overview of the `processIFBlocks` function

IF-block[12] must be bound to an input block (second column) or one of the parameters of an input block. This binding represents the assignment of a formal parameter on the IF-side (first column) to an actual parameter on the input-side (second column). Parameters can be of meta type `FactType` in case the binding at the input-side is represented by a non-specific fact. Considering the `SleepingNoise` template, there is no such binding, but if one has look to the template, shown in Figure 4.24 of the `AliceSleeps` event, previously shown in Figure 4.19, one can observe that both `Location` and `Person` blocks at the input-side do not have a specific fact assigned, i.e. their drop down lists are set to "*Any*", meaning that the end user has to assign a specific fact to them when using the template. As such, all parameters in template `Sleeping` are assigned to `FactTypes`. In case there is a binding from an IF-block to a specified fact on the input-side, as is the case for parameter *room*, bound to *Bedroom* in input-block `Location`, shown in Figure 4.21, that parameter is said to be of meta-type `Fact`. In case a custom event such as `AliceSleeps` is being used, `FactType` is also used, as determined by the design of CMT. One might ask why this is important to understand the mechanism to export a template. As previously mentioned, all parameters on the IF-side must be bound to an actual value, represented by a block on the input side. The expert user creating the template is responsible to fulfil this requirement. However in case a custom event, such as `AliceSleeps`, is dragged as an IF-block into the template editor, the CMT

[12]Functions can only be used as IF-blocks, never as input blocks

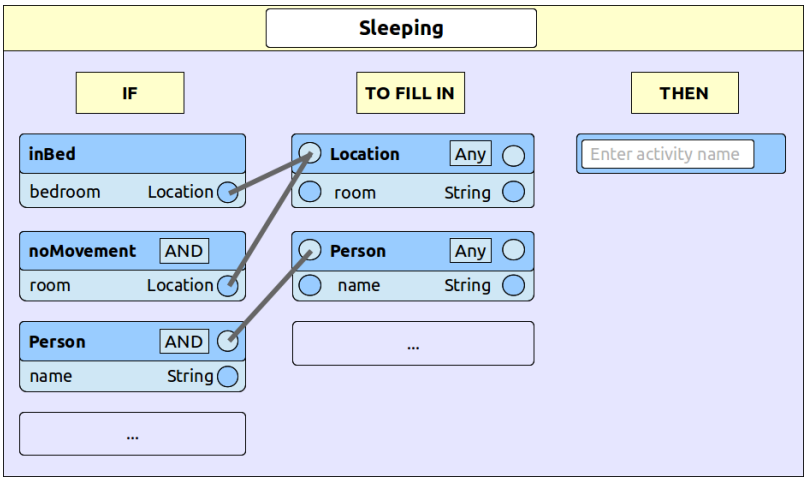

Figure 4.24: Template of rule creating event `AliceSleeps`

GUI will automatically create a binding to itself on the input side. For example in Figure 4.21, custom event `AliceSleeps` on the If-side is bound to `AliceSleeps` (i.e. itself) on the input-side. In other words, an expert user cannot change the bindings of a custom event (e.g. bind to another custom event) as this would not make any sense: a custom event is an actual parameter on its own. Hence, this means that the end user cannot change this either, explaining why `AliceSleeps` is greyed out in Figure 4.20. Yet, this does not mean that user could not use the parameters of an event. An expert user could for example decide to use the *room* parameter in the input-side `AliceSleeps` custom event to be bound to the formal *room* parameter of function `highNoiseLevel` in Figure 4.21.

Having this said, how can we now detect custom events in an efficient way without encoding them twice? The best thing to do is to loop over the end-bindings of the IF-blocks, leading to the input blocks. This covers both the case of custom events as IF-blocks, as these always have an end-binding to themselves, as well as any other IF-blocks that could have a binding to a custom event, the latter then required to be encoded too. As will become clear later on, this is also the most efficient way considering how CMT is implemented. Let us now have a look to how this is actually implemented by the `processBindings` IF-block. A flowchart is shown in Figure 4.25

Each binding encountered is converted to JSON using the built-in converter. As can be seen in Figure 4.25, a binding as object consists of two parts: first the `startBinding` representing the IF-block where the bindings starts and

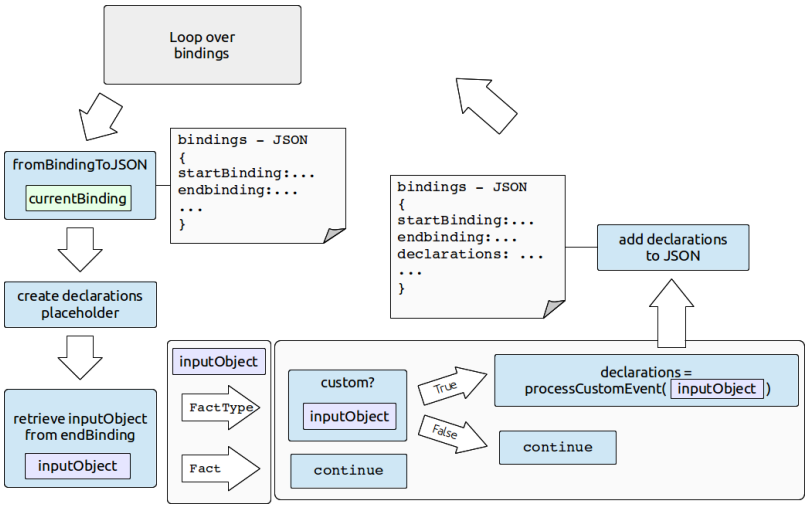

Figure 4.25: Overview of the `processBindings` method

second the `endBinding` representing the input-block where the binding ends. Next, a JSON place holder for declarations is created. In case a custom event is encountered at the `endBinding`, these declarations will contain the to JSON encoded template creating it, hence representing the recursive character of the export algorithm. More information about this will be given later on. Next, the `endBinding` part of the current binding (coloured in green) is retrieved, containing the actual object, called `inputObject` (coloured in grey-purple), to which the IF-block is bound. Applying this to the example template *SleepingNoise* in Figure 4.21, the first binding encountered would be the one between the two `AliceSleeps` custom events. As such, the input object on the end-binding will be of type `FactType`. Back to our flowchart in Figure 4.25, a check is applied on `AliceSleeps` to determine whether or not a custom event is involved. If this is the case, we will have to encode the template behind this custom event as mentioned earlier. In the current example, this is indeed the case as `AliceSleeps` is a custom event. Therefore, the appropriate function `processCustomEvent` is called in order to handle the recursive call to encode the template of the custom event. After this is done, this method returns the encoded template which is then fit into the declarations of the current binding JSON. In this current case, this means that template `Sleeping` will be encoded in the declarations of the binding between the two `AliceSleeps` custom events, as this template is responsible for `AliceSleeps`.

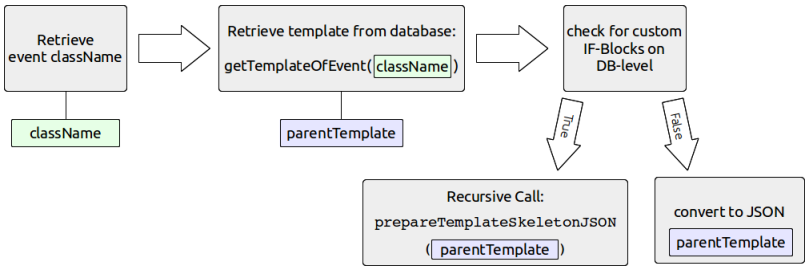

Figure 4.26: Overview of the `processCustomEvent` method

Processing custom events

In practice, encoding a custom event means looking up the template responsible for it (i.e. the 'parent template' of the custom event) and, recursively, encoding that one too. This also means that we have to check that parent template on custom events and if necessary, start a new recursive step. Figure 4.26 clarifies this further. Starting from a custom event, it retrieves its `className` which is just a field in a custom event object. Using that `className`, the database is queried to obtain the template responsible for our custom event. Mapping this onto our example in Figures 4.20 and 4.21, we first retrieve the `className` of `AliceSleeps`, which is "`AliceSleeps`". Using that `className`, we can now retrieve the template responsible, which is `Sleeping` as shown in Figure 4.24. In Figure 4.26, it is called `parentTemplate`, depicted with a grey-purple box. Using a built-in functionality of the database layer we can quickly check whether or not this `parentTemplate` contains any custom events itself. Using this database functionality prevents loading the whole template and, later on, the building blocks it is constructed by into memory and as such slowing down the program. Yet, in case the `parentTemplate` does contain at least one custom event, a recursive call to `prepareTemplateSkeletonJSON` is launched, restarting the program flow to beginning with this `parentTemplate` as an argument. `parentTemplate` will then be converted to JSON in the first step already. Yet, if `parentTemplate` does not contain any custom events, we still need to convert it to JSON and put it in our declarations. In the example given, the `Sleeping` template does not contain any custom events, as such no recursive call is necessary.

Backtracking and result

After having recursively descended to a template containing no more custom events, we can start to backtrack and build up our JSON export format. Assuming a template with a custom event and a function as IF-blocks, this

looks visually as in Figure 4.27. Note that this Figure has been limited to the parts necessary in order to understand how the import and export mechanism works.

A template consists of three major aspects. First of all we have the *output*-part, representing what event the template actually generates. This includes the name of that event and its bindings (e.g. parameters the expert user wants to make available when an event created using this template is reused in another one).

Next comes the most important part: the *IF-blocks*. Depending on the type of IF-block, there are some small differences in the structure of it. Let us start with an IF-block representing an event (in Figure 4.27 `eventBlock`). First of all, we have the representation of the event itself (*event*). This includes its class name (e.g. `AliceSleeps`), its type which will equal `activity` in case of an template to create events with (`TemplateHA`) and an array of `fields` representing the parameter of the event that have been made available by the expert user (or the programmer in case of a non-custom event). Second, we have `JSONObject` representing the type the IF-block holds (`typeBlock`). In this particular case, `typeBlock` will equal `activity` as it holds an event.

The third and last part discussed concerning an IF-block, are its bindings. Each binding object consist of a `startbinding`, representing from what type of block or parameter the binding starts. If it is just a regular binding starting at an IF-block (and p.e. going to an input-block), the type of `startbinding` will equal `BindingIF`, which will always be the case considering IF-blocks. More important is to know on what block the binding ends: `endbinding` provides a full description of that block, which is held in its `inputObject`. It consists of the `className` of that block (e.g. `AliceSleeps`, but now as input-block) and its fields (e.g. *room* of type `Location`). In case the IF-block holds a custom event, a `declarations JSONArray` will always be provided containing the `TemplateHA` responsible for that custom event. Looking to an IF-block holding a function, the only (minor) differences are mostly terminology. `className` becomes `methodName` and *fields* becomes `parameters`. Further on, depending on the `endbinding`, the binding of a *functionBlock* do not necessarily need to have declarations, as they are not always bound to a custom event.

To finish, the JSON-description keeps some general information about the template itself. These are i.e. the *name* of the template, a user-defined cate-

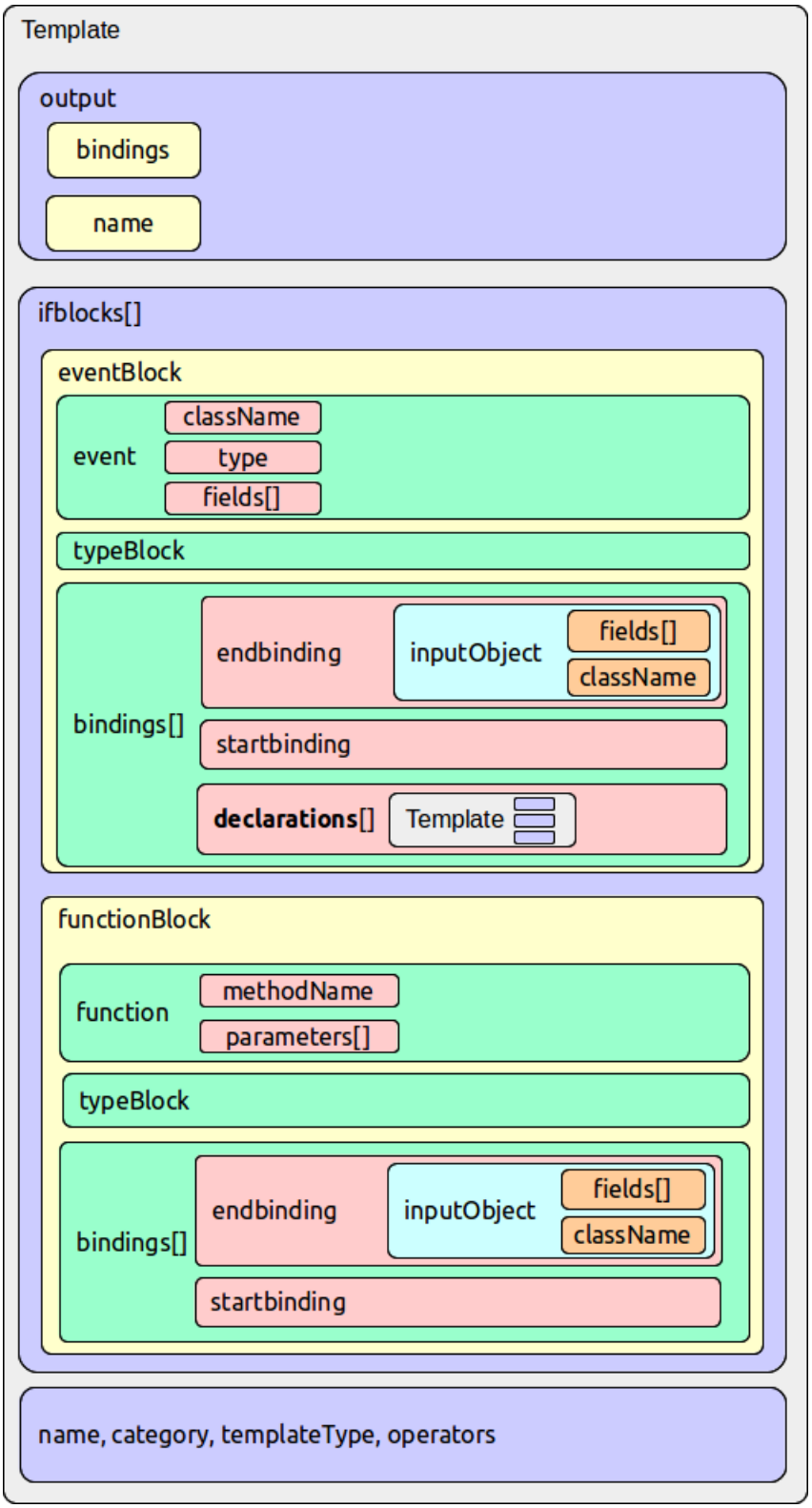

Figure 4.27: Example export format of a template with event and function as IF-blocks

gory (such as home, work or sports), the type of the template (`TemplateHA` or `TemplateActions` and the operators operating between the IF-blocks (`AND, OR, NOT`). Based on this description, the importing algorithm (on Bob's server) will be able to correctly import a building block without losing too much intelligibility as is discussed in the next section.

4.5.2 Importing an event

Importing a custom event from another user in practice means importing its template first and only then importing the shared event itself. Given the JSON we got from the exporting CMT instance, in our example, Alice's CMT server, there exist two programmatic approaches to import it: either looping through it in a bottom-up or in a top down way.

Importing a template means importing its building blocks first and then installing it into our system. Yet, as mentioned before, one or more of these building blocks can be custom events which need to be imported too if we want to import the template. In this way, a bottom up approach looks the best solution: for every custom event we encounter, we recursively go down to the deepest point in the declarations. Only during backtracking we resolve each template: the "deepest" template can automatically be resolved as no custom events are part of its building blocks; the template one level higher can then be resolved as we just resolved the custom event(s) it has in its building blocks. On a high level, this would visually look as shown in Figure 4.28: To resolve template A, we loop all the way down its declarations until we reach template D. As D does only contain 'basic' building blocks (coloured grey in Figure 4.28), we can resolve it. Hence, the first building block of B (coloured magenta) is resolved. As the other building blocks of B are not custom events, we can resolve B and therefore the first building block of template A (turquoise). Although, as the second building block of A (blue) is a custom event, we need to loop down its declarations, starting a new recursive process. Once all custom building blocks of A are resolved, template A as well as the custom event (orange) it generates can be imported. Having this said, the bottom-up approach looks the ideal way to solve our problem: it is not that difficult to program and works in an efficient way without many side cases. Yet, it has one major disadvantage.

As explained earlier, we want to involve the user in the resolving process, in our use case this is Bob. Assume that Bob already has a rule generating the custom event `BobSleeps`. When his server now tries to import the `AliceSleeps` custom event, which is semantically equal to `BobSleeps`, it

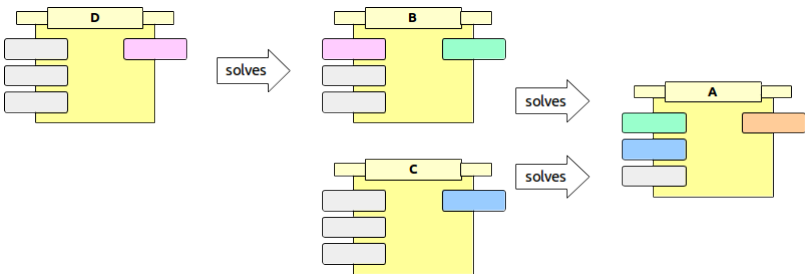

Figure 4.28: Resolving templates using the bottom-up approach

should be possible to use `BobSleeps` instead of importing `AliceSleeps` with all the overhead coming together with it. Yet, only Bob can decide to do so, as CMT by design cannot decide whether or not a local custom event (`BobSleeps`) is semantically equal to an remote one (`AliceSleeps`). Later in this section, more explanation will be given on how a user, Bob, can decide between importing a foreign custom event or integrate a local one in a rule being imported. This is functionality is not limited to events only, but can also be applied on `Facttype`s and `Fact`s. Yet, as shown before, we need to make it easier for Bob to choose the right local building block out of all the building blocks in his servers' database. This will be done by using the Jaro-Winkler fuzzy string matching algorithm on the `className` of the building blocks, augmented with functionality to also look out for potential synonyms. Though, this means that it would be inefficient to use the bottom-up approach explained before. Assume, for example, that Bob decides to use his own custom event `BobSleeps` instead of `AliceSleeps` provided by the rule Alice shared with him. This means that there is no longer need for Bob's server to analyse the template behind `AliceSleeps`, hence saving memory, computation power and complexity compared to the bottom-up approach, as this particular process can be made iterative when computing it in a top-down way.

Importing a template

Having determined the approach on how to import and resolve templates, it is time to have a look at the actual implementation of the import algorithm, from the viewpoint of Bob's CMT server.

To handle the communication with potential clients, the already built-in `CMTRest` class is being reused. Considering importing a rule, a specific endpoint is provided accepting inbound POST requests with the shared rule in importable format using JSON. In other words, Alice has to send a POST re-

quest to Bob's server with the rule she wants to share as payload. After initial communication handling, this request is then relayed to the `importTemplate` method of which a snippet can be seen below. The argument *jTemplate* is the JSON representation of the rule Alice is sharing with Bob.

```
public UserDecisionRest clientRest;
private ArrayList<TemplateSuggestions> suggestionsPool;

public void importTemplate(JSONObject jTemplate) {
        suggestionsPool = new ArrayList<>();
        importTemplateRec(jTemplate, 0);
        clientRest.setSuggestionsPool(suggestionsPool);
        clientRest.sendToClient();
    }
```

First, a so-called *suggestionsPool* is created: as the reader will see later on, for every IF-block of every template encountered in the top-down recursion, suggestions based on `className` are offered to the user, requesting him to make a decision. As such, every template has its own corresponding `TemplateSuggestions` object, keeping track of these suggestions but also containing some extra information about the template necessary for the end user to make their final decision. This will become more clear later on. After this initial 'bookkeeping', the recursion down the templates and their declaration is started using the `importTemplateRec` method (line 6). When the recursion is finished, the suggestions added to the pool are sent to the client using the `UserDecisionRest` class (as discussed later). Although not necessary at the moment, this class can be used in the future to keep state between the tasks of sending the suggestions to the client and receiving the actual decisions of the end user. For now, it operates as a library preparing suggestions to be send and dispatching end user decisions when they come in.

Recursively going through a template and its declarations

Having explained this initial bookkeeping, let us have a closer look on how the import process on Bob's server works, as shown in Figure 4.29. Due to the fact that multiple IF-blocks can be connected to the same input-block, e.g. in the `Sleeping` template depicted by Figure 4.24 where both the parameter of function `inBed` and `noMovement` are bound to the `Location` input-block, we need to keep track of which input-blocks are already resolved (by the user). This is done using a simple Java `ArrayList` called `resolvedIdx`, storing `Integer`s representing the indexes of resolved input-blocks. Next the template on current recursion level is parsed from JSON back to its Java representation and then used to create a object, `TemplateSuggestions`, storing a list of all suggestions for all IF-blocks in the template. The importing algorithm will constantly need to read, change and replace input-blocks in the template

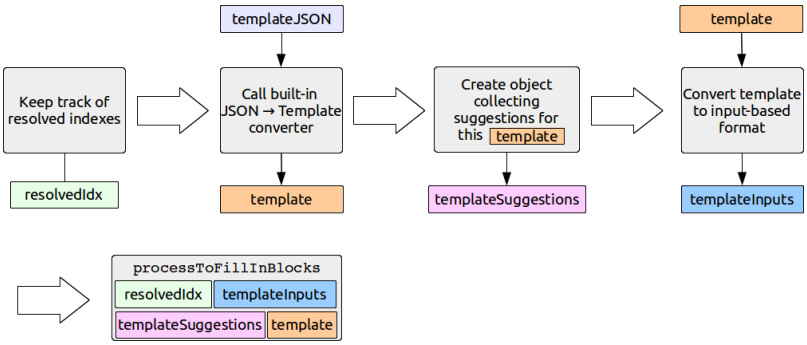

Figure 4.29: Overview of the `importTemplateRec` method

according to Bob's decisions. Yet, using the current representation of a template, it is rather complex to 'reach' the desired input-block: one first needs to find the right IF-block, then the right binding, get into the `endBinding` and cast to the right meta-type of the block residing at the `endBinding`. Even with additional abstraction, algorithms interacting with input-block would just become too inefficient and too complex to understand. Therefore, the template is converted to a Java `HashMap`, called `templateInputs`, mapping indexes of input-blocks onto themselves. Whenever an input-block needs to be changed, added or replaced, only the mapping from a index onto a block needs to be adapted. After the template is completely processed, `templateInputs` is converted back to template representation, including the modifications applied.

Just like in the export algorithm, one only needs to analyse the input-blocks in order to know if custom events are present or not. Considering the import algorithm, this remains the same. Even more, it is not necessary to let the user (Bob) change the IF-blocks of a template he wants to import. This would first of all change the template, which does not make sense a Bob could then better import another one. Second, every IF-block is bound to an input-block. Hence, the algorithm just has to make sure that a compatible type of input-block is bound to the right IF-block, or parameter of an IF-block. All of this functionality is further handled by `processToFillInBlocks`[13].

Processing the input-blocks

In order to provide Bob's client device with information and suggestions concerning input-blocks, he has to make a decision about, the algorithm needs to distinguish between the three types of input-blocks existing. Therefore, from

[13] *toFillInBlock* is a synonym for input-block

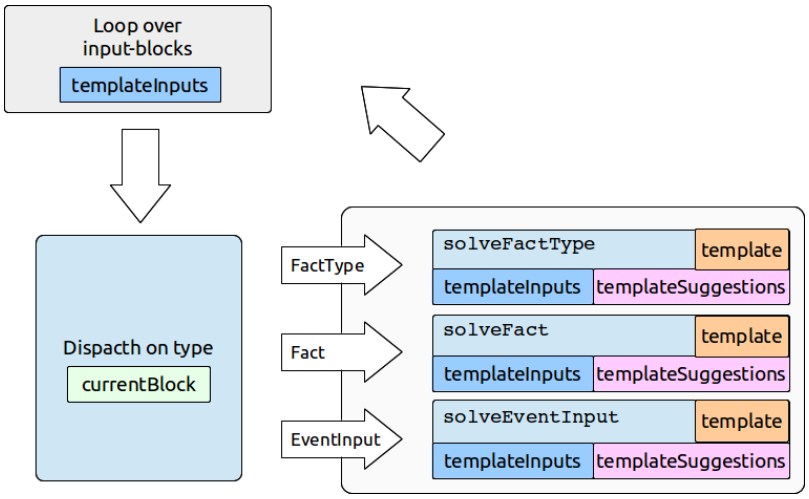

Figure 4.30: Overview of the `processToFillInBlocks` method

a technical point of view `processToFillInBlocks`, shown in Figure 4.30, is a dispatching method calling the right function depending on the type of input-block encountered. Having a closer look to Figure 4.30, one can see that `processToFillInBlocks` loops over the input-blocks residing in the `HashMap` representation of the template being processed. For each input-block encountered, depicted as `currentBlock`, one out of three functions is called. `solveFactType` in case `currentBlock` is of type `FactType`, `solveFact` in case of a `Fact` and `solveEventInput` otherwise. Identifying the actual type of `currentBlock` is done by downcasting it from `IFactType` to `FactType`, `Fact` or `EventInput`.

Mapping this to Bob's case where the template *SleepingNoise* needs to be imported, see again Figure 4.21, `processToFillInBlocks` will encounter two input-blocks: `AliceSleeps`, which will be dispatched to `solveEventInput` as it is a custom event and `Location`, dispatched to `solveFact`.

Before getting into the implementation details of each of these three methods, it is good to have a look to the functionality they have in common: determining matching blocks for each input-block of the template being imported. In order to communicate suggestions to the client in the same way, `solveFactType`, `solveFact` and `solveEventInput` use the same type of object to be sent to the client: `IFactTypeSuggestions`. Shown in Figure 4.31, `IFactTypeSuggestions` contains five fields. Considering an input-block for which suggestions are provided, then: `index` is the index of that input-block in the template to be imported, `importIFactType` the input-

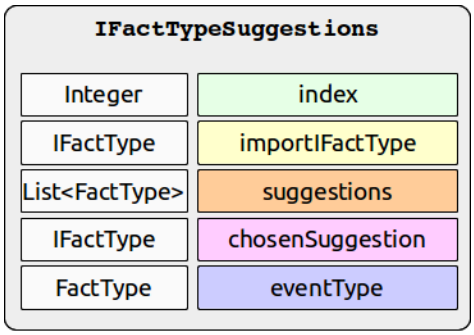

Figure 4.31: Overview of the `IFactTypeSuggestions` object

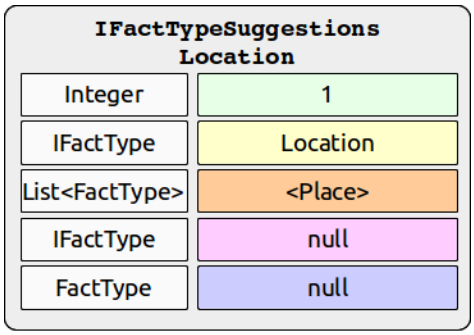

Figure 4.32: Suggestions for `Location`

block itself, `suggestions` the list of suggestions for this specific input-block, `chosenSuggestion` the block the user chose to replace the original input-block with[14] and lastly `eventType` which holds the `FactType` of the `importIFactType` in case the latter is a custom event (see later).

Assume Bob's CMT server is processing the `Location` block of the *Sleeping-Noise* template. Assume also that his system has a `FactType` called `Place` in its database. Then `IFactTypeSuggestions` might looks like shown in Figure 4.32, on the moment it is sent to Bob's client device. The index will be set to 1, as `Location` is the second input-block in the template. The `importIFactType` is the input-block itself: `Location`. Assuming Bob's CMT server only knows `Place`, a synonym for `Location`, it will be sent as a single suggestion in `suggestions`. Bob still needs to make a decision about this input-block, hence `chosenSuggestion` is initialised with a `null` value. So is `eventType` as no custom event is involved.

Suggestions for a FactType Creating suggestions for a `FactType` is taken care of by the `solveFactType` method, as shown in Figure 4.33. After having checked whether or not the current `FactType` is already resolved, the string matching, using Jaro-Winkler, component is called to find local `FactTypes` eligible to be a suggestion for it. Matching, as previously mentioned, is conducted using the name of the `FactType` in the template being processed and a local `FactType` as input. The higher the score, the better the match. After the matching process is done, the Jaro-Winkler component returns the results by means of a list containing pairs: each local `FactType` together with its score. Next, `solveFactType` loops over these

[14]Please mind that this can also be the input-block of the template being imported, in case the user does not want to replace it with a local one

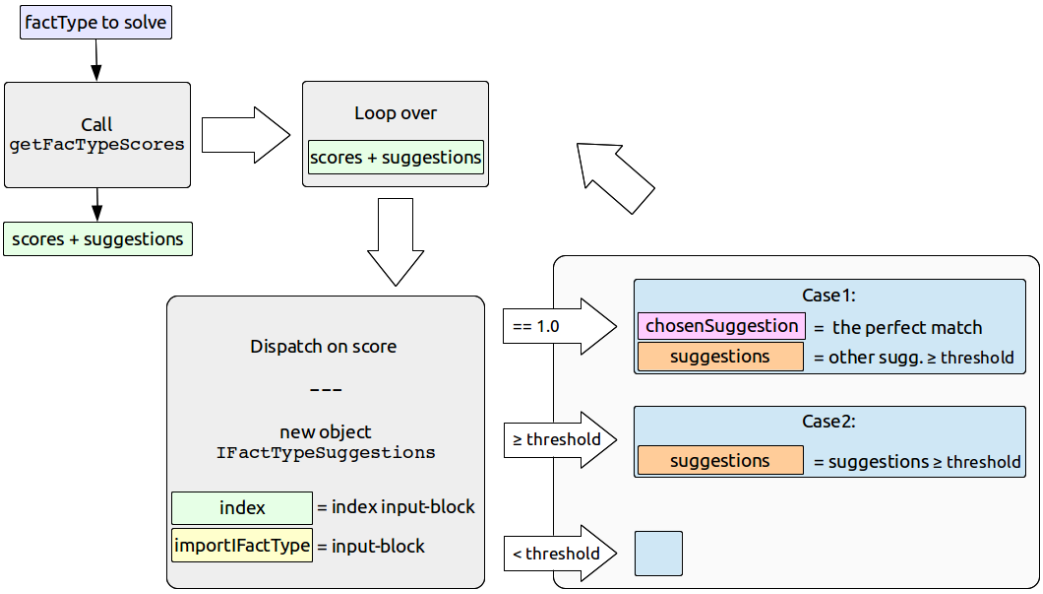

Figure 4.33: Overview of the `solveFactType` method

pairs of scores and corresponding `FactTypes` and dispatches based on score. If a `FactType` has a score of 1.0, a perfect match has been found, depicted by *case1* in Figure 4.33. In this specific case, `chosenSuggestion` of the `IFactTypeSuggestions` (see Figure 4.31) object is assigned with the perfect matching `FactType`. In this way, the client device knows a perfect match has been found and can change its behaviour to make the user aware of this. Other `FactTypes`, if their score is above a certain threshold, are added to the list with suggestions, shown as case2 in Figure 4.33). If the score of a particular `FactType` is below the threshold, it is ignored. The values assigned to the other fields of the `IFactTypeSuggestions` object are trivial: `index` being the index of the input-block involved, `importIFactType` the input-block itself. When ready, the `IFactTypeSuggestions` is added to the `TemplateSuggestions` object of the current template, keeping track of all suggestions for all input-blocks.

Suggestions for a Fact In case a `Fact` needs to be resolved, `solveFact` is called. This method, shown in Figure 4.34, is mostly analogue to the previous one discussed: `solveFactType`. To make it easier to understand for the reader, let us applied it on Bob's case, concerning the *Bedroom* `Fact` of `FactType Location` in template *SleepingNoise* (see Figure 4.21) he wants to import. First, `solveFact` checks if the current input-block, *Bedroom* has already been resolved. If not, its `FactType` is retrieved, which

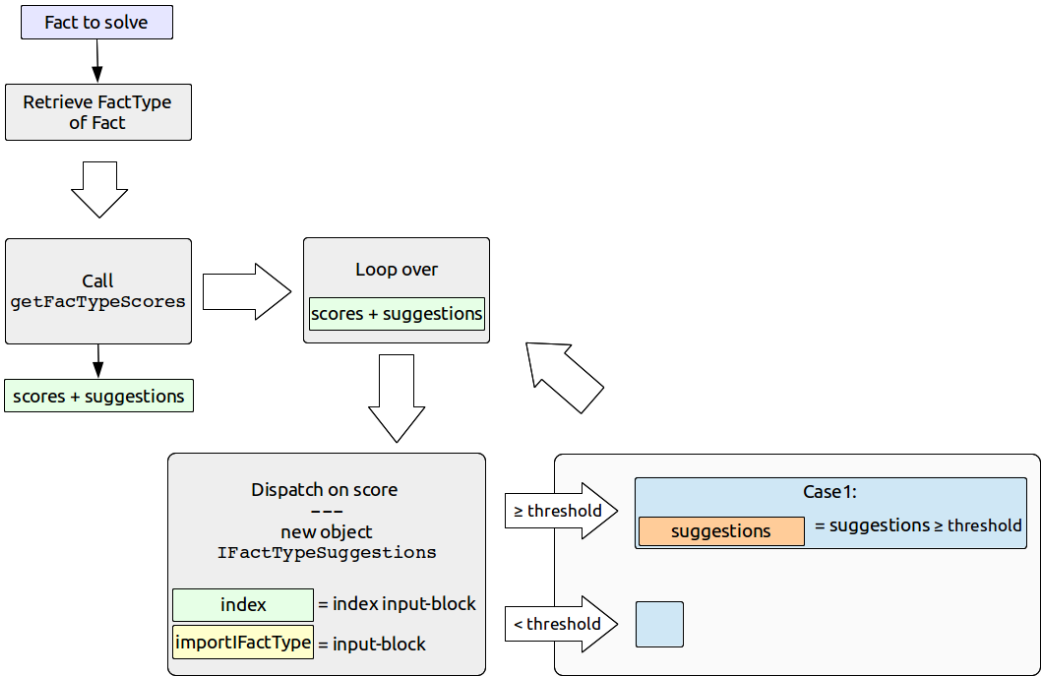

Figure 4.34: Overview of the `solveFact` method

is `Location`. Analogue to `FactType`, pairs with matching scores of local `FactTypes` are computed. Assume Bob's system only knows three `FactTypes`: `Loc`, `Light` and `Temp`. Matching these with `FactType Location` using the Jaro-Winkler algorithm returns following results: `Loc` - 0.85, `Light` - 0.60, and `Temp` - 0.00. Assuming a threshold of 0.5, Bob's server starts looping over these pairs. A major difference with `solveFactType` is the fact that `chosenSuggestion` of the `IFactTypeSuggestionsObject` for the current input-block, *Bedroom*, will not get a value assigned. The reason for this is simple: `chosenSuggestion` is expected to contain the block that will be inserted into the template after the user made his decision. In case a `Fact` is being solved, `chosenSuggestion` is therefore expected to contain a `Fact`. It would make no sense to assign a (perfect matching) `FactType` to it as, in this version of the extension, a `Fact` cannot be replaced by a `FactType`. Back to Bob's case, the list with `suggestions` will be filled with `Loc` and `Light` as only their scores are above the threshold of 0.5.

Suggestions for an event The third type of input-block a template can have, is an event. Events are represented by the type `EventInput` and are handled by the `solveEventInput` function, depicted in Figure 4.35. This method is, except for one major aspect, completely analogue to `solveFact`.

First it checks whether or not the current input-block is already solved, only then it retrieves its `FactType`. Yet, by design of CMT, this is not as straightforward as it is for `Facts`, where one can just generate the `FactType` of a `Fact` out of the `Fact` itself. Because an event does not contain enough information to extract its `FactType`, one needs to retrieve the missing information from another source. This source is the CMT instance where the event has been created. Of course, it is not feasible to contact a remote CMT server for each event encountered. Therefore, a slight modification to the JSON representation of a template has been made in such a way that it now also contains all the necessary information to extract the `FactType` of any of its events. Hence, `solveEventInput` can retrieve this and calls the Jaro-Winkler component to compute matching scores with local `FactTypes`, just like in `solveFact`. The next steps are also analogue: an `IFactTypeSuggestions` object is created of which the list of `suggestions` is then filled with all `FactTypes` having a score above the threshold and. Then both the `index` and `importIFactType` fields are assigned with the input-block index and input-block itself respectively. As mentioned before, one cannot retrieve the `FactType` of an event having only access to the `EventInput` object. To allow the client to compare `FactTypes` of events, an extra field called `eventType` is assigned with the `FactType` of the current input-block, the latter representing an event.

Until now, the flow of `solveEventInput` is, except for some minor details, similar to that of `solveFact` and `solveFactType`. Yet, as events can be custom, one additional step is required. When elaborating about the export functionality of the extension, it was shown that to import a custom events, the template responsible for it needs to be imported too. Otherwise, it is impossible to know when that custom event evaluates to true or false. As such, after having handled the suggestions part, the algorithm needs to check whether or not the current event is indeed custom. If so, a recursive call to `importTemplateRec` is made, to resolve the template responsible for it. Retrieving this template does not require a lot of effort, as it is, thanks to the export algorithm, provided in the JSON declaration of the current event. If the current event is not custom, `solveEventInput` stops.

Resolving according to the user's decisions

Having discussed the three algorithms to create suggestions for each type of input-block, i.e. `solveFactType`, `solveFact` and `solveEventInput`. All suggestions for all input-blocks are collected in a `TemplateSuggestions` object, grouping them by template. The relation between a `TemplateSuggestions` object and the `IFactTypeSuggestions` objects it holds is shown in Fig-

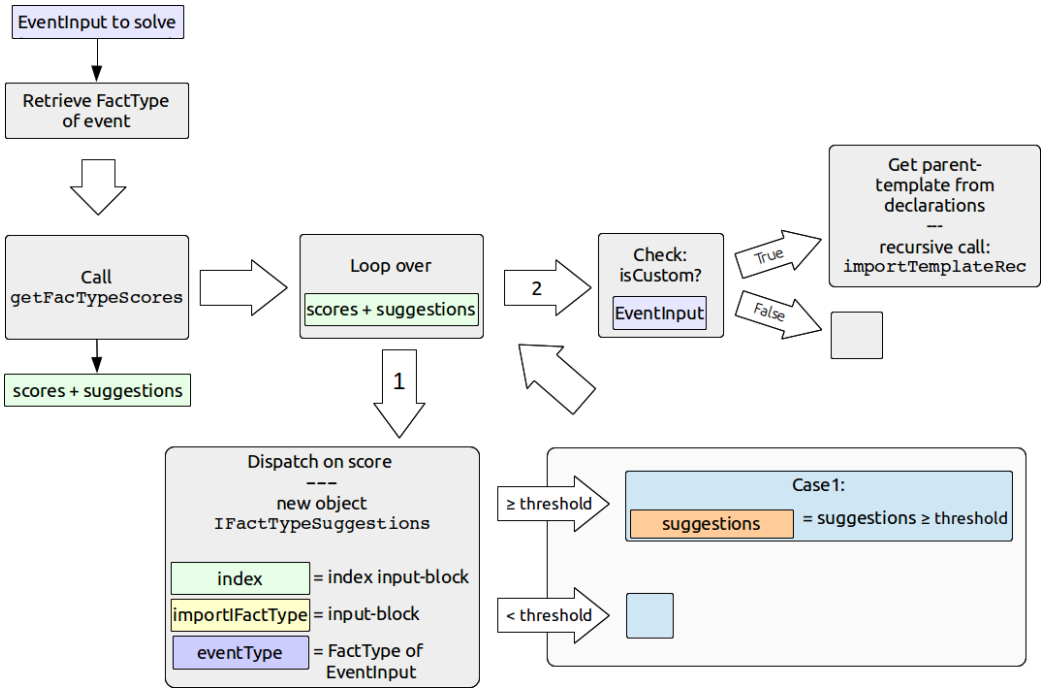

Figure 4.35: Overview of the `solveEventInput` method

ure 4.36. Once this is done, additions to the converter component of CMT are used to convert each `TemplateSuggestions` object to its corresponding JSON representation. The conversion process it trivial: every `String` and `Integer` type is converted to a `JSONString`[15]. Considering non-standard Java types such as e.g. `FactType`, the already built-in conversion methods are used. Once each of the `TemplateSuggestions` objects is converted, they are sent to the client by means of a `JSONArray`. Considering Bob's case, two `TemplateSuggestions` are created: one for the *SleepingNoise* template shown in Figure 4.21 and one for the `Sleeping` template shown in Figure 4.24. Both templates contain two input-blocks. As such, four different `IFactTypeSuggestions` objects will be created for which Bob will have to make a decision. Once Bob made a decision for each of these four input-blocks on one of his client devices (e.g. his smartphone), all the `IFactTypeSuggestions` objects with the `chosenSuggestion` field not equal to `null` are sent back to the server. Arrived there, an endpoint in the `CMTRest` component calls the `onSuggestionsReceived` method, as shown in Figure 4.37. First, the `JSONArray` is converted back to the original Java representation: a list of `TemplateSuggestions` objects. Next a nested loop starts to

[15]Using the JSON for Java library: `https://github.com/stleary/JSON-java`

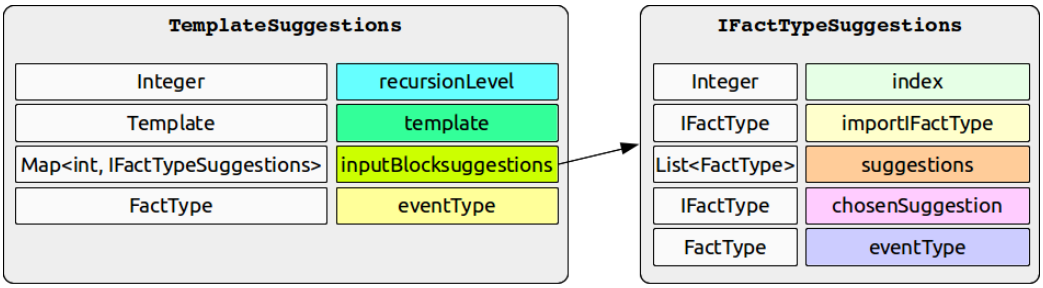

Figure 4.36: Relation between `TemplateSuggestions` and `IFactTypeSuggestions`

process every `IFactTypeSuggestions` object of every `TemplateSuggestions` object. While getting into each of these suggestions objects, the current template being processed is passed in its `HashMap` representation[16] to underlying methods. In Figure 4.37, this parameter is depicted by blue *templateInputs* boxes. Back to the loop, from each `IFactTypeSuggestions` object the `importIFactType` is retrieved. Remember that the latter represents the original input-block being resolved. Based on the type of it, `FactType`, `Fact` or `EventInput`, following methods are called respectively: `doSolveFactType`, `doSolveFact` and `doSolveEventInput`. Depending on the decision of the user, each of these methods will modify the input-block currently processed in the `HashMap` representation, i.e. *templateInputs*, of the corresponding template. When all input-blocks have been handled, *templateInputs* is converted back to the original template format and then added to CMT. This last operation makes use of an already existing method (i.e. `registerTemplate`) of the `CMTCore` interface, which is also used when a user creates a template on his own.

In the following subparagraphs, each of the three methods to which a user decision is dispatched will be discussed.

Solving a FactType An example of solving a `FactType`, respecting the decision of the user, could be the `Sleeping` template, shown on Figure 4.24, Bob's server wants to import. Both `Location` and `Person` input-blocks are `FactTypes`. In the following, only `Location` will be treated. Resolving `Person` is completely analogue.

Shown in Figure 4.38, one can see the algorithm to resolve `FactTypes` after

[16]Remind that this representation makes it less complex to modify input-blocks of a template, as mentioned before

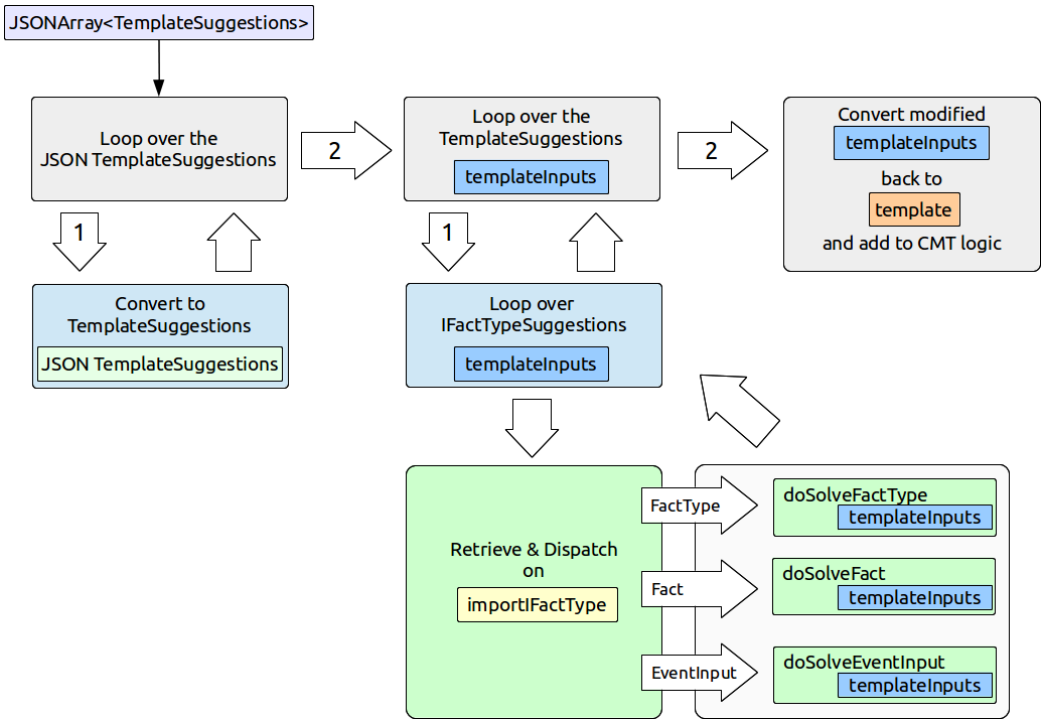

Figure 4.37: Overview of the `onSuggestionsReceived` method

the user made a decision: `doSolveFactType`. First, the value, a `FactType`, assigned to `chosenSuggestion` is checked to see whether or not it is in the list of suggestions that has been sent to the client. Remember that `chosenSuggestion` represents the decision a user has taken. Considering Bob's case, let us assume that he indeed chose one of the given suggestions, e.g. building block `Place`, to replace the original input-block `Location`. Assume that both `FactTypes` `Location` and `Place` have a field *room* of type `String`, and that `Location` also has an additional field *height* of type `Integer`. Bob chose a suggestion from the list of suggestions, implying that the first check in Figure 4.38 evaluates to true. In order for the template being imported, i.e. `Sleeping`, to work correctly, Bob's server will merge the original `FactType` of the input-block with the local one he has chosen. Hence, `Location` and `Place` need to be merged. This is not a trivial process, a `Place` is currently in use throughout Bob's system. A special method `mergeFactTypes` has been developed to abstract this complexity, as shown in Figure 4.39.

`mergeFactTypes` takes two parameters: the local and remote `FactTypes`, here `Place` and `Location` respectively. Looping over the fields, the algo-

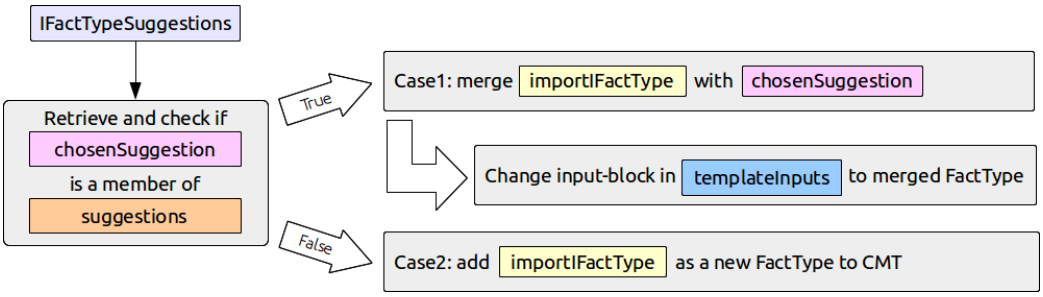

Figure 4.38: Overview of the `doSolveFactType` method

rithm checks whether or not the remote one has different fields than the local one. This check is done on both type and name of the field. Considering `Place` and `Location`, the latter has one extra field: *height*, which is added to a intermediary list of fields to be created. The next step is to call an additional method of `CMTCore`, designed to insert the new field. This is done by first inserting the new field into the list of fields of `Place` as a `FactType` object, independent from other components in CMT. The 'new' version of `Place` is then added to the database. Next, the old source and compiled code files of the `Place` type, representing it in the reasoning engine, are replaced by freshly generated source and compiled code.

Back to the `FactType` resolving algorithm shown in Figure 4.38, the only task left is to replace the original block in the template (using its `HashMap` representation *templateInputs*), with the new, merged one. In Bob's case, this means that `Location` will be replaced with `Place`, after the merge with `Location`. Considering case 2 in the algorithm, the user chose to keep the original `FactType`. Hence, the only task to be completed is registering that new `FactType` in CMT. No changes need to be made to the template.

Solving a Fact Solving a `Fact` according to the decision a user made has some similarities to how a `FactType` is resolved. As shown in Figure 4.40, the first check is to verify whether or not the value assigned to `chosenSuggestion` is already in the database. Other than in `doSolveFact`, it is not possible to check if the value of `chosenSuggestion` is in the list of suggestions, as the latter only contains `FactTypes`, while the former is required to be a `Fact`. Going back to the *SleepingNoise* template Bob wants to import, one can see that the second input-block is a `Fact`, *Bedroom*, with `Location` as `FactType`. Assume now that Bob's system already has `Facts` of `FactType Location` in its local database, e.g. *BobBedroom*, *Kitchen* and *Hallway*. Bob chose to replace the original *Bedroom* with the local `Fact`

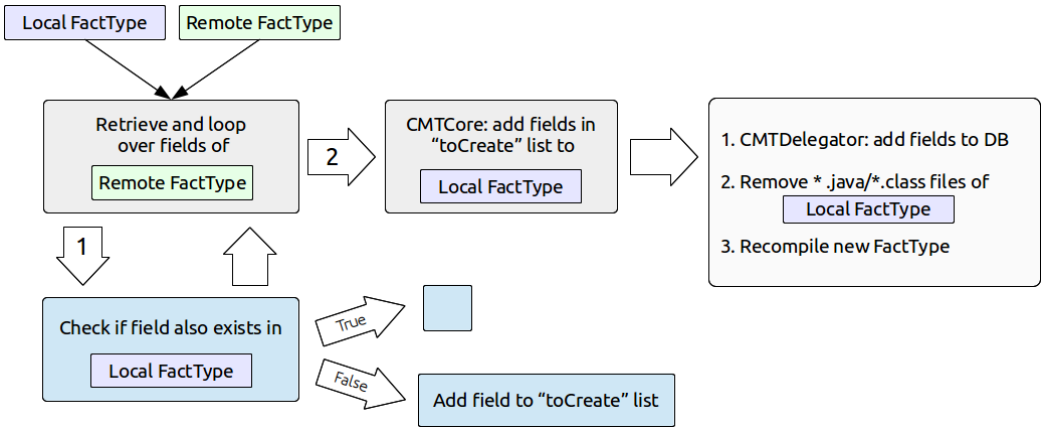

Figure 4.39: Overview of the `mergeFactTypes` method

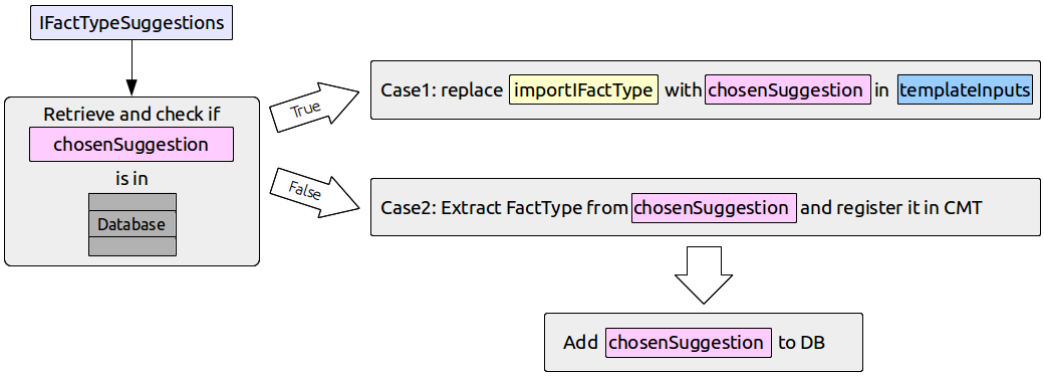

Figure 4.40: Overview of the `doSolveFact` method

BobBedroom. As such, control gets into case1. Except for replacing the *Bedroom* `Fact` with *BobBedroom* in the template, no further complex operations need to be done.

If Bob chooses to keep the original `Fact`, *Bedroom*, the algorithm gets into case2. Assuming that his system does not know the `FactType` of *Bedroom*, it needs to be extracted first and registered into CMT. After this, *Bedroom* is added to the database as `Fact` of its extracted `FactType`. Considering the template, no modifications are applied.

Solving an Event Last, there is the algorithm to solve events based on the user's decision: `doSolveEventInput`. Except for some differences in types, the flow of this method is identical to the one of `doSolveFact`, shown in Figure 4.40. If a user chose an event from the database, the original one

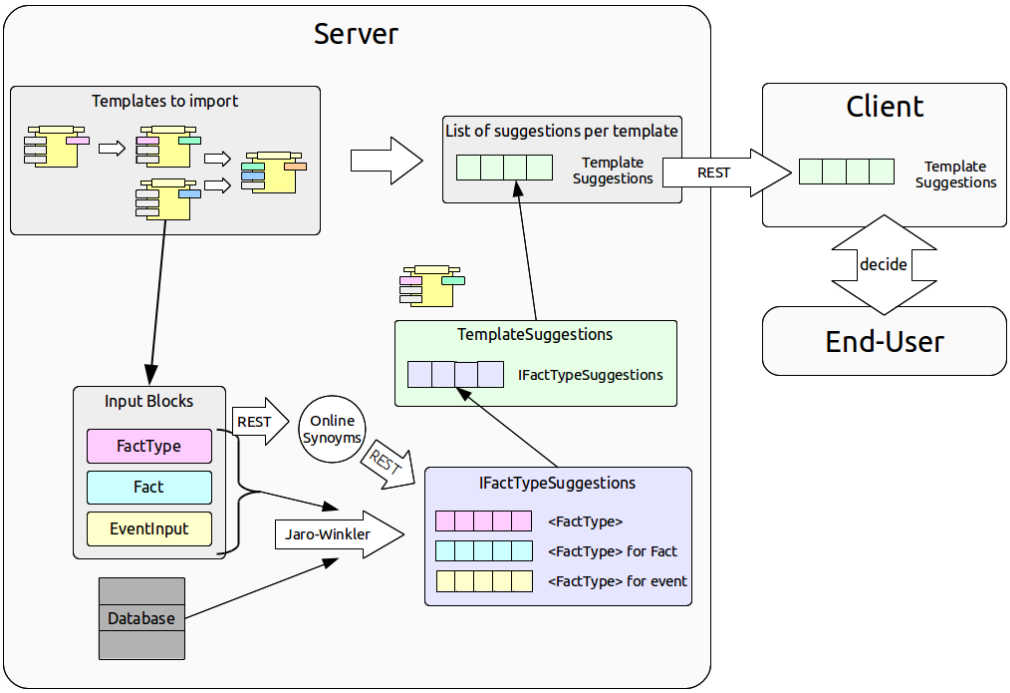

Figure 4.41: Importing a template, step one

in the database will be replaced with it. If not, the `FactType` of the original one is registered in CMT before adding the event itself to the database. Considering the template, no changes need to be made.

Summary of the importer

Having completed elaborating about the implementation of the importer, this subsection gives a summary to give the reader a better overview of what exactly happens during the import process. In case the reader would like to get deeper insights in how the importer (and exporter) are implemented, all code corresponding to all the flowcharts in this section can be found in the appendix. Yet, the reader should remind that a lot of details, extra tests, side-cases and helping methods have been left out for sake of simplicity. A summarising flowchart about the importer is given in Figure 4.41 and Figure 4.42, each respectively corresponding with the moment before sending suggestions to the client and after getting the client's decisions in return. Throughout the summary, the use case, with Alice and Bob, presented before is reused. Alice being the user sharing a rule with Bob.

Starting with step 1 in Figure 4.41, Bob's server receives the JSON representation of the template shared by Alice. Assume now that this template generates the orange event and has two custom events, green and blue, in its input blocks, as shown in Figure 4.41 in the left upper corner. In order to enable Bob's server to resolve this template, Alice's system also added the templates responsible for the green and blue custom events to the JSON file it sent to Bob. As the reader can see, the template creating the green event also has a custom event, i.e. pink, in its input-blocks. Hence, Alice's server also included the template generating the pink event. In order to resolve a template, all of its building-blocks need to be resolved. Yet, by checking the input-blocks alone, one can already know whether or not custom events are present. Three types of input-blocks exist: `FactType`s, `Fact`s and `EventInput`s. Each of them respectively coloured magenta, cyan and yellow, as shown in Figure 4.41. In order to maintain a certain level of intelligibility in the system, Bob is involved in the import process by allowing him to replace input-blocks of the template with local ones if preferred. Replacing IF-blocks or the output-block is meaningless, as this would change the semantics of the template. Hence, to make the process of finding the right local block for a particular input-block, Bob's server first checks what local building blocks might match. The latter is done by matching each local block with the input-block of the template based on `FactType` name.

The matching itself is done using a fuzzy string matching algorithm (i.e. Jaro-Winkler), as one cannot assume that two users name their `FactType`s in exactly the same way (e.g. abbreviations, different spelling and so on). Parallel to this name-based matching, an online service providing synonyms for a certain term is called with the name of the input-block in the template. Assume Bob's system is handling input-block `Person` of template `Sleeping` as shown in Figure 4.24. His system will then check if there are any local `FactType`s matching `Person` using Jaro-Winkler. For example, a local `FactType Pers` might get a good matching score. In parallel, the synonyms service is called over REST in order to retrieve synonyms for `Person`. If Bob has a local `FactType Human` and the synonyms service also returns this as a synonym, `Human` will be included in the list of suggestions. Independent of the type of the input-block, the list of suggestions will always contain `FactType`s, as these also cover the types of regular `Fact`s and `EventInput`s. This list of suggestions is contained into an `IFactTypeSuggestions` object, collecting all data needed for the client to resolve the current input-block. All `IFactTypeSuggestions` objects are then put in a list contained by a `TemplateSuggestions` object. The latter contain-

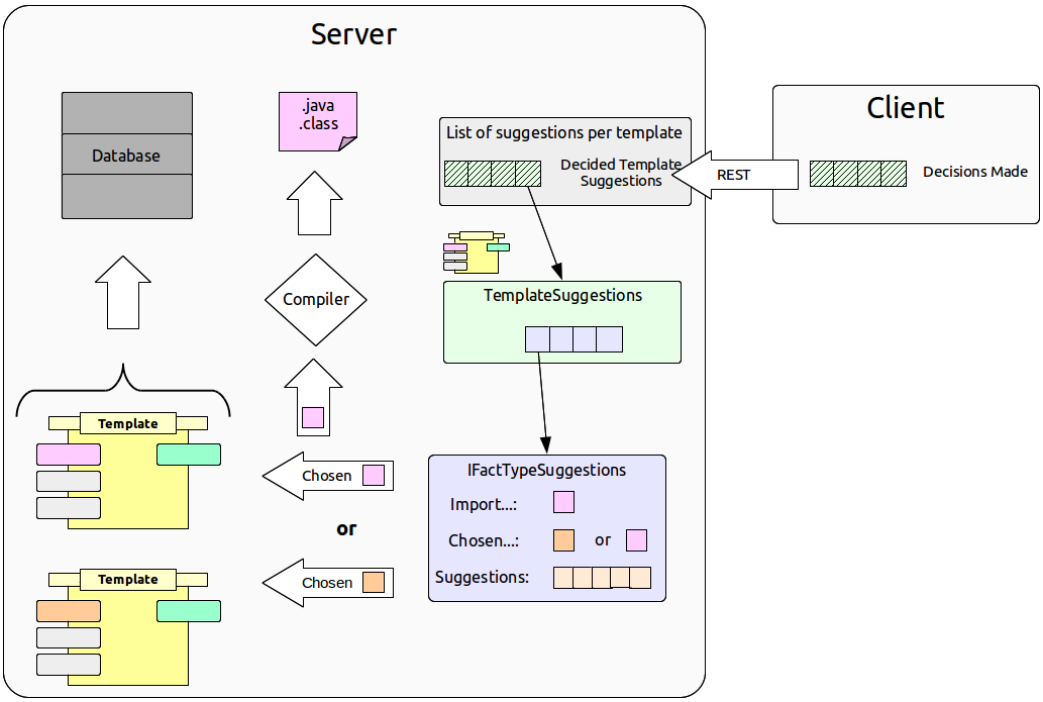

Figure 4.42: Importing a template, step two

ing all information the client needs to solve a particular template. More information about the structure and relation between `IFactTypeSuggestions` and `TemplateSuggestions` is shown in Figure 4.36. In the next step, all of these `TemplateSuggestions` objects are collected and converted to JSON before sending them to the client. In the case shown by Figure 4.41, four templates with each three input-blocks need to be solved. Once arrived at the client, the user needs to decide whether or not to keep the original input-block(s) or replacing them by local ones, i.e. the ones in their database.

Once the client, Bob, made a choice for each of the input-blocks, the list of `TemplateSuggestions` objects is sent back to the server. Arrived there, each `IFactTypeSuggestions` object is processed, as shown in Figure 4.42. Depending on the type of the input-block, `FactType`, `Fact` or `EventInput`, a different solving algorithm is used. Yet, the general flow of each of them is similar. Assume now that Bob chose to keep an input-block that came with the template from Alice. Hence, the algorithm gets into the pink case on Figure 4.42: `chosenSuggestion` equals the `importIFactType`. As such, concerning only the current input-block, no changes need to be made to the template. Yet, the `FactType` of the input-block needs to be introduced into

CMT, both on application and database level. Regarding the application level, the reasoning engine needs to be informed about the existence of a new type. Hence, source code is generated for the latter, which is then compiled to a *.class file, which is used by the reasoning engine. In case Bob chose one of the suggestions, the template is modified by replacing the original input-block (pink in Figure 4.42) with the suggestion Bob chose (orange). After all modifications to the template have been done, it is saved in the database for later use.

Using this procedure, Bob can import any rule or template from any possible user he likes. The only expertise required, is knowing what the rule being imported looks like, what its IF-blocks represent and how those blocks can be mapped on his own environment, i.e. the sensors he already installed. Yet, if we have a look back to the event definition Bob imported from Alice, i.e. `DisturbingNoise`, Bob's problem is not solved yet. The event `DisturbingNoise` is fired from the moment the ambient noise sensor detects any disturbing noise and Bob is sleeping. But at the moment, nothing will happen yet. Bob still needs to connect his 'anti-noise' rules, e.g. close all windows, to this event. This can easily be done by just using `DisturbingNoise` as an IF-block and the appropriate action to be undertaken in the same rule. If `DisturbingNoise` is fired now, all the actions Bob configured will be executed. This should hopefully give him some good nights sleep back.

5 Conclusion and Future Work

In this chapter, we look back to what has been presented in this thesis. A discussion about the rule share extension developed for CMT and the influence on its intelligibility, a key factor in this work, will be given. We will also discuss what can still be improved or made more efficient. After that, some future work will be presented, such as we can build further on this work and what the future does look like in this particular research field. Finally, some general conclusions concerning this thesis are given.

5.1 Discussion

Thanks to the revolution in hardware for mobility, we can now take mobile devices, e.g. mobile phones or tablets, with us everywhere we like. A second feature resulting from this mobile revolution is the fact that these mobile devices contain a lot of sensors. Both hardware and software sensors. A hardware sensor can be seen as a sensor detecting a physical property of the environment, e.g. light or sound, whereas a software sensor deduces facts en preferences by analysing user data, e.g. determining when a phone should be silent by looking for meetings in the user's personal agenda. More and more sensors are introduced, creating a strong growth in context-aware sensing possibilities. Nowadays, there even exist solutions to detect a person's mood and feelings [50]. Combining the data generated by all of these sensors en-

ables us to capture an accurate picture of the surrounding world [51]. With the upcoming Internet Of Things [21, 52], even more devices will get interconnected, creating an even smarter and denser grid of sensors. This will allow us to detect even more context keys more efficiently and accurately [53]. The role of context awareness in our lives will therefore only increase further.

However, this growing automation thanks to the increasing interest in context awareness also has its downside as intelligibility might get lost. Depending on the complexity of a system, an average user no longer has any idea about what exactly is happening behind the scenes [18]. This is where CMT appears on the scene. As a Context Modelling Toolkit, CMT ensures that every user can make use of it without losing insights on how the system reasons about context awareness [12]. In this thesis, an extension for CMT has been created, enabling users to share their rules and custom made events with other users. By sharing rules, users do not need to create common rules, e.g. *switch off all lights when I go out*, again and again. Second, expert users can help inexperienced end users by providing them with some more complex rules.

Having a look at other context modelling systems, one can observe that not all of them have the functionality to share rules. Systems such as IFTTT[1], Zapier[2] [54] and WigWag[3] allow users to share rules. Yet, they have two major drawbacks. First of all, the rules one can create with these tools are rather simple, as they all work using the same 'mask', i.e. `IF` something `THEN` do something else. Second, the rules one can create with them are fixed to the resources, i.e. sensors or services, they need. For example, a rule changing the colour, e.g. a Philips Hue lamp[4], will only work with that specific lamp of that specific brand. On the other hand, there do exist systems with a lot more expressiveness, but they lack the ability to share rules or are reluctant to runtime changes in the structure of the environment they are operating in. A good example of the latter is ontology-based systems [55], such as SOCAM [23]. Changing the ontology at runtime might introduce issues concerning ontology compatibility and integrity. More expressiveness might also imply more complexity for the end user and hence a loss of intelligibility.

As mentioned before, one of the key aspects of CMT is allowing users to understand how reasoning about their context rules is done. However, im-

[1] `ifttt.com`
[2] `zapier.com`
[3] `wigwag.com`
[4] `http://www2.meethue.com/`

porting rules from other users might have a negative effect on our intelligibility level. To cope with this, our extension involves the user directly into the importing process. As every rule consists of building blocks, which might be generated by other rules, these blocks need to be mapped onto building blocks already existing in our system, or on should create new ones. Technically, we could have made a fully automated process out of this mapping, but that would mean that our end users would not know of what building blocks their (imported) rules actually consist. By being involved in the decision process, users are motivated to invest some cognitive effort in the working of the importing mechanism, hence creating awareness about what is mapped. The mapping itself is based on the class name of a building block. In order to already help the user a bit in finding the right local building block to map on, we use the Jaro-Winkler fuzzy string matching algorithm on that class name. Users then only have to choose what (type of) block they want in their final rule, i.e. a local known one or the foreign one provided by the rule being imported. Out of five fuzzy string matching algorithms, Jaro-Winkler came out as one of the better to be used for our specific case. The main reason for this choice is the fact that the final score resulting from a comparison using Jaro-Winkler is in favour of prefixes. This is beneficial, as humans usually use prefixes to abbreviate terms. Matching two terms where on is a prefix of the other, e.g. *Loc* and *Location*, will therefore result in a high matching score. In case the user does choose one of the suggested high matching score blocks, the original block is merged with or replaced by the chosen one, depending on whether or not it is a type or specific fact/event. In case the user picks the original block provided by the rule being imported, a new type matching it is created in the system.

From a user's perspective, importing a rule involves some cognitive effort in order to guarantee intelligibility. Depending on the types of blocks the rule consists of, we can distinguish three cases. In case of a `FactType` the user can choose one of the given suggestions or the `FactType` provided by the rule itself. In case of a `Fact`, the user has again the choice. However, in this case, the server first sends `FactType` suggestions matching the `FactType` of `Fact` being imported. From here on, the client takes over: when the user chooses one of the suggested `FactType`s, the client sends a request to the server to retrieve all the `Fact`s being of the chosen `FactType`. The user can then pick one of these `Fact`s to insert into the rule. Otherwise, if the original `Fact` of the rule is chosen, the system will create a new `FactType` for it and add that `Fact` to its database, as already mentioned. Solving an event is almost identical from a user's perspective.

In this thesis, we mostly focussed on the back end part of solving the context rule sharing problem for CMT. Using the REST API, any developer can develop an appropriate user interface for any device and any user they like. However, to give the reader an overview on what such an interface could look like, some sketches of the importing and exporting process on a tablet device have been made. One import aspect to notice is that each of these sketches involving a specific rule always shows a visual representation of that rule. As such, we try to prevent that the user would get lost. Additionally in the block matching interface (during the import phase), each block that has been resolved already is marked. Hence, a user knows exactly how much progress they already made in mapping a particular rule to their own environment. All of these aspects have on common goal: maintaining, or even improving, the level of intelligibility a user has about the system.

5.2 Future Work

Having a look at what can still be improved and added to the extended version of CMT, the first aspect to mention is conflict management. The ultimate goal of this thesis is to allow users to share rules with each other in order to reduce the complexity of creating them. Except for matching the building blocks of these rules, there is another major issue: conflicts may arise with rules that are already in the system. A straightforward example could be a rule opening the windows when it is too warm inside and another rule closing them when it gets dark. As such, on a summer evening, a conflict might appear. Should the windows be opened or closed? One way to solve this could be introducing a priority mechanism: e.g. imported rules have a lower priority than default rules. But what about two conflicting imported rules then? This problem can be generalized to the whole of CMT. What if a user creates a rule that conflicts with a rule already in the system. How can the system check for conflicts? How could it tell this to the user without losing intelligibility? Again, if a user does not now how the system reasons internally (e.g. about conflicts), the user might not be able to solve these conflicts. As other research points out, it is important to provide an answer to a user's *why* and *why not* questions in order to get more insight into the state of the system at that moment [56]. Solving the conflicting rules problem is a research topic closely related to the design of CMT, but falling out of the main scope of this thesis.

Another (major) issue is the fact that we still need an intermediary pro-

tocol to communicate between two parties. Assume, for example, two CMT instances wit the same underlying implementation. Instance A is connected to a temperature sensor measuring in Fahrenheit while the sensor of instance B measures in Celsius. Given the current implementation, one might run into problems when sharing a rule from A to B (and vice versa). The most straightforward solution to solve this issue is to create general protocols for specific types of sensors. Unfortunately, this means that CMT might adopt some of the unfavourable properties of the extensively mentioned, in this thesis, IFTTT system where rules are directly related to specific hardware, and are therefore less reusable.

Having a closer look at the implementation of the CMT extension, we can identify some issues that can still be improved. First of all, the communication with the client contains some unnecessary overhead: at the moment, suggestions for a specific building block are encoded in a dedicated object which is then send to the client (`IFactTypeSuggestions`). When the user makes a decision about how that building block should be solved, the same object including all those suggestions is sent back to the server, although the server does not need those suggestions but only the decision made by the user. Second, some improvements can also be made in the way `FactType`s of facts and events are merged. If a user now decides to pick the fact or event provided by the rule to be imported, a new `FactType` is automatically created. This could be enhanced by also allowing the merging of the `FactType` of the fact/event considered with a similar `FactType` already in the database. However, a good balance must to be found. Introducing this feature means that we require extra cognitive effort from the user as the system needs to know with what internal `FactType` the `FactType` of the imported fact or event should be merged. It might become too complex for less experienced users to distinguish between the type of a fact or event and that fact or event itself. A third aspect of the implementation that can be enhanced further on is how synonyms are retrieved and processed. Currently, synonyms are retrieved using an external public service with the name of the building block we need to solve. Assume a rule we want to import contains a building block named "Location". In our database, we already have the semantically similar block named "Place". In case the external synonym service returns "Place" as a synonym for "Location", the "Place" block will appear in the suggested blocks shown to the user. Yet, this approach does not work to match, for example, "PersonInPlace" with "PersonInLocation", as these strings contain multiple appended words. One solution could be splitting these sequences into separate words and then trying to find synonyms matching one or more

of these words. However, this might have a significant influence on the performance of the system. Especially finding linguistic subwords in a sequence of appended words is not trivial, as one will need to poll a dictionary for every prefix, infix or suffix (depending on the algorithm being used) of the word sequence to split and check if it is an actual word. Each time we then find such a subword, we need to check if synonyms exist for it. In short, this is a rather expensive process as external services, which come with request delays, need to be called. Another option could be keeping the dictionary in CMT itself and hence preventing calls to an external one.

5.3 Conclusion

In this thesis, an extension for the Context Modelling Toolkit (CMT) developed at the Web and Information Systems lab of the Vrije Universiteit Brussel has been created. The extension allows users with different levels of expertise to share context rules with each other using CMT. Mainly due to the fact that an increasing amount of different sensors are developed together with the revolution in mobile devices, context awareness has become a hot research topic. As such, frameworks to model context allow the user to define actions and preferences on their devices (e.g. show a list of nearby restaurants at noon) depending on the context around them. Yet, most of these frameworks have only limited features to share rules. We identified two different groups of frameworks. One group having a sharing mechanism that is very easy to use, the typical example given in this thesis being IFTTT. However, due to their simplicity, context rules made with these systems are less reusable in environments other than those they have been developed for. The typical example is a rule made to control a smart lamp of one brand not being compatible with one of another brand. The other group of context modelling tools has more advanced features to model context and guarantee compatibility. Nevertheless, they are more complex to use for a general user, introducing a particular risk to lose intelligibility. During the lecture study, the term "intelligibility" has been analysed in great detail and played a major role throughout this thesis. Every extension developed for CMT should keep this in mind as CMT is fully built to take its users along in how it reasons about rules.

Considering the extension, the way a user can share a rule should be as easy as pie. A user Alice can easily share a rule with another user Bob. Once Alice picks a rule to share on her client device (e.g. her phone or tablet), Alice's CMT server transforms that rule into a shareable format. The most

significant factor in this export process is the fact that the server needs to recursively go through all custom events (i.e. an event reusing other events in its definition) to provide the importing CMT server (Bob) with all necessary data to import the rule. Three different strategies Bob's server can undertake to effectively import the shared rule have been discussed. First, there is the so-called naive method where Bob's server would treat the shared rule the same as a rule made by himself. This is not feasible, as Bob's CMT server might not have all the sensors the shared rule requires to work properly, meaning that a lot of shared rules would just not work. A second strategy discussed is the introduction of types and subtypes for the building blocks of a rule. However, this can be seen as introducing an ontology for building blocks which is very difficult to maintain in case new types are introduced or existing types are changed. Therefore, we discuss a third strategy based on fuzzy string matching applied on the type names of the building blocks of a rule. By offering the user (Bob) similar types based on type name and allowing him to reuse building blocks that are already in his system, rules have a higher chance to be fully compatible on the system they are imported by, but it is also beneficial towards the intelligibility of the importing user (Bob). As Bob decides how a rule is being imported, he can create an image about how his system will behave after importing that rule. Last, there is the Jaro-Winkler fuzzy string matching algorithm being used to match the type names of rule building blocks. In total, five fuzzy string matching algorithms have been compared and evaluated on 126 home automation related terms. Jaro-Winkler appeared to be the best choice due to its tolerance concerning abbreviations and simplicity. The matching process has also been augmented by adding a check on potential synonyms for type names in order to ensure a best possible fit.

To conclude, we can state that the extension brings fresh ideas to context modelling research, as it allows one to share complex context rules and templates. Hence, users with almost no expertise in the domain can get access to more complex context rules which they most likely would not be able to create themselves. By involving the user in the process of mapping building blocks of remote context rules to their own environment, they get more insight in how the system reasons and handles context rules. Therefore, intelligibility for all users is maintained. The idea of type name based rule matching approach using the Jaro-Winkler algorithm can be reused to enable and/or extend sharing functionality in existing systems, independent of the underlying context modelling approach they use. As such, creating/sharing complex context rules with the masses should no longer be pie in the sky.

A

Appendix: Implementation Code

This appendix consists of code snippets considering the implementation of the rule sharing extension for CMT, in particular those that are explained in section 4.5.

A.1 Exporting a template

This section incorporates all code of the template exporter explained in section 4.5

A.1.1 prepareTemplateSkeletonJSON

```
public JSONObject prepareTemplateSkeletonJSON(Template templ) {
        JSONObject res = Converter.fromTemplateToJSON(templ);
        res.remove("ifblocks");

        JSONArray nestedIFBlocks = new JSONArray();
        processIFBlocks(templ, nestedIFBlocks);
        res.put("ifblocks", nestedIFBlocks);

        ArrayList<FactType> eventTypes = new ArrayList<>();
        for(IFBlock ifblock : templ.getIfBlocks()){
            if(ifblock.getEvent() != null){
                FactType eventType = ifblock.getEvent();
                eventTypes.add(eventType);
            }
        }
```

```
        JSONArray tmplEventTypes = Converter.fromEventTypeListToJSON(
            eventTypes);
        res.put("tmplEventTypes", tmplEventTypes);

        return res;
    }
```

A.1.2 processIFBlocks

```
public void processIFBlocks(Template t, JSONArray resultBlocks){
        LinkedList<IFBlock> ifBlocks = t.getIfBlocks();

        for (int i = 0; i < ifBlocks.size(); i++) {
            IFBlock currBlock = ifBlocks.get(i);
            JSONObject jIFBlock = new JSONObject();
            JSONObject jFuncEvent;
            JSONArray jRecArray;
            jIFBlock.put("index", i);

            if (currBlock.getEvent() != null) {
                jFuncEvent = Converter.fromFactTypeToJSON(currBlock.getEvent
                    ());
                jIFBlock.put("event", jFuncEvent);
                jIFBlock.put("typeBlock", "activity");

            } else if (currBlock.getFunction() != null) {
                Function func = currBlock.getFunction();
                jFuncEvent = Converter.fromFunctionToJSON(func);
                jIFBlock.put("function", jFuncEvent);
                jIFBlock.put("typeBlock", "function");
            }
            LinkedList<Binding> bindings = currBlock.getBindings();
            jRecArray = processBindings(bindings);
            jIFBlock.put("bindings", jRecArray);
            resultBlocks.put(jIFBlock);
        }
    }
```

A.1.3 processBindings

```
public JSONArray processBindings(LinkedList<Binding> bindings) throws
    ClassNotFoundException, Exception {
        JSONArray jResBindings = new JSONArray();
        for (int i = 0; i < bindings.size(); i++) {
            Binding currBinding = bindings.get(i);
            JSONObject objBind = Converter.fromBindingToJSON(currBinding,i);
            JSONArray declarations = new JSONArray();
            BindingParameter endBinding = currBinding.getEndBinding();
            BindingInputBlock bindingInputBlock = (BindingInputBlock)
                endBinding;
            IFactType inputObject = bindingInputBlock.getInputObject();

            // InputObject is ALWAYS an event (a function can never be in
                the endbindings)
            if (inputObject instanceof FactType) {
                FactType inputObjectEvent = (FactType) inputObject;
                boolean isCustom = inputObjectEvent.isIsCustom();
                if (isCustom) {
                    declarations = processCustomEvent(inputObjectEvent);
                } else {
```

```
                    // inputObject is not custom -> no declarations needed
                    // => declarations: []
                }
            } else if (inputObject instanceof Fact){}
            objBind.put("declarations", declarations);
            jResBindings.put(objBind);
        }
        return jResBindings;
    }
```

A.1.4 processCustomEvent

```
public JSONArray processCustomEvent(IFactType event) throws Exception {
        JSONArray jResultArray = new JSONArray();

        // Retrieve the class name
        String eventClassName;
        if (event instanceof FactType) {
            eventClassName = ((FactType) event).getClassName();
        }else {
            throw new Exception(SharingImportExport.class.getName()
                    + "processCustomEvent -- event not of type 'FactType'");
        }
        // Retrieving the corrseponding template
        // SQL: Search template responsible for "event"
        CMTDelegator delegator = CMTDelegator.get();
        Template parentTemplate = delegator.getTemplateOfSituation(
            eventClassName);

        // SQL: check for custom event IFBlocks, recursively
        ArrayList<FactType> customEvents = delegator.
            getCustomEventsUsedInTemplate(parentTemplate.getSql_id());

        // There exist custom events
        if (!customEvents.isEmpty()) {
            jResultArray.put(prepareTemplateSkeletonJSON(parentTemplate));
        } else {
            // parentTemplate has no custom Events?
            // => Still have to convert the parentTemplate itself
            jResultArray.put(Converter.fromTemplateToJSON(parentTemplate));
        }
        return jResultArray;
    }
```

A.2 Importing a template

A.2.1 importTemplateRec

```
public void importTemplateRec(JSONObject jTemplate, Integer recursionLevel){
        ArrayList<Integer> resolvedIndexes = new ArrayList<>();
        TemplateSuggestions currTmplSuggs;
        TemplateHA tmpl = Converter.fromJSONtoTemplateHA(jTemplate);
        FactType eventTypeOfTmpl = getEventTypeTemplateProducing(tmpl);

        // Create TemplateSuggestions object
        if(eventTypeOfTmpl != null){
           currTmplSuggs = new TemplateSuggestions(recursionLevel, tmpl,
               eventTypeOfTmpl);
```

```
        }

        // Bookkeeping
        HashMap<Integer, IFactType> indexToToFillInBlocks =
            getInputsTemplate(tmpl);
        HashMap<IFactType, Integer> toFillInBlocksToIndex = new HashMap<>();
        for(Map.Entry<Integer, IFactType> entry: indexToToFillInBlocks.
            entrySet()){
            toFillInBlocksToIndex.put(entry.getValue(), entry.getKey());
        }
        processToFillInBlocks(indexToToFillInBlocks, toFillInBlocksToIndex,
            recursionLevel,
                resolvedIndexes, jTemplate, tmpl, currTmplSuggs);

        // Add templateSuggestions
        suggestionsPool.add(currTmplSuggs);

    }
```

A.2.2 processToFillInBlocks

```
private void processToFillInBlocks(HashMap<Integer, IFactType>
    indexToToFillInBlocks,
            HashMap<IFactType, Integer> toFillInBlocksToIndex, Integer
                recursionLevel,
            ArrayList<Integer> resolvedIndexes , JSONObject jTemplate,
            TemplateHA tmpl, TemplateSuggestions currTmplSuggs) {
        HashMap<IFactType, Integer> toFillInBlocksToIndexIter =
        (HashMap<IFactType, Integer>) toFillInBlocksToIndex.clone();

        for (Map.Entry<IFactType, Integer> entry : toFillInBlocksToIndexIter
            .entrySet()) {
            IFactType toFillInBlock = entry.getKey();

            if (toFillInBlock instanceof FactType) {
                FactType fType = (FactType) toFillInBlock;
                solveFactType(fType, indexToToFillInBlocks,
                        toFillInBlocksToIndex, resolvedIndexes, tmpl,
                            currTmplSuggs);
            } else if (toFillInBlock instanceof Fact) {
                solveFact((Fact) toFillInBlock, indexToToFillInBlocks,
                        toFillInBlocksToIndex, resolvedIndexes,
                            currTmplSuggs);

            } else if (toFillInBlock instanceof EventInput) {
                 solveEventInput((EventInput) toFillInBlock,
                         indexToToFillInBlocks, toFillInBlocksToIndex,
                             recursionLevel,
                         resolvedIndexes, jTemplate, tmpl, currTmplSuggs);
            } else {
                System.out.println("ERROR -- ShIX - processToFillInBlocks --
                    could not determine type of ToFillInBlock");
            }
        }
    }
```

A.2.3 solveFactType

```
public void solveFactType(FactType factType,...){
        // Type is already resolved
```

```
        if(resolvedIndexes.contains(toFillInBlocksToIndex.get(factType))){
            return;
        }
        resolvedIndexes.add(toFillInBlocksToIndex.get(factType));
        Integer index = toFillInBlocksToIndex.get(factType);

        ArrayList<Pair<Double, FactType>> scores = getFactTypeScores(
            factType);
        if (!scores.isEmpty()) {
            // Loop over the scores to check for perfect matches
            for (Pair<Double, FactType> score : scores) {
                if (score.getKey().equals(1.0)) {
                    // CASE1: perfectMatch: Already fill in chosensugg
                    IFactTypeSuggestions suggs = new IFactTypeSuggestions(
                        index, factType, scores, score.getValue());
                    currTmplSuggs.addIFactTypeSuggestions(index, suggs);
                    return;
                }
            }
            // CASE 2: no perfect match, but suggestions
            IFactTypeSuggestions suggs = new IFactTypeSuggestions(index,
                    factType, scores);
            currTmplSuggs.addIFactTypeSuggestions(index, suggs);
        } else {
            // CASE 3: No matches, Scores is empty -> no match
            IFactTypeSuggestions suggs = new IFactTypeSuggestions(index,
                    factType);
            currTmplSuggs.addIFactTypeSuggestions(index, suggs);
        }
    }
```

A.2.4 solveEventInput

```
    public void solveEventInput(EventInput eventInput,...) {
        if (resolvedIndexes.contains(toFillInBlocksToIndex.get(eventInput)))
             {
            return; // eventInput already resolved
        }
        resolvedIndexes.add(toFillInBlocksToIndex.get(eventInput));
        FactType eventType = null;
        if (jTemplate.has("tmplEventTypes")) {
            JSONArray jEventTypes = jTemplate.getJSONArray("tmplEventTypes")
                ;
            ArrayList<FactType> eventTypes = Converter.
                fromJSONToEventTypeList(jEventTypes);
            for (FactType currEventType : eventTypes) {
                if (currEventType.getClassName().equals(eventInput.
                    getClassName())) {
                    eventType = currEventType;
                    break;
                }
            }
        }
        Integer index = toFillInBlocksToIndex.get(eventInput);
        ArrayList<FactType> suggestions = getFactTypeSuggestions(eventType);
        IFactTypeSuggestions suggs = new IFactTypeSuggestions(index,
            eventInput, suggestions);
        suggs.eventType = eventType;
        currTmplSuggs.addIFactTypeSuggestions(index, suggs);
     if(eventType.isIsCustom()){
           JSONObject jDeclaringTemplate =
```

```
                getDeclaringJTemplateOfCustomEvent(jTemplate, eventType);
            importTemplateRec(jDeclaringTemplate, recursionLevel + 1);
        }
    }
```

A.2.5 onSuggestionsReceived

```
public void onSuggestionsReceived(JSONArray jSuggsList){
        ArrayList<TemplateSuggestions> tmplSuggsList = Converter.
            fromJSONToTemplateSuggestionsList(jSuggsList);
        for(TemplateSuggestions tmplSuggs: tmplSuggsList){
            TemplateHA tmpl = tmplSuggs.getTemplate();
            HashMap<Integer, IFactTypeSuggestions> indexToSuggs = tmplSuggs.
                getIndexToSuggestions();
            HashMap<Integer, IFactType> indexToToFillInBlocks = iX.
                getInputsTemplate(tmpl);

            for(Map.Entry<Integer, IFactTypeSuggestions> entry: indexToSuggs
                .entrySet()){
                IFactTypeSuggestions iFTSuggs = entry.getValue();
                IFactType fTypeToResolve = iFTSuggs.getImportIFactType();
                if(fTypeToResolve instanceof FactType){
                    iX.doSolveFactType(iFTSuggs, indexToToFillInBlocks);
                } else if(fTypeToResolve instanceof Fact){
                iX.doSolveFact(iFTSuggs, indexToToFillInBlocks);

                } else if(fTypeToResolve instanceof EventInput){
                    iX.doSolveEventInput(iFTSuggs, tmplSuggs,
                        indexToToFillInBlocks);
                }
            }
                iX.setInputsTemplate(tmpl, indexToToFillInBlocks);
                CMTCore.get().addTemplateHA(tmpl);
        }
    }
```

A.2.6 doSolveFactType

```
public void doSolveFactType(IFactTypeSuggestions iSuggs,...) {
        FactType importFT = (FactType) iSuggs.getImportIFactType();
        FactType chosenFT = (FactType) iSuggs.getChosenSuggestion();
        ArrayList<FactType> suggslst = iSuggs.getSuggestions();

        for (IFactType iFT : suggslst) {
            FactType currFT = (FactType) iFT;
            if (currFT.equals(chosenFT)) {
                // CASE 1: user chose a suggestion (which is already in the
                    db) => mergeFactTypeFacts
                    FactType updatedFactType = mergeFactTypeFact(importFT,
                        dbFTCheck);
                    Integer index = iSuggs.getIndex();
                    indexToToFillInBlocks.replace(index, updatedFactType);
                    return;
            }
        }
        if (chosenFT.equals(importFT)) {     // CASE 2: user chose the
            importFT => createNewFactType
            createNewFactType(chosenFT);
            return;
        }
```

```
    }
```

A.2.7 mergeFactTypes

```
private void mergeFactTypes(FactType toMerge, FactType dbType){
        ArrayList<CMTField> toMergeFields = toMerge.getFields();
        ArrayList<CMTField> dbFields = dbType.getFields();
        ArrayList<CMTField> fieldsToCreate = (ArrayList<CMTField>)
            toMergeFields.clone();
        for (CMTField importField : toMergeFields) {
            boolean alreadyExists = false;
            for (CMTField dbField : dbFields) {
                if ((importField.getType().equals(dbField.getType())
                        && importField.getName().equals(dbField.getName())))
                            {
                    alreadyExists = true;
                }
            }
            if(alreadyExists){
                fieldsToCreate.remove(importField);
            }
        }
        ArrayList<CMTField> checkedFieldsToCreate =
                checkSameNameDifferentType(dbFields, fieldsToCreate);
        CMTCore core = CMTCore.get();
        FactType updatedFactType = core.addFieldsToFactTypeFact(dbType,
            checkedFieldsToCreate);
        return updatedFactType;
    }
```

A.2.8 doSolveFact

```
    public void doSolveFact(IFactTypeSuggestions iSuggs,...) {
        Fact importF = (Fact) iSuggs.getImportIFactType();
        Fact chosenF = (Fact) iSuggs.getChosenSuggestion();
        // CASE 1 user chose an existing fact (if we can find it in de db)
        FactType ftcheck = CMTDelegator.get().getFactTypeWithName(chosenF.
            getClassName());
        if (ftcheck != null) {
            HashSet<Fact> dbFacts = CMTDelegator.get().
                getFactsWithTypeInFactForm(chosenF.getClassName());
            Fact fcheck = null;
            for (Fact f : dbFacts) {
                if (f.semanticEquals(chosenF)) {
                    fcheck = f;
                    break;
                }
            }
            if (fcheck != null) {
                Integer index = iSuggs.getIndex();
                indexToToFillInBlocks.replace(index, chosenF);
            }
        } else if (importF.semanticEquals(chosenF)) {     // CASE 3: user
            chose the importFact
            if (ftcheck == null) {
                FactType ftFact = createFactTypeFromFact(chosenF);
                CMTDelegator.get().registerFactType(ftFact);
            }
            CMTDelegator.get().addFactInFactFrom(chosenF);
    }
```

A.2.9 doSolveEventInput

```
public void doSolveEventInput(IFactTypeSuggestions iSuggs, ...) {
        EventInput importEI = (EventInput) iSuggs.getImportIFactType();
        EventInput chosenEI = (EventInput) iSuggs.getChosenSuggestion();
        FactType ET = iSuggs.eventType;

        if (importEI.equals(chosenEI)) {  // CASE1: user chose importEvent
            createNewEventType(ET);
        } else {     // CASE 2: user chose a suggestion
                Integer index = iSuggs.getIndex();
                indexToToFillInBlocks.replace(index, chosenEI);
        }
    }
```

Bibliography

[1] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

[2] Audrey Sanctorum and Beat Signer. Towards User-defined Cross-Device Interaction. In *Proceedings of DUI 2016, 5th Workshop on Distributed User Interfaces*, Lugano, Switzerland, June 2016.

[3] Daniel Camps-Mur, Andres Garcia-Saavedra, and Pablo Serrano. Device-to-device communications with Wi-Fi Direct: overview and experimentation. *Wireless Communications, IEEE*, 20(3):96–104, 2013.

[4] Tom Van Cutsem. *Ambient references: Object designation in mobile ad hoc networks*. PhD thesis, 2008.

[5] Bill N Schilit and Marvin M Theimer. Disseminating active map information to mobile hosts. *Network, IEEE*, 8(5):22–32, 1994.

[6] D Abowd, Anind K Dey, Robert Orr, and Jason Brotherton. Context-awareness in wearable and ubiquitous computing. *Virtual Reality*, 3(3): 200–211, 1998.

[7] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, 1993.

[8] Cory D. Kidd, Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth D. Mynatt, Thad Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of CoBuild 1999, Second International Workshop on Cooperative Buildings, Integrating Information, Organization and Architecture*, pages 191–198, Pittsburgh, USA, October 1999.

[9] Gregory D Abowd, Anind K Dey, Peter J Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context

and context-awareness. In *Handheld and ubiquitous computing*, pages 304–307. Springer, 1999.

[10] Louise Barkhuus and Anind K. Dey. Is Context-Aware Computing Taking Control away from the User? Three Levels of Interactivity Examined. In *Proceedings of UbiComp 2003, 5th International Conference on Ubiquitous Computing*, pages 149–156, Seattle, USA, October 2003.

[11] Thomas Strang and Claudia Linnhoff-Popien. A Context Modeling Survey. In *1st International Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004*, Nottingham, UK.

[12] Sandra Trullemans and Beat Signer. A Multi-layered Context Modelling Approach for End Users, Expert Users and Programmers. In *Proceedings SERVE 2016, First International Workshop on Smart Ecosystems cReation by Visual dEsign*, pages 36–41, Bari, Italy, June 2016.

[13] Nissanka Bodhi Priyantha. *The cricket indoor location system.* PhD thesis, Massachusetts Institute of Technology, 2005.

[14] Guang-yao Jin, Xiao-yi Lu, and Myong-Soon Park. An indoor localization mechanism using active RFID tag. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing 2006*, volume 1, pages 4–pp, Taichung, Taiwan, June 2006.

[15] W. Keith Edwards and Rebecca E. Grinter. At Home with Ubiquitous Computing: Seven Challenges. In *Proceedings of Ubicomp 2001, Ubiquitous Computing, Third International Conference on Ubiquitous Computing*, pages 256–272, Atlanta, USA, September 2001.

[16] Matthew B Hoy. If this then that: An introduction to automated task services. *Medical reference services quarterly*, 34(1):98–103, 2015.

[17] Guanling Chen, David Kotz, et al. A survey of context-aware mobile computing research. Technical report, Department of Computer Science, Dartmouth College, 2000.

[18] Victoria Bellotti and Keith Edwards. Intelligibility and accountability: human considerations in context-aware systems. *Human–Computer Interaction*, 16(2-4):193–212, June .

[19] Bob Hardian, Jadwiga Indulska, and Karen Henricksen. Balancing autonomy and user control in context-aware systems-a survey. In *Fourth Annual IEEE International Conference on Pervasive Computing and*

Communications Workshops, PERCOMW 2006, pages 51–56, Pisa, Italy, March 2006.

[20] Brian Y Lim and Anind K Dey. Assessing demand for intelligibility in context-aware applications. In *Proceedings of UbiComp 2009, the 11th international conference on Ubiquitous Computing*, pages 195–204, Orlando, USA, September 2009.

[21] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, November 2010.

[22] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, December 2007.

[23] Tao Gu, Xiao Hang Wang, Hung Keng Pung, and Da Qing Zhang. An ontology-based context model in intelligent environments. In *Proceedings of communication networks and distributed systems modeling and simulation conference*, volume 28, pages 270–275, San Diego, USA, July 2004.

[24] Jakob E Bardram. The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Proceedings of PERVASIVE 2005, Third International Conference on Pervasive Computing*, volume 3468, pages 98–115. Munich, Germany, May 2005.

[25] Jongmoon Park, Hong-Chang Lee, and Myung-Joon Lee. JCOOLS: a toolkit for generating context-aware applications with JCAF and DROOLS. *Journal of Systems Architecture*, 59(9):759–766, November 2013.

[26] Paul Browne. *JBoss Drools business rules*. Packt Publishing Ltd, April 2009.

[27] Anind K Dey, Gregory D Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-computer interaction*, 16(2):97–166, October 2001.

[28] Abraham Bernstein, Mark Klein, and Thomas W Malone. Programming the global brain. *Communications of the ACM*, 55(5):41–43, April 2012.

[29] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, December 2009.

[30] Jose Danado and Fabio Paternò. Puzzle: A mobile application development environment using a Jigsaw metaphor. *Journal of Visual Languages & Computing*, 25(4):297–315, June 2014.

[31] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter Åkesson, Boriana Koleva, Tom Rodden, and Pär Hansson. "Playing with the Bits" User-configuration of Ubiquitous Domestic Environments. In *Proceedings of UbiComp 2003, 5th International Conference on Ubiquitous Computing*, pages 256–263, Seattle, USA, October 2003.

[32] Timothy Sohn and Anind Dey. icap: an informal tool for interactive prototyping of context-aware applications. In *Extended abstracts of the 2003 Conference on Human Factors in Computing Systems, CHI 2003*, pages 974–975, Fort Lauderdale, USA, April 2003.

[33] Jisoo Lee, Luis Garduño, Erin Walker, and Winslow Burleson. A tangible programming tool for creation of context-aware applications. In *Proceedings of UbiComp 2013, The 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 391–400, Zürich, Switzerland, September 2013.

[34] Luigi De Russis and Fulvio Corno. Homerules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2109–2114, Seoul, Korea, April 2015.

[35] Natural language understanding in home automation, 2016. URL `http://kitt.ai/`. Retrieved on July 5th, 2016.

[36] Ryan Dube. How to use google now to automate your home and life, 2016. URL `http://www.makeuseof.com/tag/use-google-now-automate-home-life/`. Retrieved on July 5th, 2016.

[37] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of CHI 2014, CHI Conference on Human Factors in Computing Systems*, pages 803–812, Toronto, Canada, April 2014.

[38] Claudio Bettini, Oliver Brdiczka, Karen Henricksen, Jadwiga Indulska, Daniela Nicklas, Anand Ranganathan, and Daniele Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161–180, April 2010.

[39] Paul Prescod. Roots of the REST/SOAP Debate, booktitle = Proceedings of the Extreme Markup Languages® 2002 Conference, 4-9 August 2002, Montréal, Quebec, Canada, year = 2002, month = August, address = Montréal, Canada.

[40] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Proceedings of IJCAI 2003, Workshop on Information Integration on the Web, IIWeb 2003*, pages 73–78, Acapulco, Mexico, August 2003.

[41] Thorvald Sørensen. A method of establishing groups of equal amplitude in plant sociology based on similarity of species and its application to analyses of the vegetation on danish commons. *Biologiske Skrifter. K. Danske videnskabernes Selskab*, 5:1–34, 1948.

[42] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[43] Richard W Hamming. Error detecting and error correcting codes. *Bell System technical journal*, 29(2):147–160, 1950.

[44] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, February 1966.

[45] William E. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research*, pages 354–359, Washington DC, USA, 1990.

[46] Temple F Smith and Michael S Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[47] *Zigbee Specification*. ZigBee Alliance, 2400 Camino Ramon, San Ramon, USA, January 2007.

[48] Niels Thybo Johansen et al. *Z-Wave Protocol Overview*. Z-Wave Alliance, 47467 Fremont Boulevard, Fremont, USA, June 2011.

[49] Carles Gomez and Josep Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, June 2010.

[50] Preeti Badar and Urmila Shrawankar. Human mood detection for human computer interaction. *CoRR*, 1305.2827:4–8.

[51] Hans-Werner Gellersen, Albrecht Schmidt, and Michael Beigl. Multi-Sensor Context-Awareness in Mobile Devices and Smart Artifacts. *MONET*, 7(5):341–351, 2002.

[52] Mae Keary. The Internet of Things (The MIT Press Essential Knowledge Series). *Online Information Review*, 40(3):449–450, May 2016.

[53] Charith Perera, Arkady B. Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys and Tutorials*, 16(1):414–454, February 2014.

[54] Meghan Finch. Using Zapier with Trello for electronic resources troubleshooting Workflow. *The Code4Lib Journal*, 26, 2014.

[55] Ying Xu and Fu-yuan Xu. Research on Context Modeling Based on Ontology. In *2006 International Conference on Computational Intelligence for Modelling Control and Automation (CIMCA 2006), International Conference on Intelligent Agents, Web Technologies and Internet Commerce (IAWTIC 2006)*, page 188, Sydney, Australia, November 2006.

[56] Brian Y. Lim, Anind K. Dey, and Daniel Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of CHI 2009, The 27th International Conference on Human Factors in Computing Systems*, pages 2119–2128, Boston, USA, April 2009.