Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Graphical formalization and automated computing of safety constraints in robotics

Ludwig Nägele

Masterarbeit im Elitestudiengang Software Engineering

Software Engineering
Elite Graduate Program



Institut für Software & Systems Engineering
Universitätsstraße 6a D-86135 Augsburg

Graphical formalization and automated computing of safety constraints in robotics

Matrikelnummer:	1168616
Beginn der Arbeit:	06. Juni 2012
Abgabe der Arbeit:	16. November 2012
Erstgutachter:	Prof. Dr. Wolfgang Reif
Zweitgutachter:	Prof. Dr. Bernhard Bauer
Betreuer:	M.Sc. Andreas Angerer
	Prof. Bruce MacDonald, Ph.D.

Software Engineering
Elite Graduate Program

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 16. November 2012 Ludwig Nägele

Zusammenfassung

Robotik-Anwendungen sind heutzutage nicht mehr nur in der Industrie zu finden, sondern zunehmend auch im menschlichen Bereich und Lebensumfeld, wo die Roboter in direkter Nähe zu Menschen arbeiten. Auch die Programmierung solcher Roboter wird nicht mehr zwangsläufig von Softwareentwicklern erledigt. Der Kunde möchte das Verhalten der Roboter je nach deren Einsatzgebiet individuell selbst definieren und anpassen können. Doch solche Roboter im Umfeld von Menschen stellen aufgrund der Verletzungsgefahr ein Sicherheitsrisiko dar, zumal wenn das Roboterverhalten nicht von geschulten Ingenieuren entwickelt wurde. Daher ist nach Lösungen zu suchen, wie man mit Sicherheitsanforderungen für Roboter im Zusammenhang mit kundeneigener Programmierung umgeghen soll.

Mit dieser Arbeit wird eine visuelle Sprache vorgestellt, die der Erstellung von Sicherheitseigenschaften für zustandsbasierte Roboterprogramme dient. Bei der Entwicklung dieser Sprache wurde besonders darauf Wert gelegt, dass auch Laien, die insbesondere nicht mit Sicherheitsaspekten vertraut sind, damit umgehen und arbeiten können. Dazu wurde zu einer mathematischen temporalen Beschreibungssprache eine intuitive visuelle Darstellung erarbeitet, die einem größeren Personenkreis Zugang zu Methoden der Programmverifikation verschaffen soll. Außerdem wird in dieser Arbeit ein neues Konzept vorgestellt, das mittels einer automatischen Bestimmung von Sicherheitsbedingungen den Entwickler bei der Programmierung unterstützt. Es soll dem Entwickler dabei helfen wichtige Bedingungen zu finden, und letztendlich die Wahrscheinlichkeit erhöhen, dass Programmfehler gefunden werden.

Ein graphischer Editor wurde entwickelt, der die visuelle Sprache zur Definition und Bearbeitung von Sicherheitsbedingungen wie auch die automatisierte Bedingungsfindung integrierend zusammenführt und gemeinsam nutzbar macht.

Abstract

Nowadays robotic applications are no longer confined to only industrial use but increasingly assigned to tasks within human workspaces and living environments. Even the programming of robots is shifting from experienced software developers to the consumers themselves who need to customize the robot's behaviour in order to match individual requirements. However, both human robot collaboration as well as the robot's behaviour definition by non experts may be a risk to humans within the robots' environment. Thus, special thinking is needed about how to cope with safety requirements for robot behaviour specification by consumers.

In this work a visual language for the defining of safety constraints for state machine definition of robot behaviour is presented. The approach addresses mainly non-safety-experts and introduces abstraction from a mathematical temporal logic expression to a more intuitive visual representation to enable a wider range of software developers to do model checking. Furthermore, a new concept of development support by providing automated constraint generation functionality is proposed. It aims to help developers in defining reasonable constraints and finally to increase the probability of finding bugs.

A graphical editor has been developed which integrates and demonstrates both visual constraint creation functionality as well as automated constraint generation and makes them usable in an uniform manner.

Acknowledgment

For my master thesis it was my wish to do research abroad in a foreign country. Professor Wolfgang Reif, supervisor of my thesis, supported me in my plan and managed to give me the unique possibility to visit New Zealand for half an year. Therefore I am deeply grateful to him.

My thesis project was hostet at the University of Auckland, New Zealand, by Professor Bruce MacDonald. I want to thank him for the opportunity to stay and to work at his robotics lab, for all the meetings and discussions we had and for all the support he gave to me.

I also want to thank Felix Kaser, a fellow student of mine, who was with me in New Zealand. During several coffee breaks we frequently discussed ideas for this work and he helped me a lot with his good suggestions.

Chandan Datta, PhD student at the University of Auckland and developer of Robostudio, always offered his help regarding the use of Robostudio or general questions about the healthcare robotics project. He furthermore supported me in getting familiar with the lab and its research topics and with Auckland as a city to live in. Especially in the first time after my arrival in New Zealand I was very grateful for this.

Sincere thanks is also given to Alwin Hoffmann and my supervising tutor Andreas Angerer, both doctoral candidates at the University of Augsburg, Germany, for assisting me by discussions and hints. They did not only inspect and correct the thesis writing regularly, but they also assisted me in finding the thesis topic in the very beginning and contributed to the results of this work decisively by providing ideas and feedback during the research phase.

Last but not least I want to thank my father Rudolf Nägele for all the effort he put

in reading and correcting my work. Since this work relates to different mathematical concepts and formalisms such as LTL, model checking, RBDL, etc., he had to spend plenty of hours in front of the computer in order to get familiar with all these concepts and to check this work for plausibility and correctness.

Contents

List of Figures

Listings

1 Introduction

Safety critical robot applications require extensive testing or formal verification in order to achieve adequate safe and predictable behaviour. Even if a program does not control actuators itself and thus cannot directly cause any damage, its output might still be a trigger for dangerous actions involving several actors. Imagine a medication reminder application running on a service robot which guides a person through the process of medication intake, as investigated in the healthcare robotics project at the University of Auckland, New Zealand [5]. All interaction is guided by showing textual instructions on a display, and by speech generation. An error in the program logic could cause an already taken medication to be reminded again and may injure a patient with Alzheimer's disease by causing a medication overdose. Scientists are working on applications such as the medication reminder [6] which run on mobile robots (so called healthbots) serving in nursing homes for the elderly. Equipped with a touch display and several external devices such as a blood pressure measurement tool the robots aim to support and entertain the elderly in their daily activities [5, 7].

Figure 1.1: Service robot being used by older person [1].

With service robots coexisting with people and acting within their workspaces, coping with reliability and safety issues is essential for trust and acceptance of these robots. In some cases safety certifications based on international standards for functional safety might be even required before robots get released for use or obtain concession for sale. One such international standard is IEC 61508 [8], a basic functional safety standard targeting all kinds of industry. ISO 13482 [9, 10] is still under development and is intended to be a harmonised European standard for safety requirements for robots in personal care applications. However, considering functional safety while developing service robots is both complex and difficult; maybe one reason for the absense of safety checks and assurances in the current healthbot versions at the University of Auckland. But facing the fact that safety requirements for service robots are expected to increase in future, there is a plan of slowly integrating safety functionality into the healthcare robotics project as a next step starting with the concepts introduced in this paper.

The applications for the healthcare robots mentioned above are intended to be developed by healthcare professionals using Robostudio [1], a visual programming environment for rapid authoring and customization of complex robot services. Generally these people do not have profound knowledge of the complex mathematical syntax used for formal methods which allow software behaviour verification. Nevertheless, certain safety guarantees must somehow be delivered in order to gain users' trust and to prevent any harm or injuries by the healthcare robots. In case of the medication reminder mentioned above it might be important to ensure that notifications are sent to staff members whenever medication is not taken by the patient, or to avoid duplicate reminders for medication which has already been taken. A mechanism like this is needed for the specification of such safety constraints; one that does not require special or expert knowledge and is thus easy to use for all kind of programmers. Within the scope of this work a visual language has been developed intending to fill this gap.

Unfortunately, supplying an easy-to-use editor for constraint creation does not guarantee adequate thinking about sufficent constraints by the developer. Unmotivated or just unexperienced developers may miss important constraints needed in

order to achieve a certain safety level. To support the user in finding significant constraints it might be useful to automatically compute constraint suggestions which make true testimonies about the current designed program. First of all the generated constraints provide an opportunity for the developer to identify reasonable constraints as well as contradictions between constraints and specification. The latter would mean there are errors in the program which lead to undesired constraints. Furthermore, once constraints are created and checked for sanity they can be validated after every program change and thus ensure integrity during the entire development process. This work introduces a new heuristics for finding such reasonable safety constraints based on state machine definitions that describe robots' behaviours.

Both ideas, the viual language and the automated constraint generation, have been developed and integrated by the author into Robostudio in order to simplify respectively facilitate the use of safety mechanisms for application development for service robots. Thereby the desired safety certification for the healthcare service robots can move closer by one step.

This work starts with background information about the healthcare domain, short explanations of common safety mechanisms and insights into related work in chapter 2. After the goals have been described in chapter 3, the visual formalisms and the automated constraint generation are introduced in chapters 4 and 5. A prototype implementation is demonstrated in chapter 6, and a test scenario is described in chapter 7 where the presented tools get evaluated. Chapter 8 gives a short conclusion and an outlook how the presented work could be continued.

2 Background

The topic of this work evolved from the situation that the healthcare robotic applications of the University of Auckland should be equipped with mechanisms for ensuring safety. The motivation for it is described in section 2.1 as well as the robots used, the kind of applications running on them, and their technical background.

The visual programming environment used for the programming of these robots is presented in section 2.2. For the understanding of safety concepts which this work is based on, a short introduction about functional safety is given in section 2.3. In section 2.4 other works and programs related to this work are examined.

2.1 Healthbot application

In some countries, among New Zealand and Germany, the average age of population is constantly increasing [11, 12]. Owing to excellent achievements in medicine as well as higher living standards, humans can enjoy longer lifes. Also decreasing birth rates contribute to an upward drift of the dominating age range. As a result, the industrial sectors of professional health care and social institutions are required more than ever. However, a strong deficit of professionals is denoted in the healthcare branch and availability is far behind demand. One reason for this might be below average low salaries for employees in the healthcare sector. But, just raising the salary will not lead to the desired results since the majority of the older people can not even afford a nurse to today's prices.

Aiming for fillig the constantly increasing gap in professional personnel for healthcare of the elderly and bypassing the financial burden, service robots are supposed to undertake more and more the task of care. There are lots of possible tasks a robot

could do such as taking the blood pressure, guiding medication intake, detecting falls or just entertaining with music, video or internet. This challenge as a goal, the healthcare robotics team at the University of Auckland researches this topic and develops service robots for supporting the elderly in their daily life.

The two newest robots IrobiQ and Cafero are shown in figure 2.1. These mobile robots are equipped with a touch display for presenting screens and receiving user input. Furthermore, they are equipped with additional external tools such as the blood pressure measurement device. Applications running on the robots use the touch display to present screen dialogs to the user and use speech synthesis in order to give also accoustic feedback. User inputs or responses from devices cause dialogs to change. Figure 2.2 presents a small example of such a screen-flow based application, which uses response from a camera and user input for navigating through the screen-flow.

Figure 2.1: IrobiQ and Cafero, two robots of the healthcare project.

There is a program interpreter running on the robots which can load and execute program files. Screen dialogs get generated and displayed on the touch screen. The interpreter processes programs written in Robot Behaviour Description Language (RBDL), a domain specific language (DSL) based on XML, which has been developed particularly for the healthcare robotics applications. It allows defining state machine based program behaviour and creating user interface (UI) elements for screen visualizations. An example implementation of one state in RBDL is shown in listing 2.1.

For each state of the program, background actions can be defined. They are transparent to the user and run in background. Text messages can be sent or external

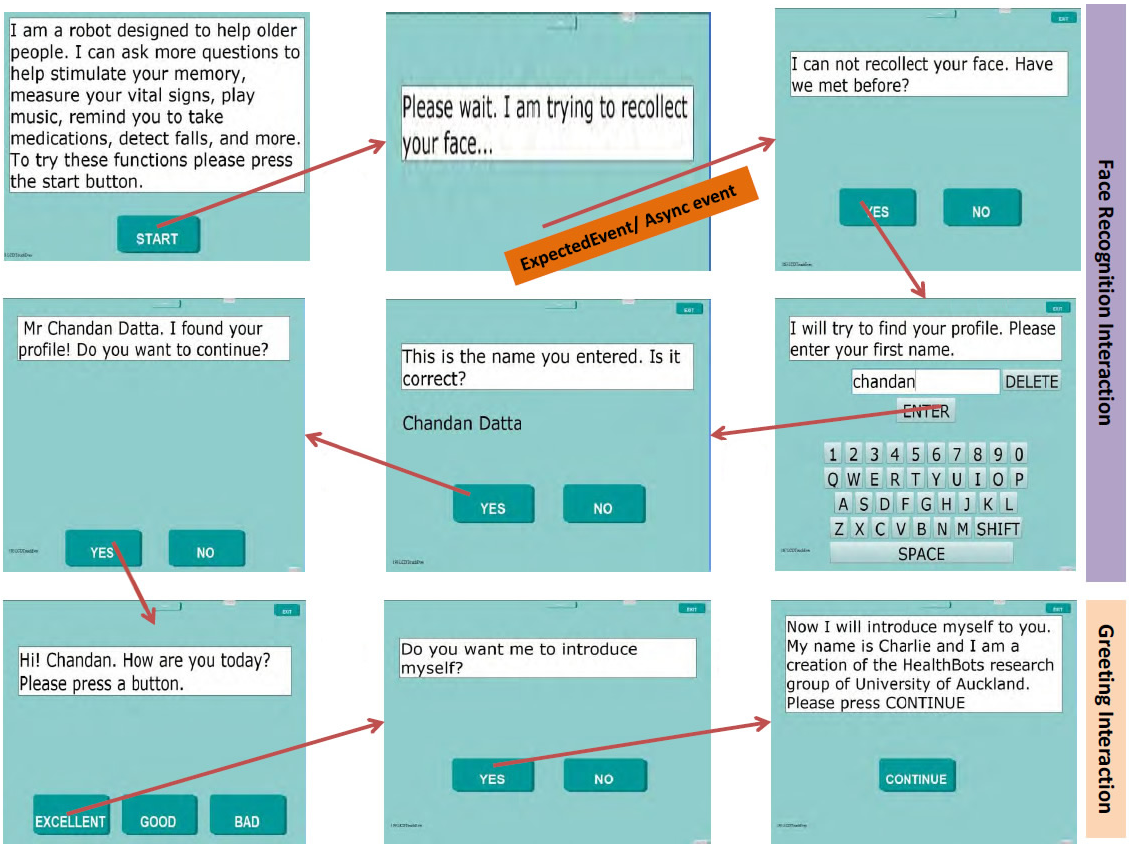

Figure 2.2: Interaction example based on robot screen-flow dialogs [1].

```
<state no="84">
  <backgroundactions>
    <rocosmessage type="EVENT MESSAGE" senderid="RCP" receiverid="NA"
       name="FDSessionStart">
      <parameter vartype="text" type="unsigned int" name="id">1</
         parameter>
      <parameter type="string" name="" />
    </rocosmessage>
  </backgroundactions>
  <screen>
    <components>
      <button label="START" width="250" height="100" x="387" y="600"
         textsize="40">
        <event name="clicked">
          <action preconditions="no" name="transition">
            <parameter>
              <type>state</type>
              <name>n</name>
              <value>86</value>
            </parameter>
          </action>
        </event>
      </button>
    </components>
  </screen>
  <expectedevents>
    <event name="TimeOut">
      <action preconditions="no" name="transition">
        <parameter>
          <type>state</type>
          <name>n</name>
          <value>VARFirstScreen</value>
        </parameter>
      </action>
    </event>
  </expectedevents>
</state>
```

Listing 2.1: Example of a state defined with RBDL [1].

devices can be accessed, for example. The latter is realized with web services [13] to obtain a modular design. Some states can also have screen definitions with buttons, messages, videos, etc., on it. These are displayed to the user when the respective state is active. Once displayed, buttons can be clicked and trigger events. The behaviour of events can be mapped in the expectedevents tag. It lists all events which are processable by the current state and defines the corresponding behaviour such as triggering transitions to other states.

Very important for this work is a fundamental feature of RBDL, the ability to define state machine behaviour with states, transitions and events. This fact was the basis for later decisions regarding the applied safety mechanisms.

2.2 Robostudio

In the beginning all programming of the robot behaviour was done by directly editing the RBDL code mentioned in the previous section by using a text editor. However, huge code files and occasional code edits made further maintenance and change requests difficult up to impossible to realise. For this reason, Chandan Datta, PhD student at the University of Auckland, developed a visual programming environment for developing robot behaviour on top of RBDL. This environment - it is called Robostudio [1] - allows to easily edit the program behaviour on a visual layer and generates the corresponding program file containing the RBDL code fully automatically. This file can be downloaded to the robot and executed by the interpreter. Figure 2.3 illustrates the steps of robot behaviour development.

Robostudio is an editor fully written in Java and implemented as a NetBeans rich client application [14]. Several windows and views provide visual tools for editing and understanding robot behaviour as well as screen dialog design. As depicted in figure 2.4, the editor integrates a *State Navigator Window* (1) in the top left corner where all existing states are listed. Below an overview about all incoming and outgoing transitions is given. New states can be created and existing states can be deleted. A click on one particular state id in the *State Navigator Window* will cause all other components to show all possible information regarding the selected state. The

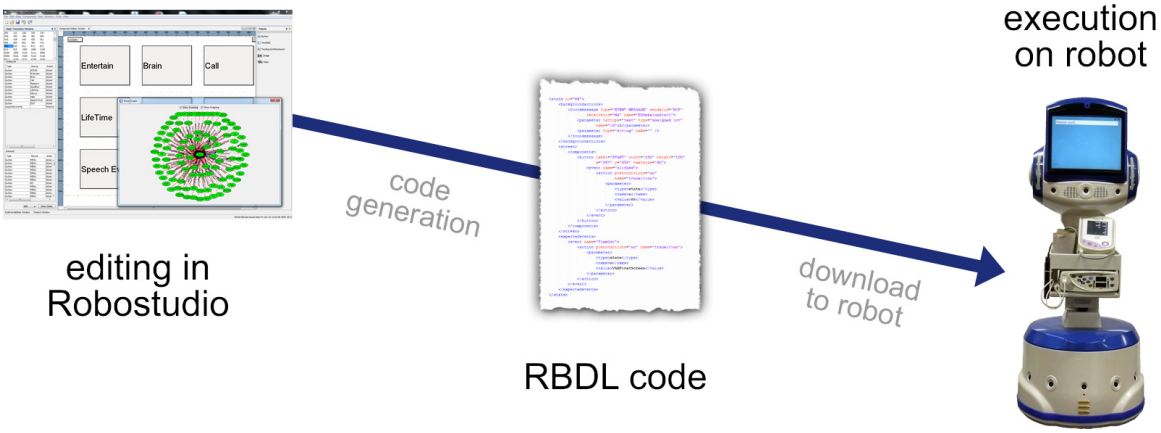

Figure 2.3: How Robostudio is used for robot behaviour development.

Expected Events Editor Window (2) in the bottom left can be used for specifying the expected events and their effects. Heart and center of the editor is the *UI Component Layout Editor Window* (3). It renders the screen dialog of the selected state and gives a preview of the screen presented on the robot during runtime. Components such as buttons, text boxes or images can be added, changed or removed via drag and drop. They are provided by the *UI Components Palette Window* (4) on the right side where the developer can choose from all supported UI components. The *Background Actions Editor Window* (5) lets the developer define background actions, which are executed transparently such as accessing external devices or sending messages. Aiming to support the developer with helpful graphical tools, the *State Transition Visualization Window* (6) gives an overview about all connections towards and from the selected state. It helps to quickly get an understanding of the local program behaviour.

2.3 Behaviour description and analysis

The use and application of software is versatile and varies from text editing programs running on personal computers, over control software for high-precision applications in aeronautics, right up to trigger mechanisms for deathly weapons. For most of them safety properties are irrelevant, but a few are ranked highly dangerous to their

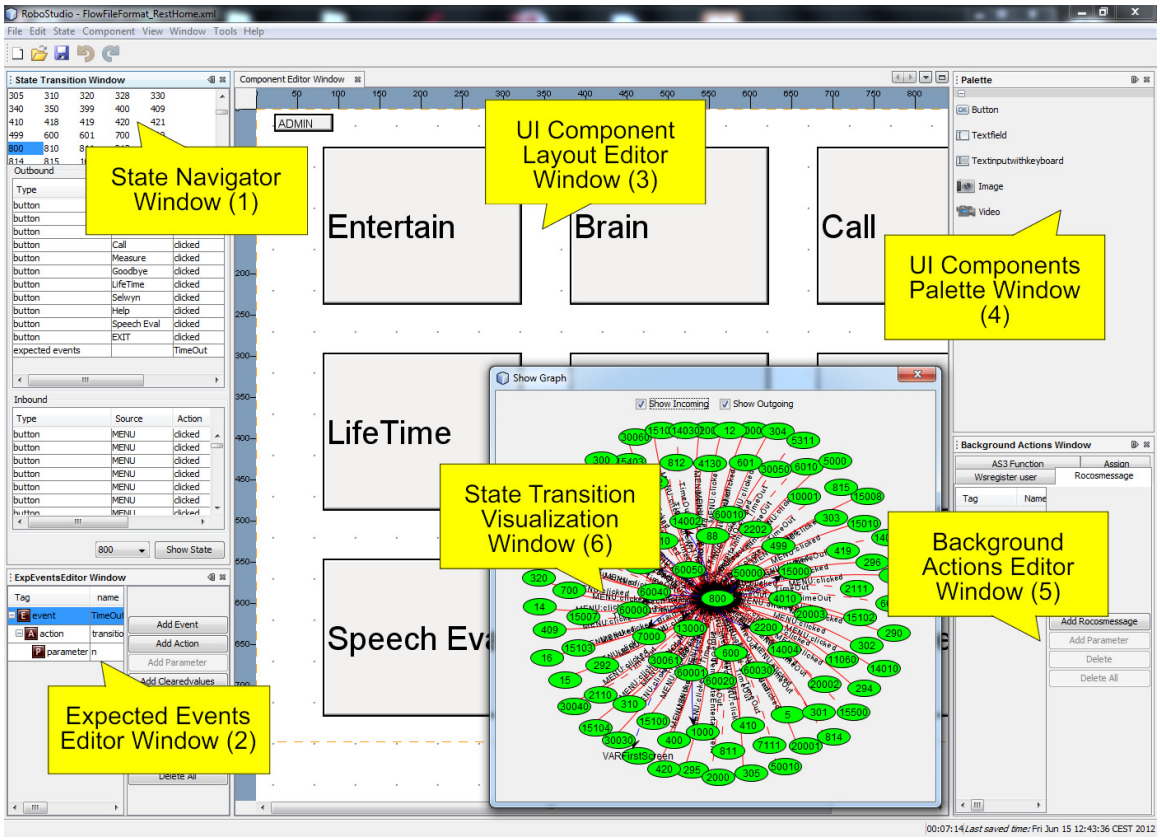

Figure 2.4: Window composition in Robostudio.

environment and also to human lifes. In order to eliminate software malfunction, there are different mathematical concepts for verifying software behaviour. One of them is model checking which allows checking state machine programs for functional safety. Since the healthbot applications mentioned in section 2.1 are realized with state machines, the concept of behaviour checking is applicable and important for this kind of programs. At first the basic idea of state machines is described in section 2.3.1, followed by an introduction into model checking in section 2.3.2.

2.3.1 State machines

A state machine is a model which can describe the behaviour of a system. It contains an aggregation of all different statuses of a system and possible changeovers from one status to another.

Each system status is represented by one *state* which stands for a particular and unique constellation of system variables and properties. One state can be declared as *start state* which is the very first state when the system starts working. The state representing the current system status is called *current state*. The current state changes (maybe into itself) whenever there is a modification to the system. Such transfers of the system from the current state to another one are defined by *transitions*. Thus, they define the possible system behaviour. All influences of the environment which cause a change to the system are called *events*. They trigger transitions which transfer the system into a new (current) state. The fact that a state machine reacts to events is the reason why state machines are also known as *reactive systems*.

State machines can be visualized, as shown in figure 2.5 which gives a small example of a state machine. All states are represented by circles which contain their names. Transitions are drawm as arrows between source and destination states, and the regarding trigger event names are written next to it.

The following example clarifies the concept of state machines. Assuming a door which can be opened, closed and locked by a key-operated bolt lock, the behaviour can be reduced to a state machine as shown in figure 2.6. Four states represent the “system statuses” of a lockable door: closed but unlocked (closed), opened and

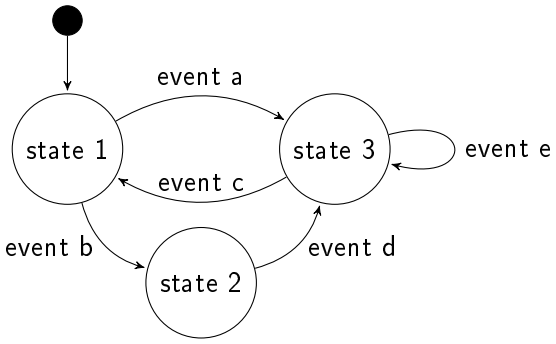

Figure 2.5: General example for a state machine.

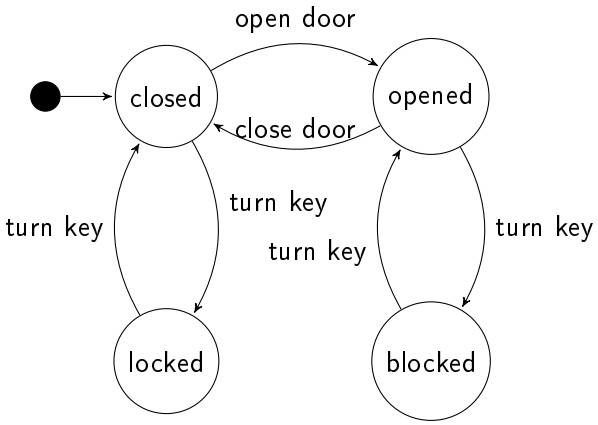

Figure 2.6: Door behaviour as a state machine.

unlocked (opened), closed and locked (locked), opened but locked (blocked). The latter case is when the extended bolt prevents the door from being closed.

The black circle defines the point of entry for this state machine, the door is closed but unlocked in the beginning. By an event of the environment - in this case a human acting at the door - the door can be opened, which turns "opened" into the current state, and closed later on again. A turn of the key locks respectively blocks the door and no further opening or closing is possible. A second turn of the key unlocks respective unblocks the door again.

The short description given in this section does not contain a complete explanation about the concept of state machines at all, but it gives a basic overview which is needed as background information for this work. Also the explanation of state machines above is no formal definition; it can be found in literature [15, 16].

2.3.2 Model checking & LTL

Whenever a software or hardware system has a special requirement regarding robustness and reliability, different formal methods are available to be used in order to verify the design. Model checking [17] is such a method which allows automatically verifying a system description against a specification by means of an exhaustive search for all states which could possibly be visited by the system during its execution. As a result, either the system description fulfills all propositions or a counter example is provided which gives evidence about a broken property. Figure 2.7 gives a schematic view about the concept of model checking.

For representing the system description - which can be a computer program, for example - a formal language is used. Also state machines are applicable to serve as a formal system description. In contrast, the specification used for model checking contains certain properties that express the desired and safe system behaviour in form of mathematical formulas. Linear temporal logic (LTL) [17] is an example for a logic that is used to define such formulas on state machines. It is a temporal logic which can express formulas concerning the future of paths, for example, that something will eventually happen.

LTL is based on propositional logic, thus, it can express propositional variables (p_1,

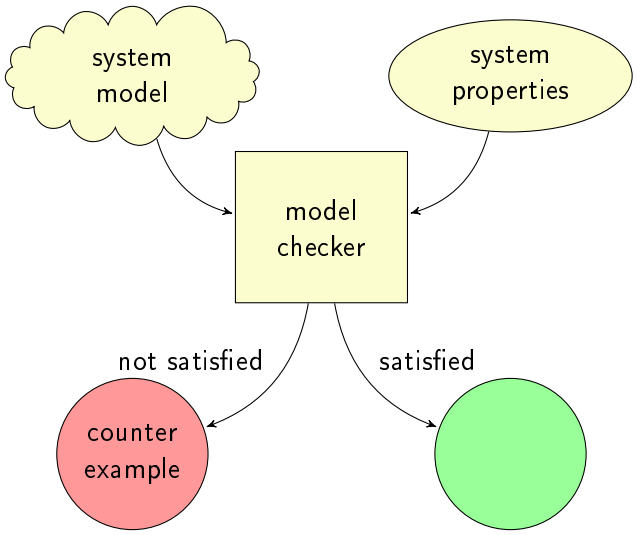

Figure 2.7: Concept of model checking.

p_2, ...) and logical connectives ($\wedge$, $\vee$, $\neg$, $\rightarrow$ and $\leftrightarrow$). Additionally further temporal modal operators are provided which enable propositions concerning the future of paths, among three unary operators:

$\Box\varphi$ This formula expresses that φ is true now and in all following states. The meaning of the rectangle is "GLOBALLY".

$\bigcirc\varphi$ With a circle representing "NEXT", φ will be true in the next state.

$\Diamond\varphi$ At least one time - now or eventually in a later state - φ is or will be true. The diamond stands for "FUTURE".

Furthermore, LTL provides binary operators as follows:

$\varphi\mathcal{U}\psi$ The "UNTIL" operator makes φ true at least until the first occurrence of a state where ψ is true. Besides, ψ will eventually be true.

$\varphi\mathcal{R}\psi$ It expresses that ψ is true at least as long as there is no occurrence of a state where φ is true. Its meaning is φ "RELEASES" ψ.

$\varphi\mathcal{W}\psi$ "WEAK UNTIL" represents the same as "UNTIL", except that an occurrence of ψ is not necessary. In this case, φ remains true infinitely.

2.4 Related work

The idea of a visual environment for the defining of system properties is not new and already pursued by some researchers. Although their focus does not completely match the subject of safety for healthcare robots, they present concepts of visualization which might be relevant for this work. In section 2.4.1, the work of Sisirucá and Ionescu is presented. Section 2.4.2 dexcribes a three dimensional concept by Del Bimbo et al., followed by an introduction of HomeTL in section 2.4.3.

2.4.1 Sisirucá and Ionescu

A first step towards a graphical tool for designing safety constraints is taken by Sisirucá and Ionescu [2]. They developed an object-oriented graphical environment for creating temporal logic sentences and rules visually. As shown in figure 2.8, the tool presents all operators as blocks which have inputs and outputs. Whereas

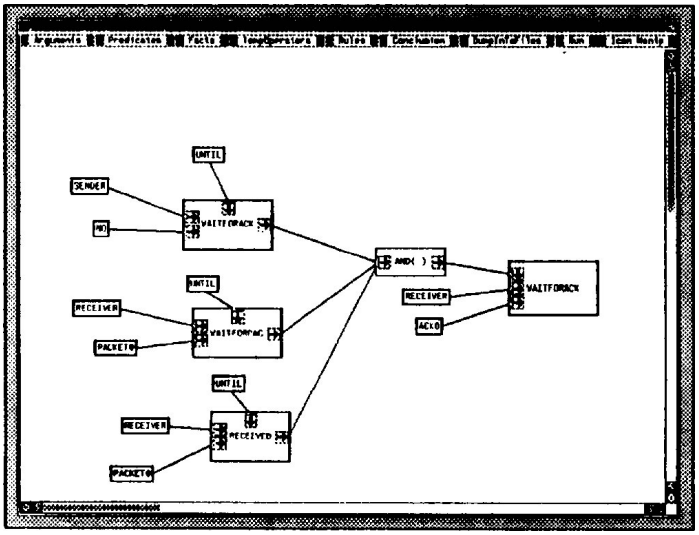

Figure 2.8: The visual programming environment. An example showing the rule construction process [2].

logical operators such as *AND* and *OR* have each one boolean output and several boolean inputs represented by arrows pointing to the right, generic blocks for temporal

expressions additionally have an input symbolized by an arrow pointing downward where the temporal operator itself is to be attached. By adding and connecting the operators a big expression graph can be built. In case that a diagram becomes too complex, an implode feature allows to compress the graph to a single box. It is also applicable for the reuse of diagrams.

The presented environment aims to be an effective tool for developing temporal logic sentences and rules. In fact, it gives a schematic overview of even complex formulas and allows substitution and reuse. The visual formalism is an alienation of LTL which might be suitable for researchers, but probably not for non-experts.

2.4.2 Del Bimbo et al. (3D)

Del Bimbo and his colleagues worked on a visual tool for temporal logic, too, but with a special focus on hierarchical representation of formulas [3]. It is based on computation tree logic (CTL) which is a branching-time logic with support for quantifiers. In contrast to LTL where properties are fulfilled by all possible paths, CTL expresses properties which depend on the future branching, for example, that at least one path exists which has a particular property. As indicated by figure 2.9, the visual tool provides all operators known from LTL except that temporal operators are extended

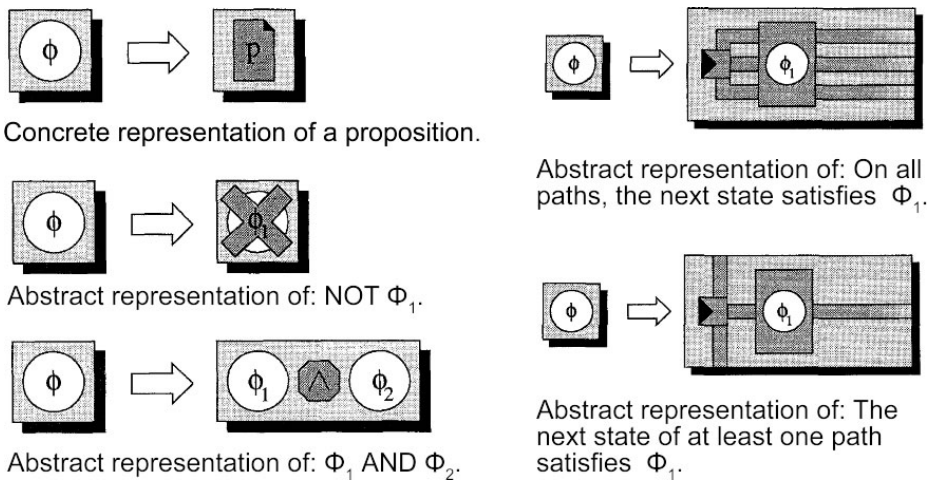

Figure 2.9: Visual representation of some of the operators of Del Bimbo et al. [3].

by an existential quantifier (e.g. “at least on one path...”) or an universal quantifier (e.g. “on all paths...”).

Except propositions all operators are abstract and have placeholders representing their subformulas which can be connected to other operators. The positioning of these operators is organized in layers where the parent operator lays one level above its children. It can be compared to a tree upside down with the topmost operator as root. This formula structure can be visualized as a 3D branching tree within a virtual space. Figure 2.10 shows the result of considerations done by Del Bimbo and his colleagues.

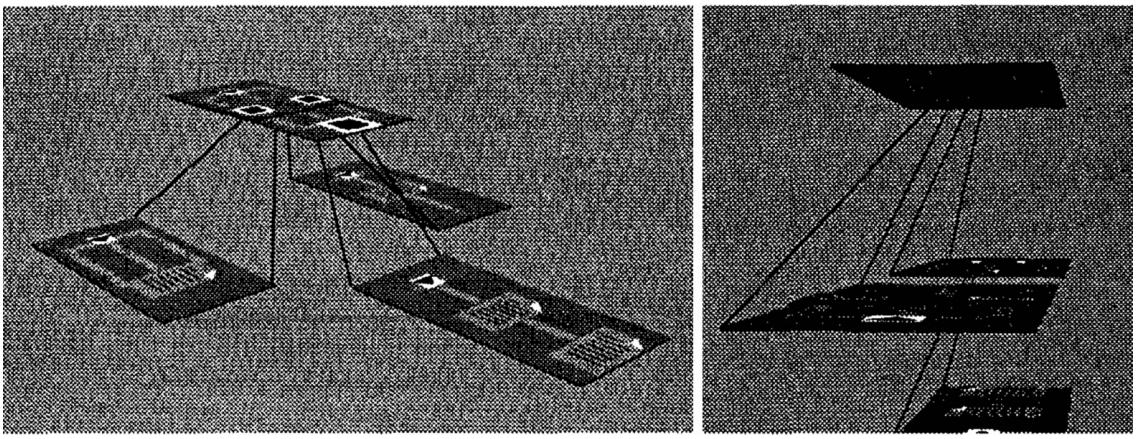

Figure 2.10: Formula trees’ 3D representation within a virtual space [3].

The presented visual formalism directly maps the syntax of the temporal logic onto graphic symbols and composition patterns. The researchers found well expressive graphical representations for all operator especially for the temporal ones, which makes the tool fairly intuitive. The three dimensional visualization gives a good overview of the structure, although it shuld not be used for editing. Computer monitors can naturally display only two dimensions and also input devices such as mouse and keyboard are customised to it. Editing of the third dimension needs special gesture definitions and 3D modelling techniques the most users might not be familiar with. Thus, a 3D representation for editing should only be used if necessary. This perception has been tried to be considered in this work.

2.4.3 HomeTL

The step towards a more abstract level of development is taken by HomeTL [4], a visual formalism for the design of home based care. It allows healthcare professionals to specify rules and sequences of actions in a quite nontechnical manner. Based on these conditions a monitoring system in a patient's home environment can detect and report abnormal situations within the daily routine. The visual notation provides propositional variables and logical connectives, the latter is shown in figure 2.11. In contrast to LTL, the temporal operators of HomeTL use absolute time limits (seconds, for example) which are represented by τ in figure 2.11. Thus, not only the relative order of states is defined but also a time by which a particular action has to happen.

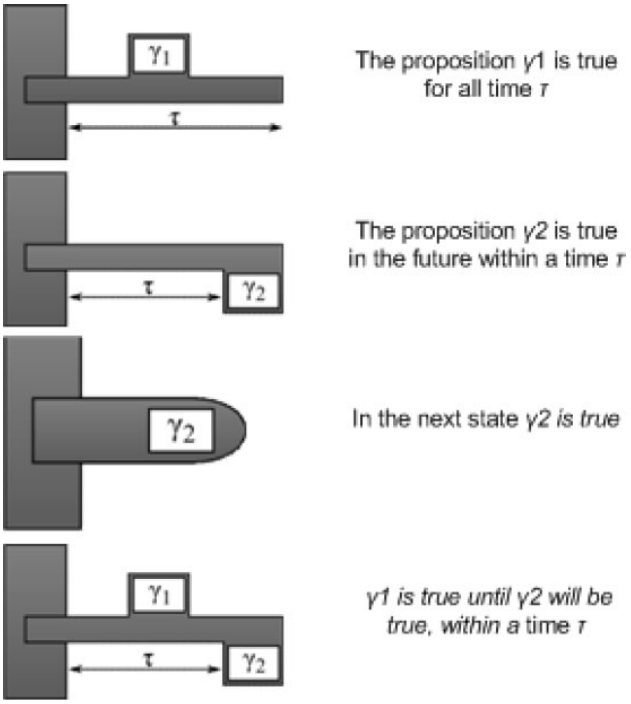

Figure 2.11: Visual notation for temporal operators within HomeTL user interface [4].

This notation gives special meanings to the two dimensions: The horizontal dimension shows the temporal evolution of states, while the vertical dimension (see figure 2.12) expresses the logical relationships at the respective current state. A pro-

gramming environment allows editing and creating of constraints by using the elementary operators shown in figure 2.11. Figure 2.12 presents a snapshot of this editor.

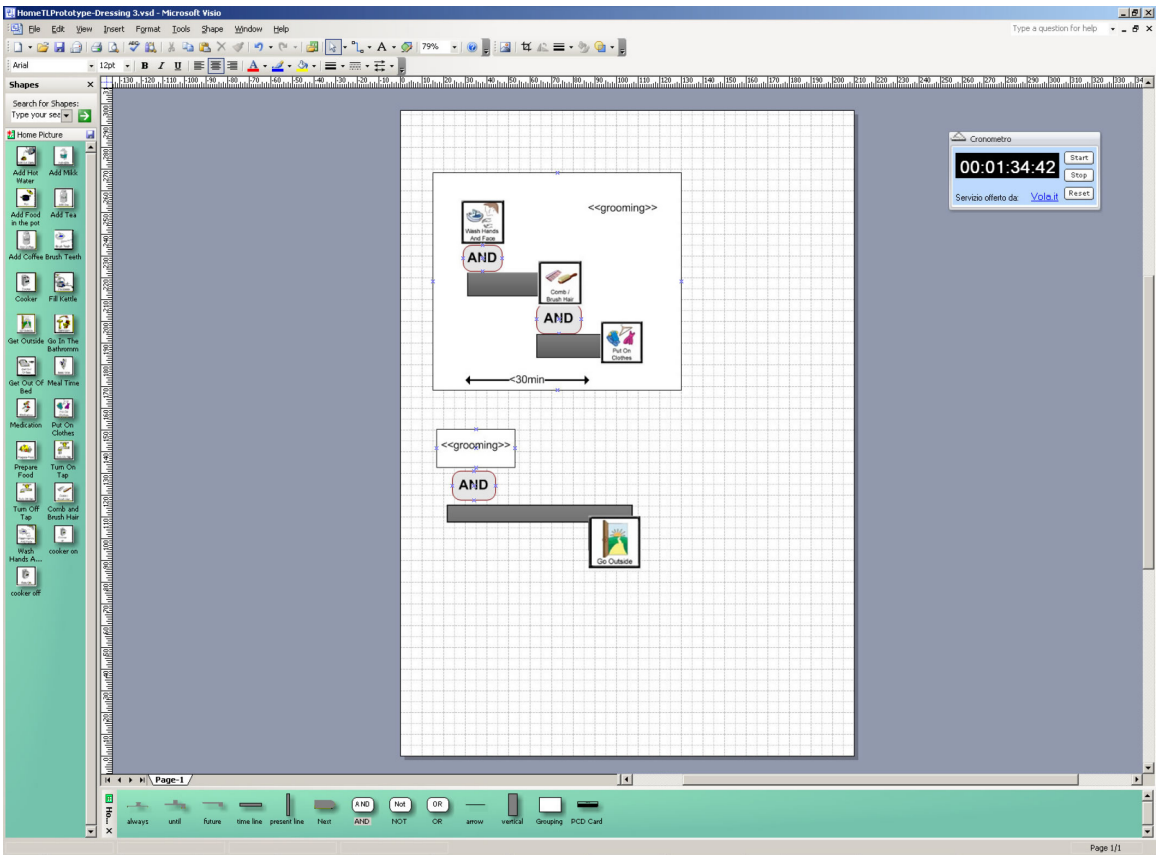

Figure 2.12: HomeTL Visual Editor [4].

Even though this approach touches the subject of healthcare it barely matches the problem stated in this work. HomeTL focuses more on monitoring temporal boundaries of a patient's behaviour than on ensuring functional safety of an implemented program. It does however depict how nontechnical constraint design can be realized and its graphical design is a good example of handling time dependencies.

3 Goals

Based on the purpose of bringing the healthcare applications of the University of Auckland's healthcare project and certain safety standards closer together, and thereby laying the foundation of aspired safety certifications, Robostudio should be extended with essential tools for this purpose.

So the visual programming environment, used for developing robotic applications in the form of state machines, shall provide a new interface in the future, which can be used by developers for evaluating safety properties of robot behaviour. On the one hand, this extension shall be seamlessly integrated in Robostudio. On the other hand, it should provide a clear API which allows an easy integration also into other program environments. Thus, the new concepts are to be universal and not confined to Robostudio.

The main goal is to introduce such a visual language for defining safety constraints in state machine definitions of robot behaviour, that is suitable for people who are very familiar with the desired actions of the robot and the necessary safety conditions but are not experts in formal methods. The visual language should be easy to use and intuitive to read, in order to make the program easy to understand and to facilitate maintainability. Hence, the graphical formalism should abstract from the usually complex and mathematical concept of conventional temporal logic but still be significantly expressive.

Furthermore some mechanisms should be integrated aiming to support application developers in finding suitable constraints for a certain application. It is assumed that applications are modeled by state transition graphs. This modeling paradigm is used in many healthcare robots that employ dialogue systems for communicating with robot users [18]. Other possible fields might be shop assistant robots or entertain-

ment robots. For such systems, the approach is suitable. It is based on heuristically analyzing state transition graphs and deriving constraints related to the graph structure. This work investigates a particular heuristic as a first step towards developer support.

Finally, this heuristics will be applied to the medication reminder example mentioned before in order to retrieve meaningful results. The minimum goal that should be achieved is the identification of the following constraints:

- The application will always eventually check the database for new reminder jobs. Thus, it is ensured that pending reminders are processed.
- Whenever there is a pending reminder, either medication will be finally taken or caregivers will get notified in case of the patient refusing medication intake.

Some of the statements do not make sense if patients lie or pretend to take their medication but do not take it. As the presented approach validates state machine behaviour rather than human attitudes, it is assumed that there is either cooperation from the patient or a separate tool that can verify medication intake. Furthermore the correctness of external services used by the medication reminder such as the database module holding all reminders is not taken into account.

All the functionality mentioned above shall be combined in one editor and subsequently for evaluation purpose be integrated into Robostudio, a programming environment for state machine editing.

4 Visual formalisms for safety constraints

This chapter introduces two different approaches of visual formalisms and states later on the final desing of the visual language. A main difficulty in finding a proper visual formalism is to glean the right level between abstraction and expressiveness since these two characteristics appear to be mutually exclusive in some points. A first approach, the so called template constraint formalism, is described in section 4.1. Subsequently a different design - the operator constraint formalism - is presented in section 4.2. Section 4.3 gives a comparison of the two formalisms presented before and states which one is suitable to be used for the prototype.

4.1 Template constraint formalism

Which visual presentation of constraints would suit a healthbot application developer and what are the main types of safety constraints he could want to check? Led by this question, a visual language based on templates has been developed. These templates make different propositions about visits of states and their relations. Application developers can choose from several graphical constraint types and customize them by parameterising state variables. So far there are two different kinds of template constraints:

1. Whenever state (x) is active, state (y) must be visited before state (z) can be reached.

   In this constraint template, x, y and z are state variables which can be replaced

by real states. The idea behind this constraint is that propositions about temporal dependencies can be made. Under certain circumstances the visit of a particular state is allowed only if a specific precondition is fulfilled. An example constraint for this template is: Whenever a person is eating (x), he must brush his teeth (y) before he can go to bed (z). Thus, going to bed without brushing teeth after meal should never happen.

2. Each visit of state (x) will eventually result in a visit of either state (y) or state (z) [or state (w) [...]]. All of them are actually reachable.

   As before, x, y, z and w are state variables. This constraint type enforces at least one particular event to happen after a specific precondition. An example of such a constraint would be: Every time a person enters a supermarket (x) he will eventually decide to buy something (y) or to leave the shop with empty hands (z). He finally has to make a decision, but he can not avoid each of both possibilities. Of course the person could also steal something without buying it, and thus not buy anything but leave the shop with full hands. In fact, exactly in this case the constraint would signal an undesirable "program" behaviour with its invalidity.

Figure 4.1 gives a visual suggestion how the two constraint types could look like. Each state of a constraint is visualised by the image of the robot appearance to make the developer think he is directly working with the robot. Furthermore, the state labels could be replaced by the real screen displays of the regarding states. It has to be noticed that the arrows used for visualizing template constraints are not necessarily equal to transitions of the state machine. All parameters for the template constraints shall be directly customizable with the robot surface symbols. Figure 4.2 demonstrates how states could be choosen as parameters.

All template constraints are additionally equipped with textual expressions in order to explain their semantics. These textual expressions are template sentences with placeholders for the states available for parameterisation. Whenever there is a change to the constraint, te text is being updated automatically regarding to the parameter states. These additional expressions are also shown in figure 4.1.

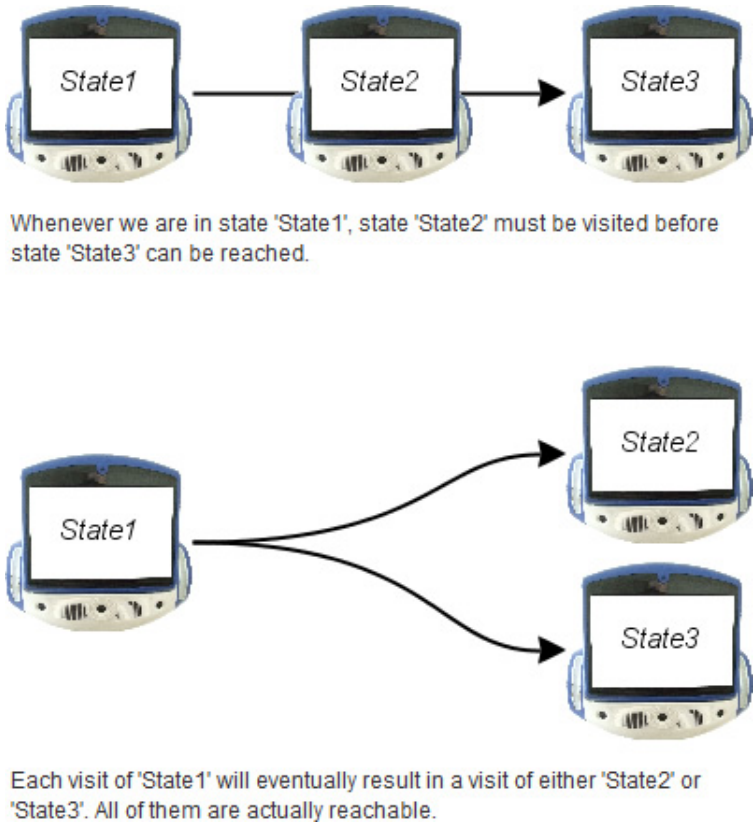

Figure 4.1: The two template constraint types, parameterized with particular states.

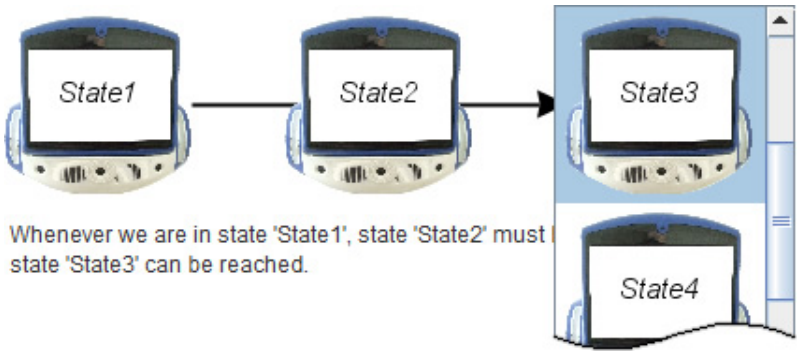

Figure 4.2: Possible way of specifying template parameter.

Each template constraint describes a certain behaviour which might be later used for checks. Thus, specific formulas have to be provided which represent the semantics of the constraints. They have to be in a specially defined format so that they can be processed by a model checker. For the first kind of template operator, a linear temporal logic fomula (LTL, see section 2.3.2) expresses its semantics (with x as "x is current state" and similarly for y and z):

$$\vDash \Box(x \rightarrow [(\neg z)\mathcal{W}y]) \tag{4.1}$$

"Globally is true: When x is the current state, z can not not be visited before y is visited". Compared to the example above it means: Whenever a person eats something (x), he can not go to bed (z) before he brushes his teeth (y). Otherwise, if he never goes to bed, he does not necessarily need to brush his teeth.

The semantics of the second operator template is more complex since it has a secondary condition that all y, z, ... states are actually reachable after a visit of x. Possibilities can not be expressed by LTL, thus, it needs to use computation tree logic (CTL) for describing this auxiliary condition in a second formula. A short explanation about CTL is already given in section 2.4.2, but the two new constructs used for this formula are described below. The main formula for the second template uses LTL:

$$\vDash \Box(x \rightarrow \Diamond(y \vee z(\vee ...))) \tag{4.2}$$

"Globally is true: When x is the current state, in a future step either y or z (or ...) will be visited". Applied to the example mentioned above it means: every time a person enters a supermarket (x) he will eventually buy something (y) or buy nothing (z). The fact that paths really exist in the program which eventually lead to x and y is expressed by the following CTL formula. AG replaces the $\Box$ sign and stands for "on all paths is globally true...", EF is related to $\Diamond$ and expresses "a path exists where is true in future...".

$$\vDash AG(x \rightarrow ((EFy) \wedge (EFz)(\wedge ...))) \tag{4.3}$$

"It is globally true on all pahts: Whenever x is the current state, there is at least one path leading to y and at least one path leading to z".

4.2 Operator constraint formalism

The operator constraint formalism is a new approach with a special focus on expressiveness and uniqueness of the visual language. It is designed as a visual formalism for linear temporal logic having a visual operator for each LTL operator. Since the visual constraints are directly mapped to LTL expressions, the visual language is equal to LTL in expressiveness. If it turns out to be usable easily though it is a mathematical concept, it can also be used by non-experts who do not already know LTL.

4.2.1 Requirements

To provide the required functionality, the fundamental logical operators are required, as shown in (a) through (e) below.

(a) *AND* (*AndOperator*): $\varphi \wedge \psi$.
This conjunction evaluates to true only if both φ and ψ are true.

(b) *OR* (*OrOperator*): $\varphi \vee \psi$.
At least one φ or ψ being true makes this construct true. The result is false when both parameters are false.

(c) *IF THEN* (*IfThenOperator*): $\varphi \rightarrow \psi$.
Instead of an *IMPLIES* operator an *IF* operator is provided. Its meaning seems to be more intuitive for non-experts; moreover it is a common construct in almost every programming language.

(d) *NOT* (*NotOperator*): $\neg\varphi$. This operator negates the value of φ.

(e) Proposition (*StateOperator*): ρ.
Until now there is only the *state proposition* type, which gives evidence about the currently active state. Other types such as numeric or string equations may be added but are less important for the healthcare subject.

In addition to these five logical operators the visual language should be furthermore capable of expressing constraints about future steps, both “any future state” (h) and

"the next state" (g), that "events should always happen" (f), and that "a property must be true until some future event" (i):

(f) *ALWAYS* (*AlwaysOperator*): $\Box\varphi$.
φ must be true now and in all following states.

(g) *NEXT* (*NextOperator*): $\bigcirc\varphi$.
φ has to be true in the next state.

(h) *FUTURE* (*FutureOperator*): $\Diamond\varphi$.
At least one time - now or in a later state - φ must be true.

(i) *UNTIL* (*UntilOperator*): $[\varphi\mathcal{U}\psi]$.
Now and in all following states φ must be true at least until there is a state with ψ being true. In addition eventually there has to be a future state with ψ being true.

All the operators mentioned above shall be suitable for constraint editing in a way that the visual language is very easy to use, especially for non-experts. This implies simple editing as well as easy readability of constraints. Furthermore it should provide an intuitive application which is ideally learnable within little time.

4.2.2 Design decisions

Like the formula representation introduced by Del Bimbo et al. [3], a constraint consists of nested operators and propositions in a hierarchy. However, the new visual formalism surrenders the three dimensional idea and focuses on two dimensional graphical blocks and their compositions instead, which is used by HomeTL [4] as well. Each operator of the visual language is represented by such a block.

The visual language has support for eight unary or binary operator types and one proposition which have their specific semantics. In order to support the reading of constraints, different significant visual appearances of the operator types shall enable easy classification and a faster subconscious identification [19]. For this purpose

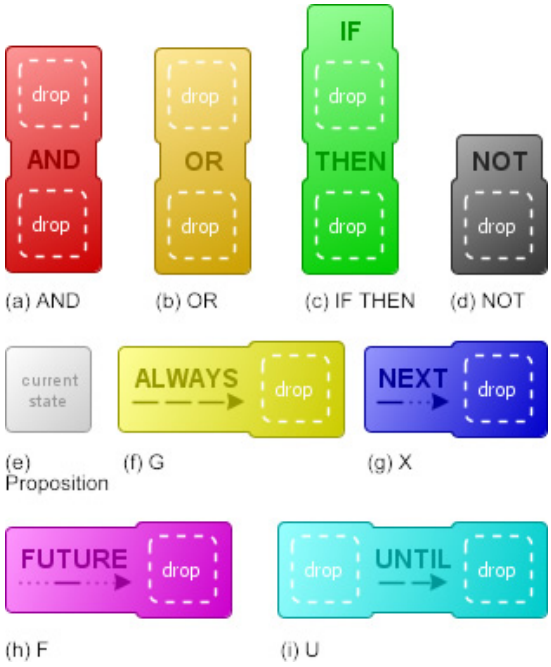

Figure 4.3: Visual representation of all supported operators.

an individual color is used for each operatot type. A suggestion for the operators' graphical representation is shown in figure 4.3.

A second effort for making the perceptibility of the visual language as intuitive as possible is done with the orientation of constraints. Operators which consist of nested operators have particular meanings for the two dimensions: The "logical" vertical read direction and the horizontal direction for variation in time. Thus, logical operators are arranged in a vertical row whereas time relevant operators are aligned horizontally. Figure 4.4 demonstrates an example constraint composed along the logical and time axes. It is read from left to right and top to bottom: "ALWAYS is true: IF state *'Not taken yet'* is active THEN state *'Polling'* can NOT be visited UNTIL state *'Staff notified'* OR state *'Well done!'* is reached."

The two dimensions form a special requirement also for the layouting of nested operators. Each operator adapts its size so that it surrounds all recursively nested operators. But also the positioning of nested operators within teir parents have to be taken into account. Whereas the operator's centered positioning and resizing along

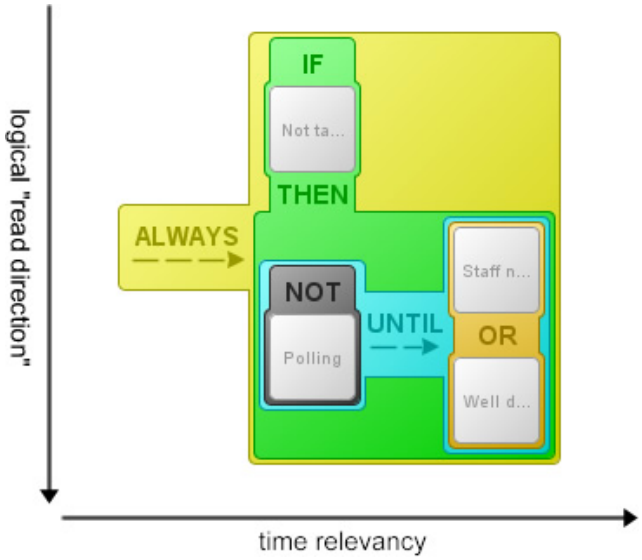

Figure 4.4: Easy understanding of visual constraints due to intuitive read directions.

the logical direction only depends on the sizes of all suboperators, the arrangement within the time axis turns out to be more complex. Here, suboperators can not just be aligned centered as it is the case vertically. To face the determination of horizontal positioning, every operator gets time reference lines which specify its positioning within its parent and the positioning of its own children within itself. As depicted in figure 4.5, logical operators - i.e. *IfThenOperator*, *AndOperator*, *OrOperator*, *NotOperator* and *StateOperator* - have one reference line. Time relevant operators - i.e. *NextOperator*, *FutureOperator*, *AlwaysOperator* and *UntilOperator* - have two such reference lines, one for their own positioning within their parents and one for the positioning of their children.

As postulated in the requirements for the visual language, constraints shall be easy to read. In fact, there is a possibility to improve the presentation of certain visual constraints. Multiple nested operators of the same kind which have no order priorities such as *OR* or *AND* appear graphically unnecessarily complex. These operators can be visually merged in the manner of unifying their border lines, for example. As a result, all associated operators appear as just one operator with a couple of suboperators (see figure 4.6).

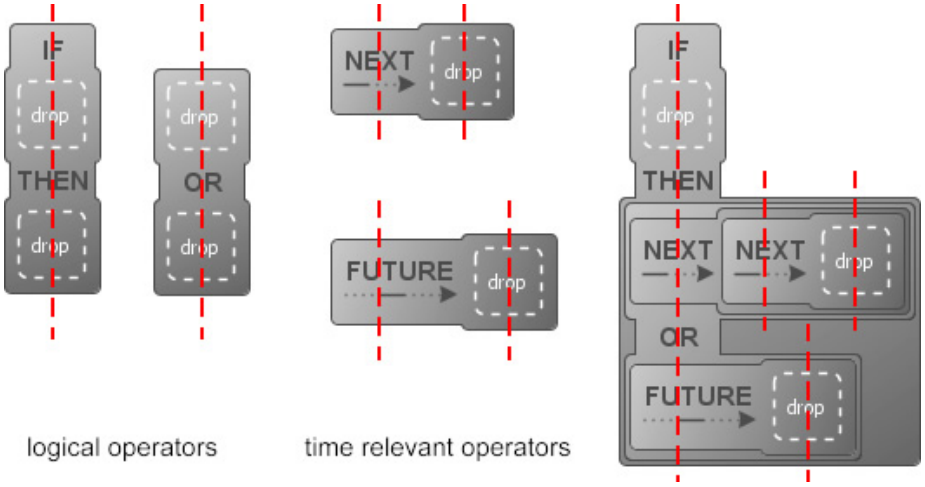

Figure 4.5: Time reference lines and their adjustments.

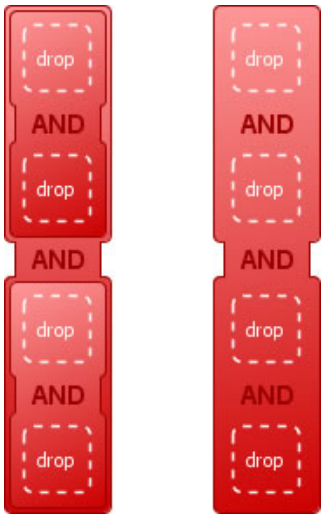

Figure 4.6: Default nested *AND* operator and visually merged nested *AND* operator.

4.2.3 Guidelines for an editor

Also an editor for this visual language which holds the functionality for composing operators can contribute to usability and clarity. Some requirements are as follows:

- All editing is performed only by simple drag and drop. Operators and propositions can be moved and dropped onto other operators. A tool bar provides all operator and proposition types to be instantiated and a possibility for deleting existing operators.
- In order to simplify the reading of constraints an operator can be hovered over by the mouse pointer. It is intended to be similar to line coloring when reading a digital text.
- Every individual edit step to either the constraint or the underlying state machine results in an automatically (re-)validation of the constraint. The result is immediately displayed to the user.
- There is no button for constraint creation respectively editing except operator and proposition providers, which let the developer create new instances. Having no additional buttons reduces complexity and keeps it simple.

4.3 Comparison

Due to its arrow based design the visual language for template constraints is a good visualization of time dependencies. Furthermore, it allows expressing possibilities by the use of CTL formulas which is not possible with the operator costraint formalism. Nevertheless, the template visualizations have some trouble with clearness of their semantics. Even though there are well defined formulas behind each constraint and textual expressions which explain the meanings of constraints, the visual presentation seems to be ambiguous in some settings which leads to the risk of being interpreted differently. For example, in the context of the visual language it is unclear whether the first constraint template allows state (x) to be visited more than once before state

(z) is reached. It might also be problematic to interpret in the second constraint whether more than one of the states y, z, ... can actually be visited after a visit of x. In contrast, the operator constraint formalism is not ambiguous due to to its direct mapping to LTL which is consistently defined. All visual operator expressions represent exactly the same semantics as their corresponding textual LTL formulas.

Additionally, there is a second problem with the template constraints. Whereas the safety constraints specific to the example within the healthcare domain mentioned in section 2.1 are expressable by this visual language, there are other common constraints which have no matching template and thus can not be described. For example, checks of states like "being visited never" or "being visited at least once" are not possible. For every new constraint type a new template has to be created which means an infinite amount of templates is needed in order to achieve full expressiveness. On the contrary, the visual language for operator constraints is not limited in expressiveness compared to LTL, disregarding that there is only one proposition type available. But this is an extension issue of the prototype rather than a lack in expressiveness.

Since a visual language has to be developed which suits non-professionals as well as experts, the risk of ambiguity is undesirable. Especially for a tool for safety and consistency it would form a really bad basis. For this reason, the approach of template constraints is discarded and the operator constraint formalism is used for the editor prototype instead. Owing to the recursive nesting of operator constraints, a visual formula can be recursively translated to a LTL sentence which can be processed by the prototype using any ordinary model checker.

5 Automated generation of safety constraints

Safety constraints primarily constitute essential properties which describe correct and safe behaviour of safety critical software applications. In order to discover whether a program meets certain safety requirements, its behaviour is checked by the constraints. This happens mostly as a last check before software gets released. But regarding updating and maintenace work, often further changes are made to the software even after release. However, safety constraints can be applied again and assure the adherence of all properties. Safety constraints thus are also suitable for sustaining consistency during the different phases of software life. In fact, not only safety constraints, also properties of program behaviour which are not safety relevant at all are applicable for that. The more constraints are defined the better can be the reliability. But as already mentioned before, it might be difficult for developers to find all reasonable constraints by hand. In this case it can happen that a prior correct software version becomes faulty over time because of minor adjustments once in a while which are not covered by constraints.

As a solution to this problem, the concept of automated constraint generation for state machines is proposed. Once the initial implementation of a program is finished, the behaviour of this state machine can be analyzed. In particular, heuristics can be elaborated which determine significant parts of the state machine automatically and put them into constraint representation. These constraints are of course valid on the current program. On the one hand, the constraint suggestions may help the developer identifying reasonable constraints he did not think about as well as contradictions between program and specification, eg. if a computed constraint does

not make sense at all. On the other hand, all constraints can be revalidated after each change to the software and thus ensure consistency.

In order to support the developer in developing safe programs and to simplify maintainability, automated generation and suggestion of constraints for state machines might be helpful. But it is helpful only if the suggested constraints are relevant and reasonable.

5.1 Subgraph definition

Whether suggested constraints are relevant and reasonable depends on the domain and use case. It has to be mentioned that the approach of finding an algorithm for constraint generation presented in this work has a special focus on the healtcare use cases and is not necessarily applicable for other domains. Among others, each safety constraint postulated in chapter 3 was analyzed on a state machine program which represents the medication reminder application of the health care project. It turned out that all states which are named in the constraint are always connected in an identical manner and seem to fall into one of the following categories:

1. States having two or more outgoing transitions (α-states).
2. States in which all paths starting from one α-state come together again (β-states).
3. All States immediately before β states (γ-states).

In order to compute the required constraints automatically, and possibly additional ones, for each α-state all associated γ-states and the one β-state have to be determined. Such a group of states of a state machine is named subgraph from now on. It is defined as follows:

- There is a start state (α-state) with at least two outgoing transitions.
- There is one end state (β-state) which is the first state where all possible paths of the state machine starting from start state come together again.

- All states between the start and end state also belong to the subgraph. They have no other incoming transitions than the ones originally coming from the start state. That means there is no path to visit these states except through the start state.

- The start state may have incoming transitions from states not contained in the subgraph.

- The end state may have self transitions or incoming and outgoing transitions from and to states which are not contained in the subgraph. Transitions back into the subgraph are not allowed except to the start state.

- Except self transitions of the end state, no cycles are allowed in the subgraph.

A graphical example of such a subgraph is shown in Fig. 5.1. Using this specification all possible subgraphs of this kind can be detected within a state machine. An algorithm how subgraphs can be computed programmatically is discussed later in section 6.3.

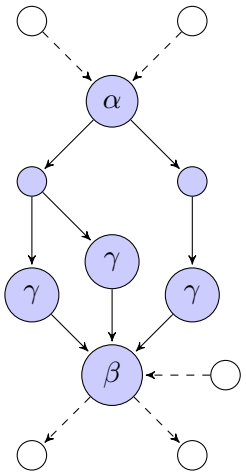

Figure 5.1: Example for a subgraph.

5.2 Constraint determination

For the healthcare applications, all important states which concern safety constraints belong to subgraphs. Thus, several safety properties can be formulated based on the structure of subgraphs. For each subgraph the following valid LTL formulas can be derived if it contains neither inner (infinite) loops nor final states (states having no outgoing transition). In the following α is proposition for "α is current state" and similarly β and γ:

$$\vDash \Box(\alpha \rightarrow \Diamond\beta) \tag{5.1}$$

$$\vDash \Box(\alpha \rightarrow \Diamond(\bigvee_{\gamma} \gamma)) \tag{5.2}$$

$$\vDash \Box(\alpha \rightarrow [\neg\beta \mathcal{U} \bigvee_{\gamma} \gamma]) \tag{5.3}$$

Additionally, for all states not contained in the subgraph (let's call them δ-states, and the proposition δ stands for "δ is current state") the following formula is true:

$$\vDash \Box(\alpha \rightarrow [(\bigwedge_{\delta} \neg\delta)\mathcal{U}\beta]) \tag{5.4}$$

In case of $\alpha = \beta$ - i.e. when all paths starting from α-state eventually return again (cycle) - even another constraint can be assumed to be true:

$$\vDash \Box \Diamond \alpha \tag{5.5}$$

These five constraints apply to each subgraph matching the conditions mentioned above. Formula 5.1 says that an α-state is always eventually followed by a β-state. Formula 5.2 ensures that there will always at least one γ-state be eventually visited after each visit of an α-state. The quite similar but stricter formula 5.3 additionally demands a visit of at least one γ-state before the β-state can be reached. Formula 5.4 simply says that a state not contained in the subgraph can not be visited between α- and β-states. The constraint that the α-state will always eventually be visited again

is expressed by formula 5.5.

All formulas can be translated to the visual formalism presented in section 4.2 and shown to the user. The appropriate LTL formulas for the constraints postulated in chapter 3 relate to number 5.2 and 5.5 of the subgraph based formula templates and are as follows:

$$\vDash \Box \Diamond \text{'Polling'} \tag{5.6}$$

$$\vDash \Box(\text{'Not taken yet'} \rightarrow \Diamond(\text{'Well done!'} \vee \text{'Staff notified'}) \tag{5.7}$$

Formula 5.6 ensures that state “Polling” will always eventually be visited again, and formula 5.7 specifies that either state “Well done!” or state “Staff notified” will be eventuelly visited after each visit of “Not taken yet”.

6 The LTLCreator prototype

As stated in chapter 3, appropriate support by a graphical editor is important for the usefulness of the developed visual language. Therefore a prototype of such an editor has been designed and implemented which contains the visual language as well as the proposed constraint generation heuristics. The implementation is a Java Swing component and has a simple API definition, making it easy to integrate the *LTLCreator* tool into other java applications.

First of all the model implementation for representing state machines is described in section 6.1. Then, some details about the implementation of the visual language are given in section 6.2, directly followed by an explanation about the realization of the automated constraint generation functionality in section 6.3. The assembly of all the previous features in one tool is described in section 6.4. However, the tool's functionality shall also be available in Robostudio in order to provide safety mechanisms for the healthcare service robots and to be a prototype for evaluation. Section 6.5 gives a main idea how an integration into Robostudio is possible.

6.1 State machine model

For the checking of constraints as well as for the constraint generation a model is needed which represents the state machine program. Three classes are responsible for holding the state machine program with all its necessary information: The class *Fsm* represents the finite state machine and lists all contained states. It is called finite state machine, because the number of states is finite which applies to the domain of this work. *State* represents the states of the machine and holds a list of all transitions from or to itself. Transitions are represented by the class *Transition* which specifies

the source and a target state. Figure 6.1 shows all three class definitions for the state machine model.

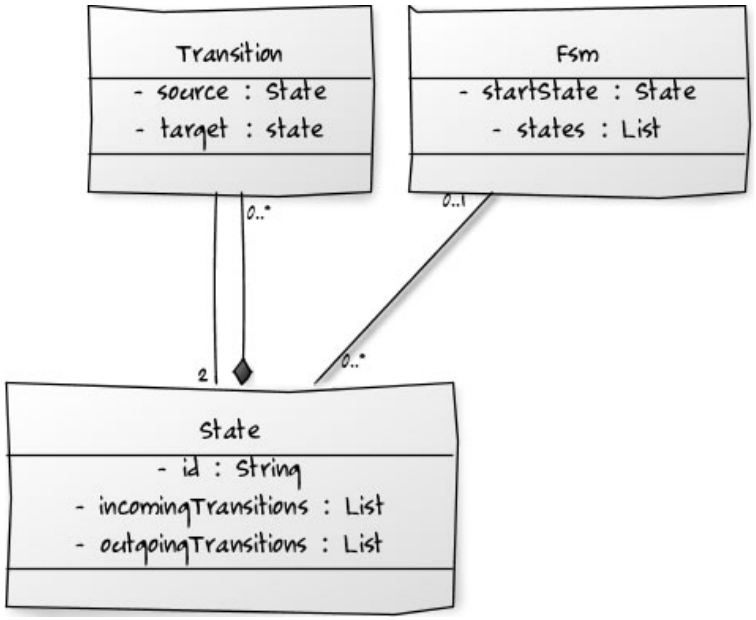

Figure 6.1: Class definitons for the state machine model.

6.2 Operator constraints

All operator types, i.e. *IfThenOperator*, *AndOperator*, *OrOperator*, *NotOperator*, *NextOperator*, *FutureOperator*, *AlwaysOperator*, *UntilOperator* and *StateOperator*, are realized by dedicated classes. All of them inherit from one abstract class *AbstractOperator* which provides the basic functionality beeing used by all operators. The *AbstractOperator* class is shown in figure 6.2 and its methods are described as follows:

getLTL(): This function returns the LTL formula of an operator and all its children recursively. Applied to the most outer operator it returns the complete formula for the constraint. This function is abstract and implemented by each operator type separately.

getColor(): Each operator type has got is own significant color which can be retrieved by the abstract function *getColor()*.

createNewInstance(): In order to provide operator creation over a tool bar, this function instantiates a new operator object, depending on the implementation.

isSimilar(AbstractOperator): Two operators can be compared for semantic equality. Equality is given if the compared operators are of the same type and similarly their children. This functionality is important for avoiding duplicate constraints during constraint generation, for example.

addChangeListener(OperatorChangeListener): Whenever there is a change to the operator or to one of its children, an operator change listener gets notified about it. The hierarchical architecture of *AbstractOperator*s propagates all change events to the root operator. The function *addChangeListener()* allows registering for such change notifications.

removeChangeListener(OperatorChangeListener): According to *addChangeListener* this function allows deregistration again.

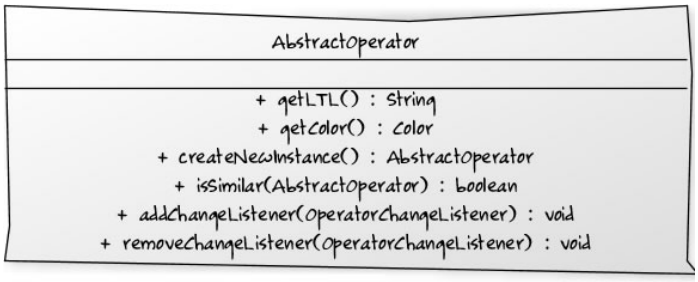

Figure 6.2: Class definition for *AbstractOperator*.

Whereas *StateOperator*s have no operator children and are leafs in the syntax tree of LTL formulas, all other operator types are either unary or binary operators with one or two optional nested children respectively. The graphical representation of both unary and binary operators thus needs to have place holders and docking stations

for sub operators. These are realized by buckets which display a small "drop" label as shown in figure 6.3 and allow operator adding or removing by mouse drops. For this a class *Bucket* is defined with get and set methods for defining a sub operator. Figure 6.4 depicts the relationship between *AbstractOperator*, *Bucket* and operator implementations. Once operators are composed in a way that no empty buckets are left over, a complete constraint such as the one shown in figure 6.5 is retrieved. This diagram illustrates the correlation of nested operators and buckets.

Figure 6.3: Buckets for graphical nesting of operators.

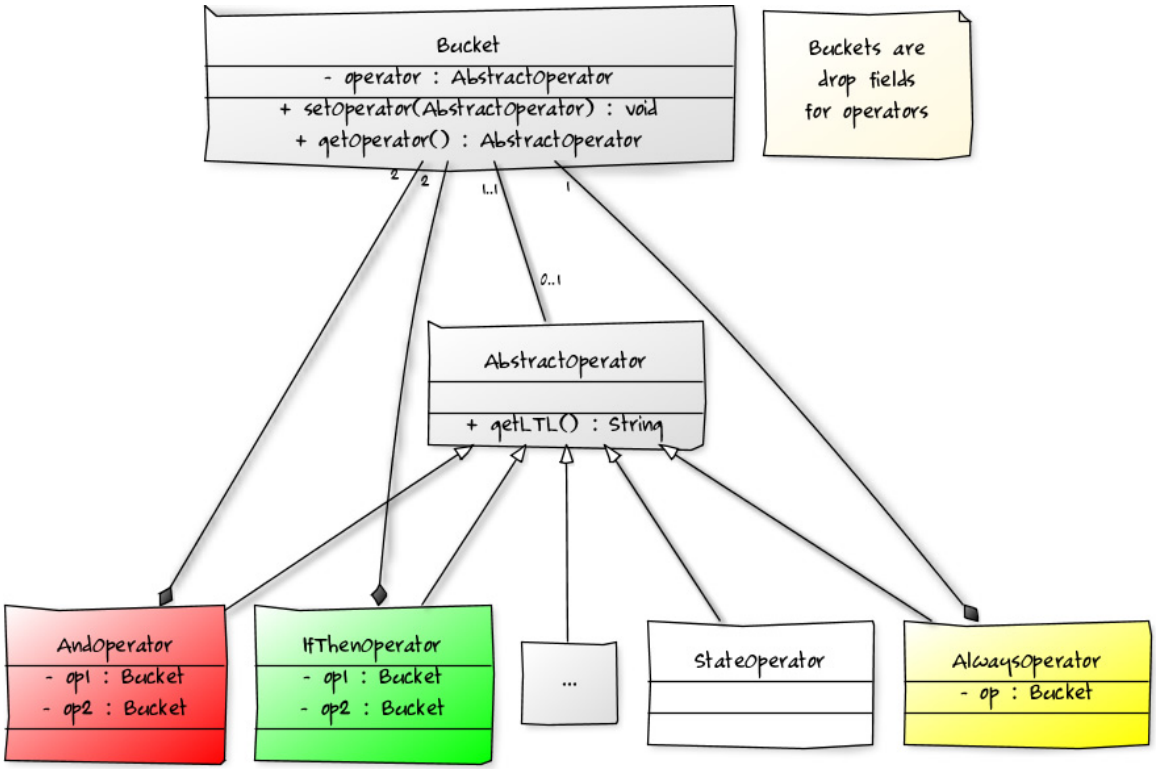

Figure 6.4: Class diagram of operator structure.

For the prototype implementation, all graphical components such as buckets and operators are implemented as Java Swing *JComponent*. Thereby a lot of functionality

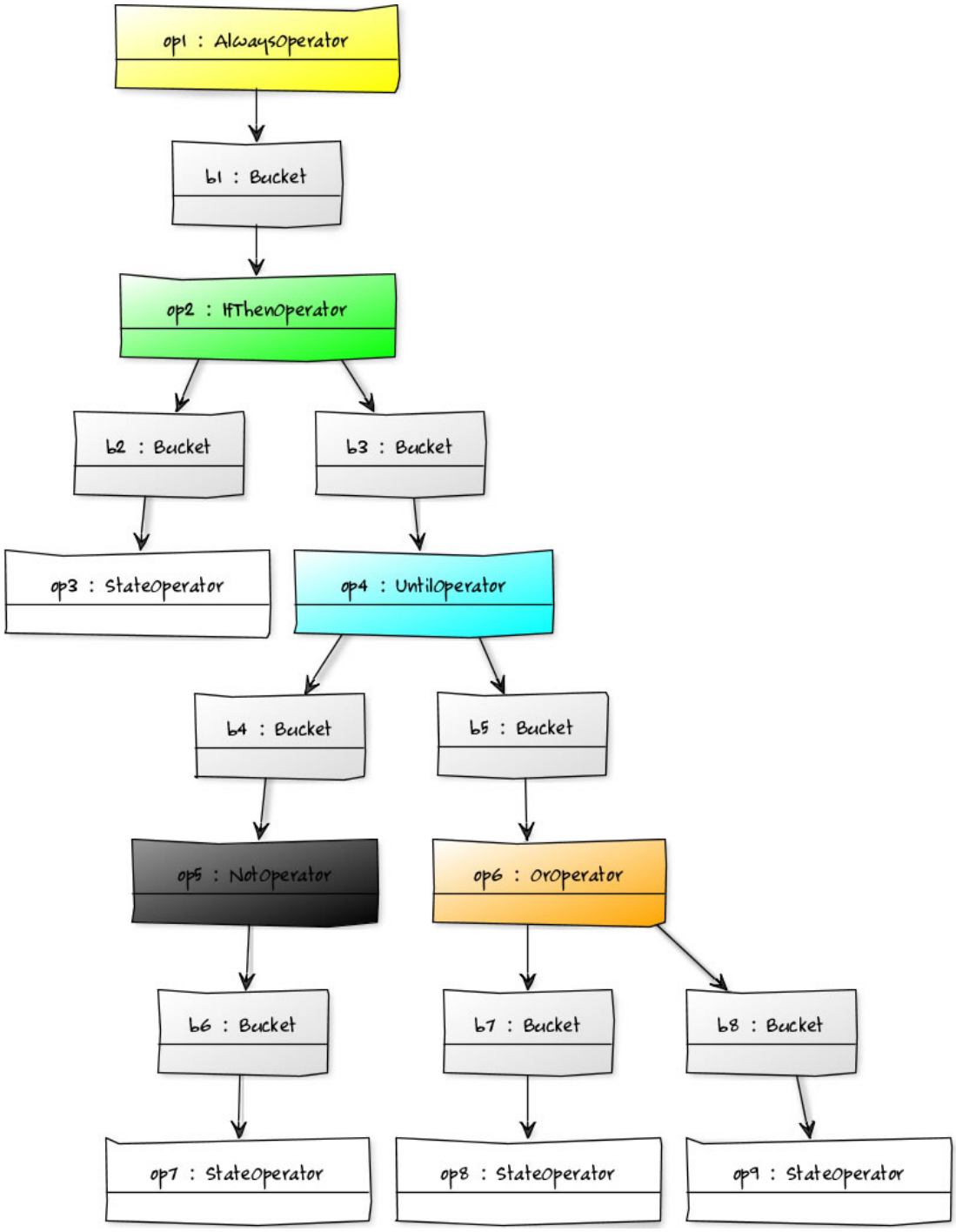

Figure 6.5: Instantiation of a constraint with several operators.

for the visual configuration is already available such as the complex layout and repaint concept and the container functionality which is used for unary and binary operators as well as buckets.

Swing allows the implementation of a *paint()* method which is responsible for rendering the component. This is used for giving every operator its special look. Swing's *Graphics2D* functionality provides a great support in painting rounded shapes and color gradients. An operator is displayed as a rounded rectangle with dents around sub operators; it is displayed with a flowing backgound and borders and labels in the operator type's specific color. For the significant shape multiple rounded rectangles with and without borders are arranged in a way that the result appears to be one piece. Figure 6.6 demonstrates how the process of painting works: For painting the *IfThenOperator*, first of all a blank rectangle with border is drawn, directly followed by two filled rectangles with border which are placed where the two child operators shall be later on. In the third step the same rectangle as from step one is painted on top, this time without border. This makes all crossing borders disappear. Only labels and buckets still have to be added to create a complete *IfThenOperator*.

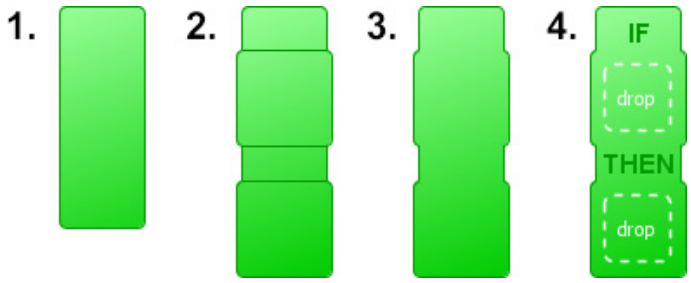

Figure 6.6: Four steps of painting an operator.

As proposed in the design decisions for the visual language in section 4.2.2, constraints have particular meanings for the two dimensions: A vertical read direction and the horizontal direction for variation in time. Swing's mechanism for layouting components is used for the positioning of operators. After determining the aspired sizes of child components which represent sub operators, reference lines which have been already demonstrated with figure 4.5 in section 4.2.2 are computed mathematically. Based on this information operators are sized and positioned correctly.

Regarding to the requirement for simple reading of constraints based on mouse hovering, the editor prototype supports context sensitive highlighting of particular constraint parts. The built-in mouse support of Swing is used for identifying the operator positioned under the mouse cursor. All operators except the hovered one and its sub operators are presented bleached out which causes the hovered operator to appear emphasized. The most right operator composition in figure 6.7 gives a demonstration of this idea.

However, the concept of operator highlighting causes trouble in combination with the graphical merging of visual operators which was introduced in section 4.2.2. Since a merged operator does not have its own border, it can not be displayed as desired when hovered. This fact is also a problem for the editing: The graphical presentation does not support changing of merged operators, because it seems that they can not be picked by the mouse. In order to retain full editability, hovered operators are excluded from the merge rule and still have to be displayed separated (see figure 6.7).

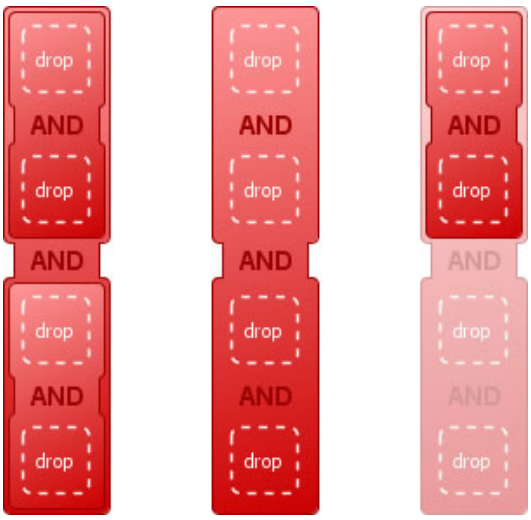

Figure 6.7: Default nested *AND* operator, visually merged nested *AND* operator and visually nested *AND* operator with mouse over (from left to right).

Besides its importance for mouse hovering, Swing's mouse support is also used for determining the position where operators have to be added after drag and drop

edit actions. So far a simple drag and drop mechanism called *DndHandler* has been implemented whose class structure is shown in figure 6.8. Once a *JComponent* is

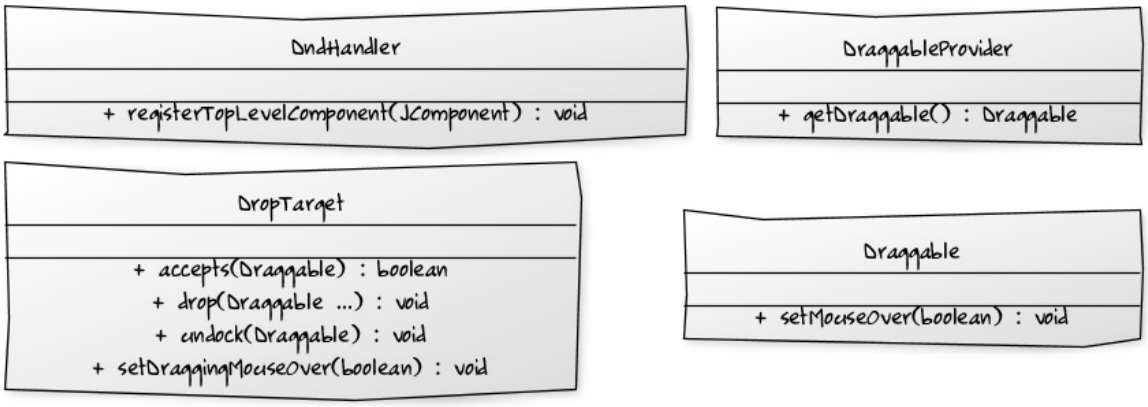

Figure 6.8: Class definition for drag and drop functionality.

registered to the *DndHandler*, it automatically manages any drag and drop actions with *JComponent*s which implement either the *Draggable* or *DropTarget* interface. The former is implemented by all operators and specifies that this object can be moved from and to buckets or the dashboard, the latter is implemented by buckets and the dashboard and determines that it can accept and contain *Draggable*s such as operators. Furthermore there is a class *DraggableProvider* which is similar to *Draggable* except that it is not moved itself but creates a new instance of the respective *Draggable* implementation type which is then moved instead.

LTLCreator is equipped with a tool bar which provides all functionalities needed for constraint editing: For each operator type a *DraggableProvider* is displayed as an icon in the respective operator color. Each time the mouse is pressed on it (drag action start), a new instance of the respective operator type is created which can be moved and dropped to the dashboard. Additionally, a trash can icon is displayed within the tool bar. It implements the *DropTarget* interface and can thus receive operators. Its purpose is to provide a remove functionality for already instantiated operators which are not needed anymore.

6.3 Automated constraint generation

As stated in section 5.1, the automated generation of constraints for a state machine is based on subgraphs. For the finding of such subgraphs an algorithm has been developed. Section 6.3.1 aims to explain the basic concept of the algorithm, section 6.3.2 introduces the slightly modified algorithm which is used for the prototype.

6.3.1 Subgraph finding concept

First of all a start state (α) is defined which has two or more outgoing transitions. All possible paths are generated which start from it and lead through the state machine. Every path ends before it visits states twice. The result is a finite set of finite paths since loops are eliminated. Now every state gets as label assigned a number which represents the total number of occurrences of this state in all collected paths. In other words, the label of a state expresses the number of paths which start in the α-state and contain this state. The α-state itself is labeled with the total number of paths. If during walking through any of the paths a second state can be found with the same label, it means that all paths starting at α-state come all together in this state again. It is determined to be a potential corresponding end state (β). A last check has to be applied whether there are incoming transitions from outside into the subgraph which do not originally come from α-state. If no such transitions are detected, a subgraph is found. If there is no state with the same label, or a transition exists which comes from outside, the state machine as a whole can be assumed as subgraph beginning with α-state insofar as the state machine does not contain further loops or terminating end states. Figure 6.9 illustrates this concept of subgraph finding and shows all possible paths as well as the number of occurrences of each state. This procedure is suitable for checking whether a state is the beginning of a subgraph and for discovering the corresponding β-state. In order to detect all subgraphs within a state machine, this procedure has to be applied to every single state which has two or more outgoing transitions. In order to improve the further finding of subgraphs and walking through the state machine, all already found subgraphs can be assumed as single states since it is already known where their paths come finally together again.

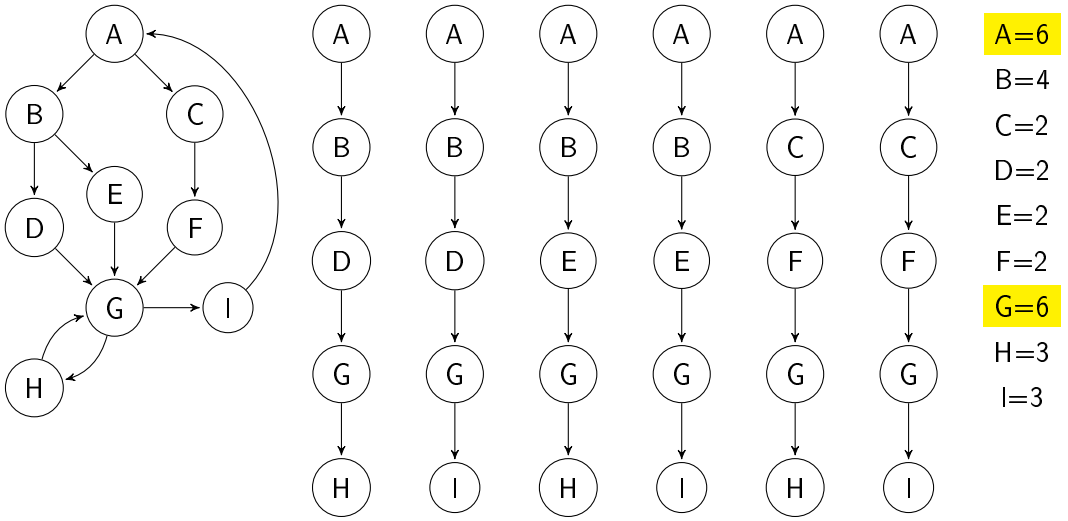

Figure 6.9: Example for the subgraph finding algorithm: The occurrence labels of the states lead to G as an end state (β) corresponding to A.

6.3.2 Subgraph finding implementation

Following the previous description an algorithm has been implemented in Java. Its realization differs from the presented concept: Paths are not listed but states are directly labelled since not saving all possible paths is more memory efficient. The state machine is traversed similarly to a depth-first search and each visited state's label is incremented by one. If the current state holds more than one outgoing transition and thus constitutes new paths, all states on the current stack, i.e. the previously visited states on the current path, get their labels incremented by ('number of outgoing transitions' - 1). The algorithm turns back whenever it comes across a state which has already been visited within the same path stack of the depth-first search.

The overall implemented way of subgraph finding is illustrated in pseudo code with algorithm 1. Line 1 defines a variable *subgraphs* which will contain all results of the algorithm after execution. In line 4 the labelling process described above is executed by using one particular start state. Line 5 checks whether there is a state having the same label as the start state. If so, it is assumed as potential end state (line 6). Otherwise, the entire state machine is assumed to be a subgraph and the start state

Algorithm 1 Finding of all subgraphs.

```
1:  subgraphs: List
2:  for all states as S do
3:     start ← S
4:     doLabelling(start);
5:     if state E exists with label(E)=label(S) then
6:        end ← E
7:     else
8:        end ← start
9:     end if
10:    if noCycles(start, end) and noExtTransitions(start, end) then
11:       subgraphs.add(start, end);
12:    end if
13: end for
```

is defined as end state as well (line 8). Subsequent checks in line 9 finally determine whether it can be considered as proper subgraph. There are no cycles allowed between start and end state since they are forbidden by the subgraph definition in section 5.1. Furthermore also foreign transitions are prohibited: Transitions coming from a state outside the subgraph are allowed to lead to the start state or the end state but not to other states of the subgraph. If all checks are successful, start and end states get added to the result list *subgraphs* in line 11. This procedure is applied to all states of the state machine by the surrounding foreach loop (line 2). In the end, the list *subgraphs* contains all pairs of start and end states which form subgraphs.

6.3.3 Constraint generation

After determining a subgraph, constraints are derived from it as defined in section 5.2. The entire process of constraint generation is illustrated by figure 6.10. Once the user triggers the constraint generation for a state machine (stm), the subgraph finding algorithm starts working. When a subgraph is found, constraints are created and passed to the user. For each subgraph multiple constraint types can apply. This procedure is done iterative for each subgraph found.

The runtime of the algorithm depends on size and grade of branching of the state

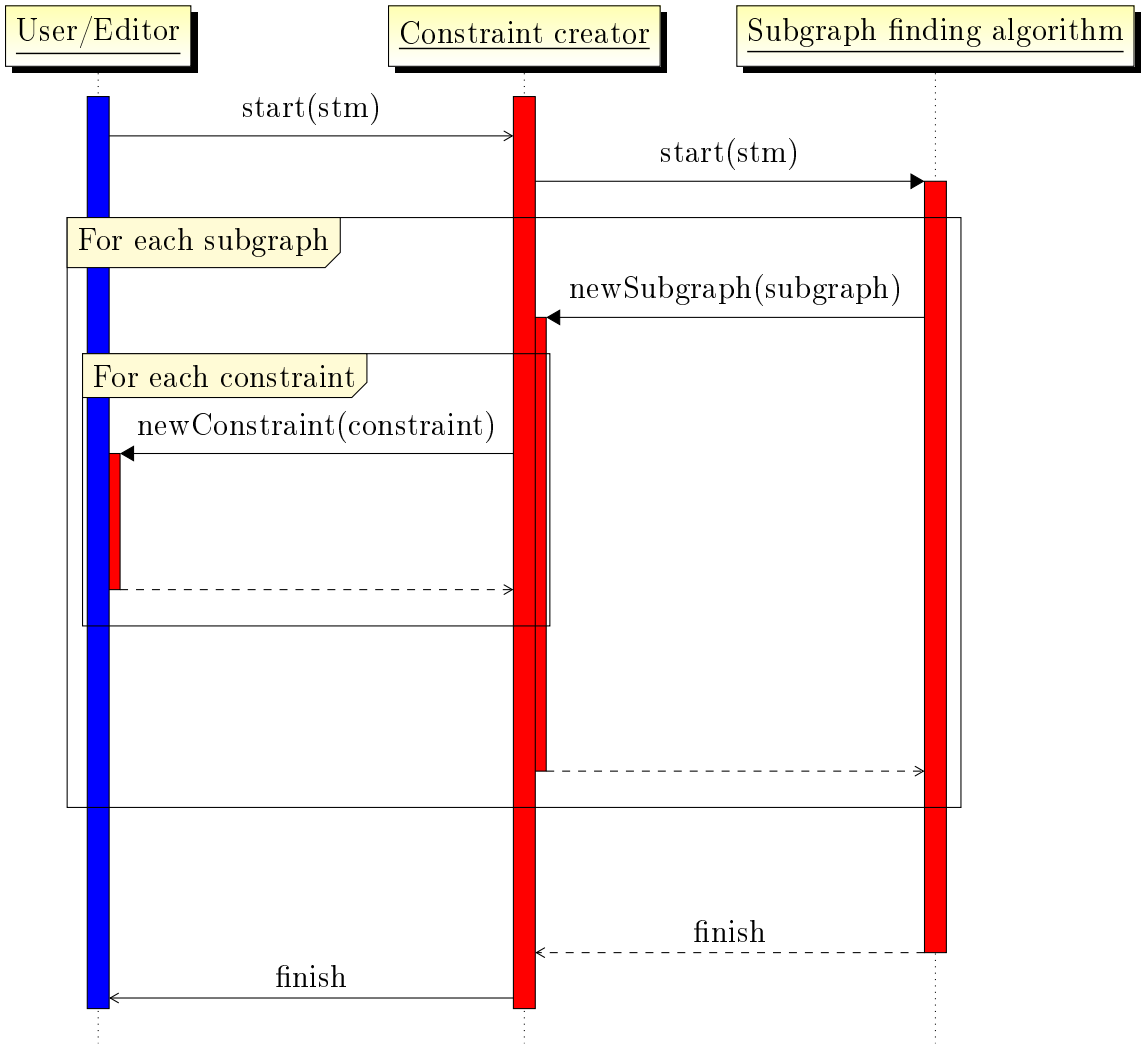

Figure 6.10: Process of constraint generation.

machine. An estimation about the execution time is presented later in section 7.2.1. In order to improve waiting times and thus user experience during constraint generation the entire result might not be returned after termination but each subgraph should be processed as soon as it is found. For this, the implementation for the algorithm has been extended by code snippets which frequently interact with so called *ResultListener*s (figure 6.11). An implementation of *ResultListener* has to provide three methods which allow the algorithm to publish the current progress by calling *void setProgress(int)* or to notify about newly found subgraphs by invoking *void newSubGraphFound(SubGraph)*. The method *boolean continueGeneration()* is called frequently and the execution will terminate as soon as it returns false.

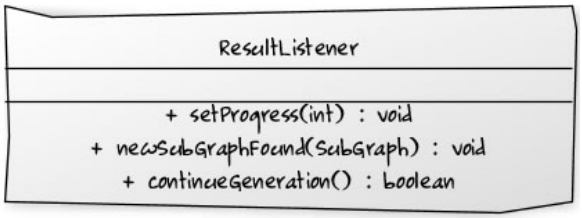

Figure 6.11: Interface definition of *ResultListener*.

As a second approach aiming for better user experience during constraint generation, a dialog has been implemented which informs the developer continuously about the current progress. This includes the number of constraints found so far and a progress bar which gives a percentual estimation of execution time. Furthermore a cancel button permits premature stopping of the execution. A snapshot of the Dialog is shown in figure 6.12.

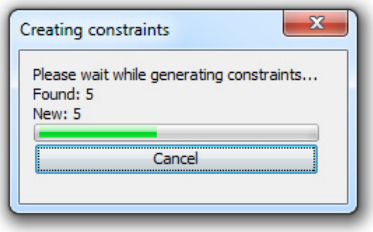

Figure 6.12: Snapshot of the progress dialog displayed during constraint generation.

6.4 The LTLCreator

Both features, the editor for the visual language presented in section 4.2 and the automated constraint generation in chapter 5, were assembled in one tool, the *LTLCreator*. Figure 6.13 shows the assembled parts of *LTLCreator*: A dashboard is provided where constraints can be edited visually by the developer and result displays indicate the validation results of constraints graphically. For validating the constraints an external model checker is linked to *LTLCreator*. This component does not have an UI as well as the constraint generator, which is also part of *LTLCreator*. The *LTLCreator* itself

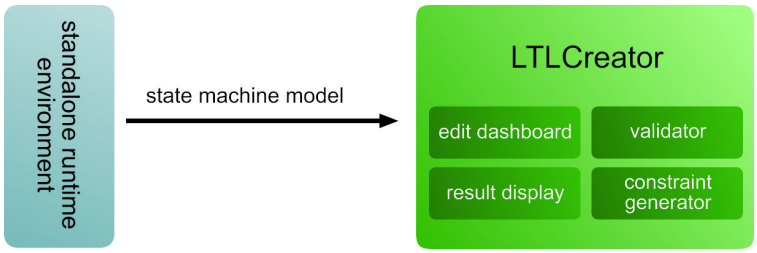

Figure 6.13: Integration of LTLCreator.

does not provide functionalities for editing state machines although a state machine model is needed for the visual language as well as for constraint generation. Therefore the standalone version of *LTLCreator* uses a hard coded state machine as model which represents an example program for test purpose.

Figure 6.14 shows a snapshot of the editor's environment: All functionalities needed for constraint editing are provided in the tool bar on the right (1). It contains draggable elements for creating all operator and proposition types as well as a trash can for deleting. Constraints can be composed by drag&drop in the dashboard in the center of the editor (2) which holds and displays the operator constraints. The tab functionality on the left allows multiple constraints to be managed (3). Each tab shows a small thumbnail of the constraint and a symbol indicating its current status of validation result: For syntactically invalid constraints, an "incomplete" sign is displayed. Otherwise, an animated ring indicates that validation is in progress and will finally result in either a "valid" or "invalid" sign as shown in figure 6.15. The

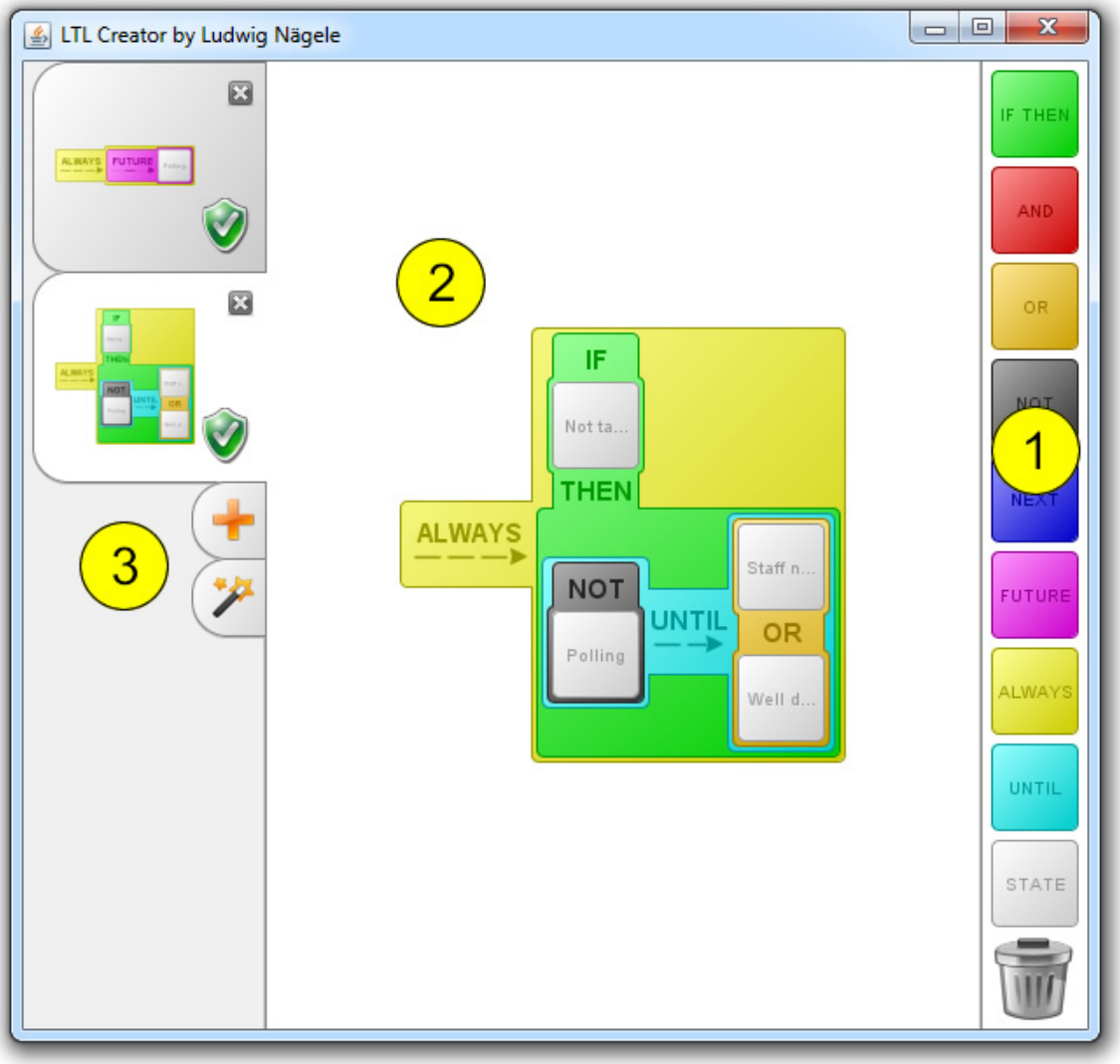

Figure 6.14: Snapshot of the visual editor LTLCreator.

Figure 6.15: Status indicators displaying validation results: incomplete, validation in progress, invalid and valid.

magic wand button within the tab pane makes the constraint generation functionality accessible within the editor and triggers the constraint finding process.

6.4.1 Model checker

A main requirement to the *LTLCreator* tool is that all constraints can be validated immediately and allow fast feedback about the correctness of programs. For this a *ConstraintValidator* class has been defined which provides a method *validate(AbstractOperator)* for proving constraints. For the constraint validation itself a particular model checker can be specified. As a default, the symbolic model checker NuSMV [20, 21] is used. It is an open source project and has been designed to be an open architecture for model checking. NuSMV does not have a Java API but it can be externally accessed over command line. For this both the state machine program and the LTL formulas have to be translated to an input file which has to match a NuSMV specific format. For LTL formulas NuSMV uses the char "G" (for globally) instead of $\Box$ and the char "F" (for future) instead of $\Diamond$. An example of this input language is given in listing 6.1.

A state machine is encoded in the variable *state*, and its value represents the active state. Thus, each state is identified by a number. The state machine described in the example of listing 6.1 contains eleven states, this is determined by the number range for *state* from one to eleven. All variable assignments are specified under *ASSIGN* and represent the semantic equivalent to transition definitions. The initial state can be determined and a case distinction regulates the next possible values, i.e. transitions to reachable states. If there are more than one outgoing transition, all target state identifiers have to be surrounded by brackets. The last expression *TRUE : state;* is a self transition for every state not considered in the case distinction. The last line contains the constraint formula which has to be checked on the state machine. The prefix *LTLSPEC* just says that NuSMV has to interprete the following constraint as LTL formula.

This kind of output is generated by the *ConstraintValidator* whenever a constraint needs validation. All states are then mapped to numbers and transitions get transformed into case distinctions. The result is stored to a file and then loaded by NuSMV.

```
MODULE main
  VAR
    state: 1..11;
  ASSIGN
    init(state) := 2;
    next(state) := case state = 2 : {3, 1};
                        state = 3 : 1;
                        state = 1 : {2, 4};
                        state = 4 : {1, 5};
                        state = 5 : {6, 9, 10};
                        state = 6 : {7, 8};
                        state = 7 : 8;
                        state = 8 : 1;
                        state = 9 : 11;
                        state = 10 : {9, 11};
                        state = 11 : 1;
                        TRUE : state;
                   esac;

LTLSPEC G (state = 5 -> (F (state = 8 | state = 11)))
```

Listing 6.1: Example of a NuSMV diagram file.

The corresponding valdidation result can be subsequently read from the command line and displayed in the editor next to the visual constraint. Due to issues of this command line API, each constraint is checked in a new NuSMV instance.

Model checking can be a quite ressource inefficient activity and due to high temporary data generation for each check, a parallel execution of multiple checks might overload the main memory which causes slow data swapping to start. This phenomen causes the overall execution to significantly slow down even on modern multicore CPUs since every process or thread has to wait for the memory. To avoid this it might be recommended to execute all validation tasks consecutively rather than in parallel. However, it can happen that there is more than one validation task at a time. This is the case when the constraint generator creates multiple constraints in succession or a change of the state machine causes all constraints to be revalidated at once. In order to avoid the concurrent execution of multiple NuSMV instances, the *ValidatorThread*, a queue based mechanism for scheduling validations, manages a sequential processing of all validation tasks. Its class definition is presented in figure 6.16. Every validation task of a constraint gets enqueued in the *ValidatorThread*

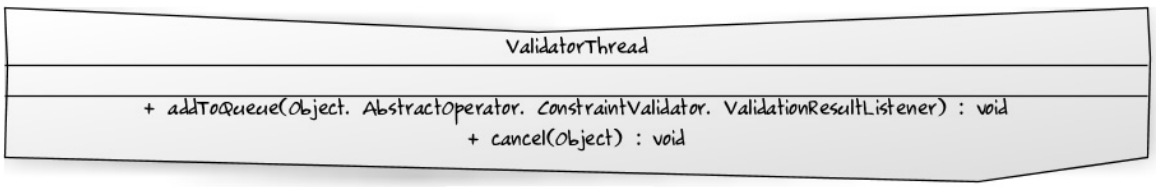

Figure 6.16: Class definition of *ValidatorThread*.

after calling the method *addToQueue(...)*. An identifier is used for each constraint which enables elimination of duplicate tasks for the same constraint in order to minimize the execution time. Tasks can also manually be removed from the queue by calling *cancel(...)*. After a validation task is finished the result is published via callback to the *ValidationResultListener*.

6.4.2 External interface of LTLCreator

LTLCreator should be integratable into other state machine based programming environments. This is ensured by its implementation as a Java Swing component and a clear API which enables a wide range of use for this tool.

Figure 6.17 points out interactions between *LTLCreator* and programming environments it is integrated in. The programming environment provides the state machine

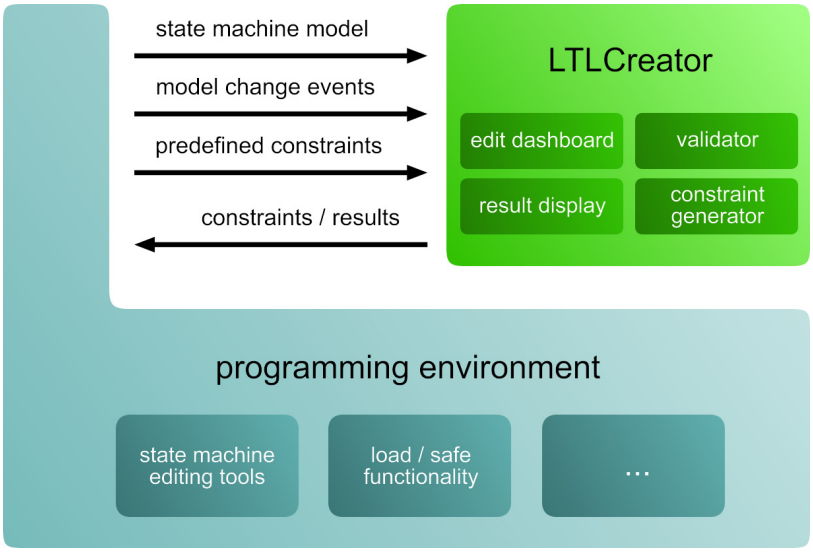

Figure 6.17: Interaction between LTLCreator and programming environment.

model and notifies the *LTLCreator* component about all changes made to the model. This will automatically revalidate all existing constraints and always hold the up-to-date model for constraint generation. For load and save functionality, all existing constraints can be received and new ones can be added to the *LTLCreator* component programmatically. All this functionality can be accessed through methods provided by the component *LTLCreator*. Its class definition is shown in figure 6.18. Whenever there is a change to the state machine, *setModel(Fsm)* has to be called with the current state machine as parameter. This will cause the editor to revalidate where necessary. Constraint exchange can be done by using the methods *addNewDashboard(AbstractOperator, ...)* and *getOperators()*.

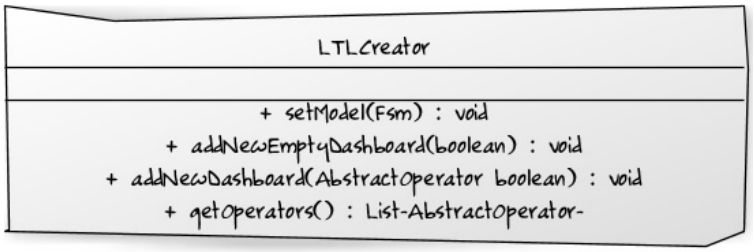

Figure 6.18: Class definition for *LTLCreator*.

6.5 Integration into Robostudio

As stated in the previous section, the *LTLCreator* provides an API for integration into other Java based applications. Robostudio, a visual programming environment for rapid authoring and customization of complex services on a personal service robot, is such an application which was used for evaluating the visual language as well as the automated constraint generation feature and is also supposed to provide all this functionality for development of service robot behaviour in future.

Robostudio uses NetBeans [14] as a platform and organizes all tools and features in windows. Thus also *LTLCreator* was put into a window and added to the Robostudio platform. This was done by placing it into a NetBeans module which was added to Robostudio's modules. This module imports the *LTLCreator* tool, archived as a jar file, and further consists of only two classes: *LTLCreatorTopComponent* and *XmlDataToLTLEditorDataWrapper*. The former implements the *TopComponent* provided by NetBeans which can be used for displaying UI elements. In order to visualize the *LTLCreator* component it is directly placed on the *LTLCreatorTopComponent*. It retrieves changes of the model and passes them to the *LTLCreator* component. Since the defining of safety constraints is an important issue not only after but also during the process of developing service robot behaviour, the *LTLCreator* window as a vital tool has been positioned in the center of the platform where also the *UI Component Layout Editor Window* is located. The *XmlDataToLTLEditorDataWrapper* forms the bridge between Robostudio's own model and the state machine model used by *LTLCreator*. All complex data coming from Robostudio has to be cleaned from

unimportant layout information and transformed to an elementary state machine representation before it can be processed by *LTLCreator*.

6.5.1 Problems

As already proposed in section 6.4.2, all changes to the state machine program made within Robostudio must of course cause the *LTLCreator* module to revalidate all existing constraints. This demands a concept of change notification management which is responsible for receiving all changes and making this information available for the *LTLCreator* component. As a solution, all parts of the code where model changes are made should be extended with short code snippets which send notifications. However, this is not as easy to realize in Robostudio since some of the relevant code was generated or even imported as external jars which makes code manipulation impossible.

A second challenge is the current data handling policy in Robostudio. Since this platform is mainly made for developers who used to edit RBDL xml code manually, Robostudio aims for visual rapid editing rather than for syntax checks. Thus, syntax errors are not detected or even ignored by methods like static analysis or input restrictions. In fact, such features have not been requirements for Robostudio. During integration of *LTLCreator* into Robostudio problems came up when loading already existing RBDL programs of the healthbot project: Transitions leading to non existing states, transitions leading to nowhere (null) or transitions whose target is depending on runtime facts. This characteristic seems to be a problem for the *LTLCreator* which needs a complete and correct state machine as input and does not know how to cope with inconsistent data retrieved from Robostudio.

6.5.2 Solutions

To guarantee that all model changes cause the constraints to be revalidated, all relevant code parts have to report on this. But modification of the code has not been possible in all cases. Whereas all state changes are announced, transition changes stay unnoticed. This decision is sufficient for the prototype, but the developers of

Robostudio announced an adapted architecture to solve this problem in the next version.

The problem with inconsistent data could be solved by plausibility and syntax checks during the transformation process of *XmlDataToLTLEditorDataWrapper*. After successful transformation the state machine is passed to the *LTLCreator* component. Otherwise no validation result is displayed for constraints and a dialog lists all identified syntax errors. In the context of safety and reliability a restrictive approach which gives no result at all seemed more applicable than an attempt to interpret the inconsistent data.

7 Test scenario and evaluation

Both, the visual language and the automated constraint generation functionality, have been integrated into Robostudio in order to evaluate the proposed safety functionality. Robostudio is a visual programming environment which allows the specification of state machine based programs whose behaviour can then be examined by visual constraints. As a test scenario the simplified medication reminder application is considered whose underlying screen-flow is explained in figure 7.1. Starting from "Menu" the user can choose from several services such as blood pressure measurement or entertainment. In the given example these services are simplified to just one state "additional services" since the focus concentrates on the medication reminder part. In state "Polling" the database is checked for a pending reminder and, if there is one, the medication intake guidance is triggered. The patient can state if he or she has already taken the medication, or decide whether to take it or not. In the latter case, he or she may give a reason for it and a caregiver is notified about this incident. Otherwise the progress will result in the state "Well done!" after completing intake.

This state machine with all its states, transitions and screen dialogs has been implemented in Robostudio and in the following sections 7.1 and 7.2 the visual language as well as the potential of constraint generation shall be evaluated. Section 7.3 describes how a non-expert deals with the visual language and what the final judgement is.

7.1 Operator constraints

Before constraints can be created it might be useful to think about a reasonable term to be expressed. Let's take the example "Whenever there is a pending reminder, medication will be finally taken or caregivers will get notified in case of the patient

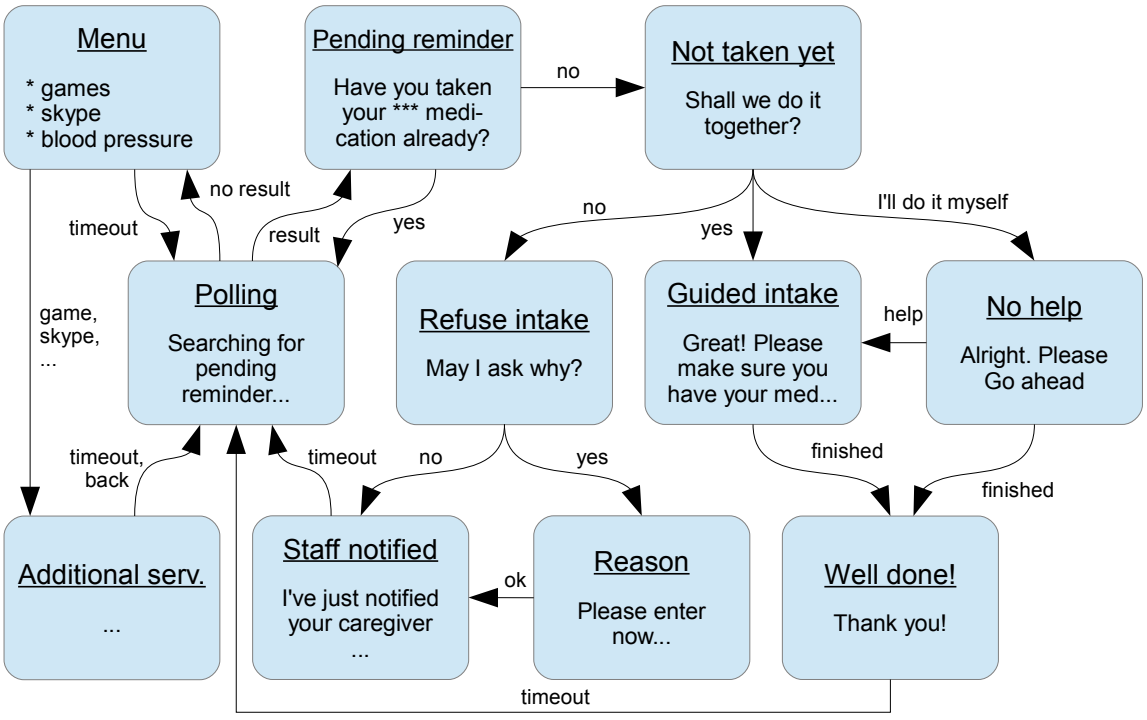

Figure 7.1: Simplified screen-flow of the medication reminder application.

refusing medication intake." given as a requirement in chapter 3.

In order to find a corresponding graphical constraint the sentence has to be analyzed just as developers read it. Since "whenever" is a semantic equivalent to "always if" first of all an *ALWAYS* operator gets dragged to the dashboard directly followed by an *IF* as shown in figure 7.2. The condition for this *IF* is that there is a pending reminder, so a state "Not taken yet" has to be added to the upper bucket of the *IF* operator. Whenever the just mentioned condition becomes true, there also has to be true in future: Medication is taken properly or caregivers get notified about refuse. Accordingly a *FUTURE* operator containing an *OR* forms the second part of the *IF* operator. Finally two states 'Well done!' and 'Staff notified' get added to the disjunction. As soon as the visual constraint is complete, it is translated to the following corresponding textual LTL formula by the editor:

$$\vDash \square(\text{'Not taken yet'} \rightarrow \Diamond(\text{'Well done!'} \vee \text{'Staff notified'})) \tag{7.1}$$

After each change on the dashboard, the constraint is automatically recompiled and revalidated by using the underlying model checker. For syntactically invalid constraints, an "incomplete" sign is displayed in the respective tab. Otherwise, an animated ring indicates that validation is in progress and will finally result in either a "valid" or "invalid" sign.

7.1.1 Usability

Due to the optimized performance of NuSMV, even the validation of constraints on huge and complex programs is fast and allows rapid feedback. In addition, the validation itself runs in the background without locking the dashboard. Therefore waiting times and disruptions during constraint development can be avoided, which would likely be the case with conventional model checking where constraint development and constraint validation are alternating processes. Thus the *LTLCreator* tool is considered an improvement for the developer experience.

It could be observed that the different color flavours of the visual operators are a good support for fast reading and understanding of constraints. Also the two

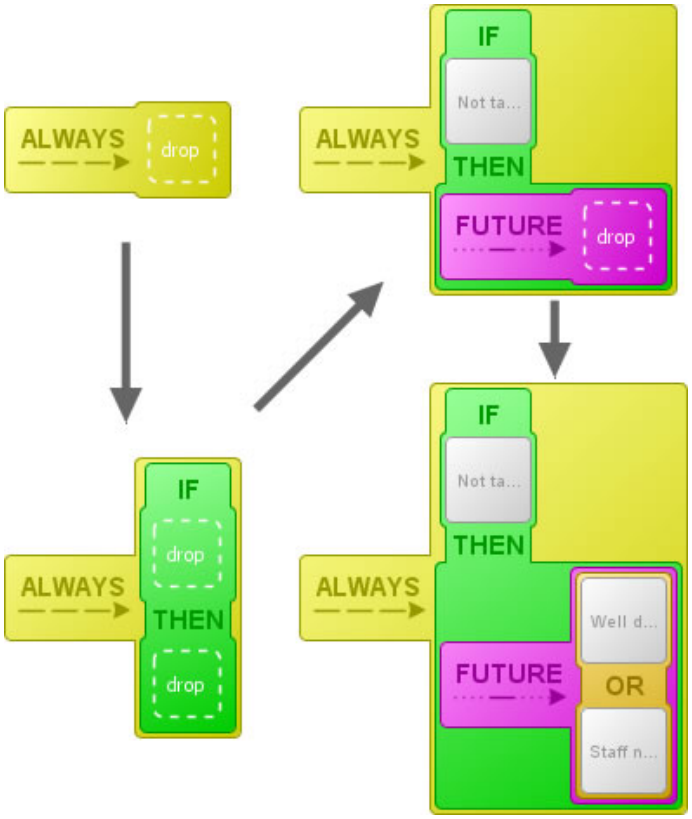

Figure 7.2: Visual constraint creation for LTL formula 7.1.

dimensional composition and the round shapes of the operators turned out to give the language a schematic but not too rectangular look. Besides it is considered eye-candy what is the best motivation for using this visual language.

7.1.2 Expressiveness

Due to being based on LTL, the visual language's expressivenes is equivalent to LTL's one of course. More precisely, that means any temporal logic formula can be expresssed except potentiality. Furthermore there is one more limitation in the current visual language concerning the variety of available proposition operators. The visual language was originally designed for validating healthcare applications where just one kind of proposition is needed: "currently state x is active". Thus other propositions such as equations, comparisons with greater than etc. are not implemented yet, but can be easily integrated. Figure 7.3 depicts how an extension of *LTLCreator* with additional proposition operators could be realized.

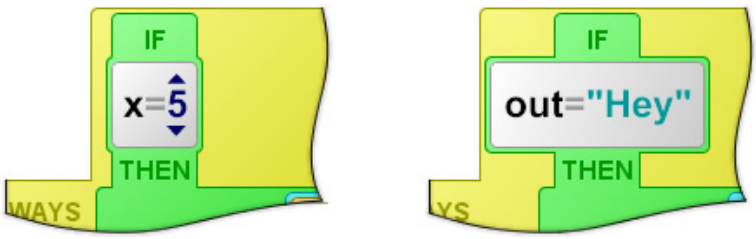

Figure 7.3: Possible proposition extensions: Numeric and string equations.

7.2 Automated constraint generation

The automated constraint generation can be triggered by a click on the magic wand button in the tab area. After activation a dashboard is opened in a new tab for each constraint found, and validation is initiated immediately. For the medication reminder application six constraints are found, including *a)* and *b)* shown in figure 7.4 which match the postulations in the requirements in chapter 3. The constraint used

for demonstration in the previous section is also generated, however *b)* forms an intensified restriction of it. Constraint *a)* ensures the "Polling" state is always eventually visited again. If a pending medication is not already taken, constraint *b)* guarantees intake or staff notification before the next reminders can be read from the database. Two more constraints are found which might be quite interesting for other programs; however, they do not contribute to safe behaviour of the medication reminder example and are less relevant. The search and generation process finishes within less than a second and thus does not let developers wait for a long time.

It was showed that the subgraph approach is working for the medication reminder application, and there was even one more reasonable constraint found by the heuristic: Constraint *c)* ensures that during the medication intake process the program can not switch back to the menu or other services such as entertainment or blood pressure measurement.

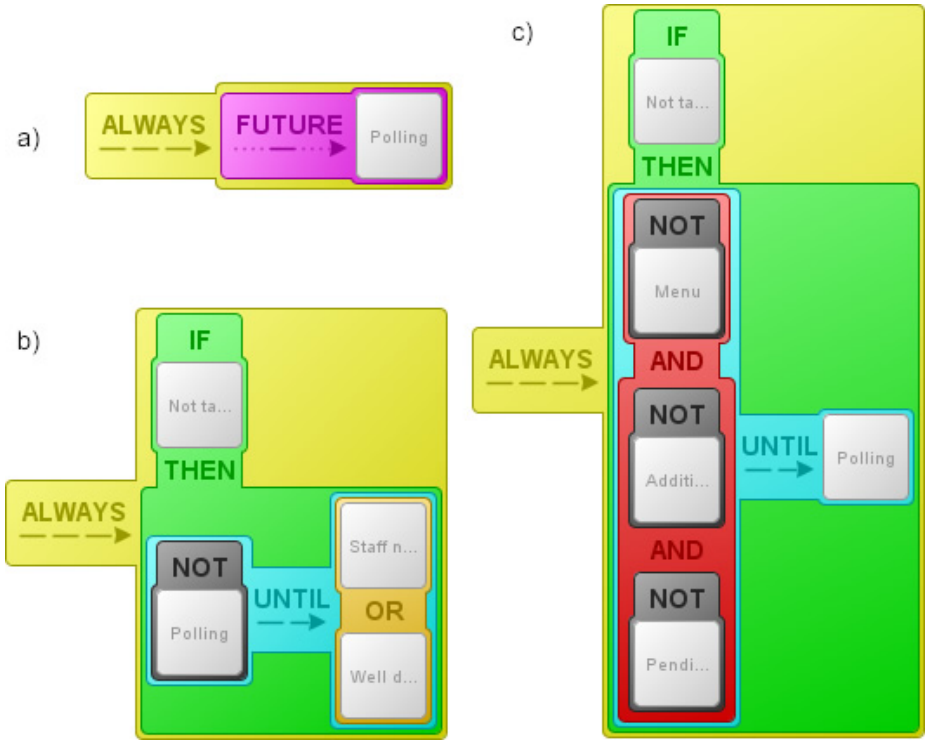

Figure 7.4: Automatically generated constraints which match the required postulations.

7.2.1 Performance

In order to examine the performance of subgraph finding and constraint generation, the implementation details of the algorithm have to be analyzed. The subgraph finding algorithm is realized similar to a depth-first search. Indeed all healthcare programs are infinite and contain cycles and a depth-first walk-through would never end, but as stated in section 6.3 the algorithm does not revisit already visited states.

The performance of this algorithm of course highly depends on the grade of branching. In contrast to a usual depth-first search in trees, the modified depth-first search is applicable for even cyclic state machines and will walk through all possible paths starting from one particular start state but without visiting states twice. Being S the set of all states and $|S|$ the number of all states, and assuming the worst case in which every state has transitions to all other states, each of the remaining $|S|-1$ states can be possibly visited next, similarly $|S|-2$ for the third step and so on. As a worst case execution time the following result is obtained:

$$O(\text{“depth-first search”}) = O((|S|-1) * (|S|-2) * ... * 2 * 1) = O((|S|-1)!) \quad (7.2)$$

In order to find a subgraph with the presented algorithm the depth-first search has to be startet on a start state of a subgraph. Since it is not known in beforehand which states might be possible candidates for such start states, the algorithm has to be applied on every single state of the program which has two ore more outgoing transitions. In the worst case all $|S|$ states fulfil this condition, so the depth-first search has to be executed $|S|$ times. This implies for the estimated worst case execution time:

$$O(\text{“subgraph finding”}) = O(|S|) * O((|S|-1)!) = O(|S|!) \quad (7.3)$$

7.2.2 Scalability

For all the healthcare applications that were examined, the presented algorithm works efficient and fast. But their program sizes are still quite manageable. The test

program has eleven states, an average branching grade of $1.\overline{63}$ outgoing transitions per state and a constraint finding execution time of a split second on a standard computer. Of course an exponential or factorial execution time is bad if the number of states arises. A clear slowdown can therefore be felt if the algorithm runs on larger state machines with plenty of states and a high grade of branching. A test on a state machine with over 250 states and partially high branching grade resulted in approximately five minutes execution time.

However, this algorithm for subgraph finding is not sophisticated regarding efficiency. It is rather a prototype to show the concept of automated constraint generation working and to provide this feature in the editor for evaluation. In case that this algorithm shall be used for other huge programs regardless its tailoring to the healthcare domain, the author is convinced that a more efficient algorithm for subgraph finding might be found by researchers.

7.2.3 Reasonability of generated constraints

The constraint generator algorithm found six constraints for the medication reminder application. Two of them were actually the ones proposed in the requirements. Three of them are either intensified versions of the proposed ones or just describe the behaviour of the program and therefore make sense, but in matters of safety they are not fundamental relevant. Nevertheless the generator tool actually found a new important constraint which had not been considered before. It is constraint *c)* in figure 7.4 of section 7.2.

To put it in a nutshell, in context of this example all automatically generated constraints are practical, and approximately half of them are significant. This shows that the constraint generator concept works well for the healthcare application and can be a very helpful feature.

7.3 Practical experiences

Some parts of the healthcare robot's behaviour are appropriate to be specified by non-programmers, and that is the purpose of the presented visual language as well. Different tools and environments, among Robostudio and *LTLCreator*, try to provide a particular interface suitable for such people. Now, it might be interesting to find out about user's experiences and how they get along when using these tools.

The *LTLCreator* has been presented to several people, technical engineers as well as ordinary persons. The overall feedback was good and the idea and realization of the visual language was well-received. One of the laypersons who have been introduced to *LTLCreator* was Alina Kracker. The 21 yeras old German student of educational science is also working for the elderly in a care facility since more than five years. In this social institution Ms. Kracker takes care of persons having different disease patterns and necessities. This spectrum varies from physical disabilities over mental problems such as Alzheimer's desease to being bedridden. Her professional care background makes her feedback about usability and comprehensibility of the visual concept especially interesting since its application and testing was done in the healthcare domain.

Ms. Kracker was introduced to the example of medication reminder, and the program behaviour was described with the help of figure 7.1. Afterwards she got familiarized with surface and functionality of *LTLCreator*. Now, she was asked to formulate the two constraints postulated in the requirements, just by using the available means of the editor placed at her disposal. The first constraint, which ensures the state "Polling" always eventually to be visited again, was constructed quickly and accurately by her. The second constraint specifies, that every visit of state "Not taken yet" will eventually result in a visit of either state "Well done!" or "Staff notified". It turned out that this constraint held some difficulties for her: The *FUTURE* operator was not considered initially by her. But it was also finished after a few consideration breaks.

Ms. Kracker did not know in beforehand what state machines are and never came in contact with LTL or model checking in general. But she dealt with this example

and the visual language surprisingly well, and she is convinced that she could use it after some exercise. As reason for the success she pointed out the easy readability of the visual language and its close relationship to spoken language. Furthermore the intuitive drag and drop allowed fast editing. Her examination on the visual language gives the evidence of success with the approach of creating an easy to use concept for defining LTL formulas. The visual language developed in this work enables even non-professionals to use the usually quite complex concept of model checking.

To the question, whether she would recommend such robotic services like the medication reminder application for use in the care facility she is working in, Ms. Kracker replied: "I can envision such robot applications beeing an excellent assistance for the elderly who are mentally fit but maybe a little doolally. However, in our facilities most of the patients are not capable enough anymore of making decisions based on what the robots ask. In fact, due to frequent changes of the patients in our facilities it is a big issue for us care givers not to forget time-dependend medication intakes of our patients. Thus, such a reminding robot application would constitute an excellent support for us staff!".

8 Conclusion & future work

This work presented a visual formalism for the defining of safety constraints and first steps towards a concept of automatic constraint generation. Both features are implemented and assembled in an editor for standalone or integrated use. The visual language is reasonably simple which makes it accessible to a wider range of software developers. Nevertheless it is reasonably expressive and allows serious model checking.

The concept of applying hueristics for proposing possible constraints to users of the visual language produced very good results in the healthcare use case, the medication reminder application. All postulated constraints were found and even a new reasonable constraint was discovered. Thus we are convinced that this approach, as well as the visual language for defining constraints itself, has the potential to really support users in development of safety critical robot applications in the healthcare domain and also other domains.

Both the visual language and the automated constraint generation have been indeed developed in the context of the healthcare domain, particularly of the rather simple medication reminder example. But the goals achieved in this work are not limited to it at all. The results, in particular the visual formalism and the concept of developer support by automated constraint finding, can rather be transferred to completely different domains and use cases.

Therefore, perhaps the most important next step which can be identified is the application of the language and the constraint generation heuristic to other domains and applications. This will reveal if the identified subgraph-based heuristics for constraint generation provides reasonable results also in different scenarios. Most likely, it will be necessary to identify further heuristics and integrate them in the visual language editor to improve developer support. A study among a group of developers from the

healthcare robotics domain could indicate the potential of usability improvements.

Some improvements on the feature side of the language or the editor were already identified. For example, the state proposition (currently the only supported proposition type) used in constraint formulation might be not enough for other domains. Supporting propositions based on strings or numerical equations might be necessary. For reasons of enhancing the visual language, a substitution operator could be provided. Already existing constraints could be reused as parts of more complex constraints and thus simplify the overall look. Finally, the only result of constraint checking presented to the user currently is whether the constraint is valid or invalid. In contrast to conventional model checking no additional explanation for the outcome such as a counter example is given. It can, for example, be considered to extend the visual language with support for bug identification by a visualization of counter examples.

Bibliography

[1] C. Datta, C. Jayawardena, I.H. Kuo, and B.A. MacDonald. Robostudio: A visual programming environment for rapid authoring and customization of complex services on a personal service robot. In *Intelligent Robots and Systems, 2012. IROS 2012. IEEE/RSJ International Conference on*. IEEE, 2012.

[2] A. Sisiruca and D. Ionescu. A visual-programming environment for a temporal logic language. In *Electrical and Computer Engineering, 1993. Canadian Conference on*, pages 245 –248 vol.1, sep 1993.

[3] A. Del Bimbo, L. Rella, and E. Vicario. Visual specification of branching time temporal logic. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 61 –68, sep 1995.

[4] A. Rugnone, E. Vicario, C.D. Nugent, M.P. Donnelly, D. Craig, C. Paggetti, and E. Tamburini. Hometl: A visual formalism, based on temporal logic, for the design of home based care. In *Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on*, pages 747 –752, sept. 2007.

[5] C. Jayawardena, I. Kuo, C. Datta, R.Q. Stafford, E. Broadben, and B.A. MacDonald. Design, implementation and field tests of a socially assistive robot for the elderly: Healthbot version 2. In *IEEE International Conference on Biomedical Robotics and Biomechatronics*, pages 1837–1842, Rome, Italy, June 24-27 2012.

[6] P. Tiwari, J. Warren, K. Day, B.A. MacDonald, C. Jayawardena, T. Kuo, A. Igic, and C. Datta. Feasibility study of a robotic medication assistant for the elderly. In *Australasian User Interface Conference (AUIC)*, 17–20 January 2011.

[7] C. Jayawardena, I.H. Kuo, U. Unger, A. Igic, R. Wong, C.I. Watson, R.Q. Stafford, E. Broadbent, P. Tiwari, J. Warren, J. Sohn, and B.A. MacDonald. Deployment of a service robot to help older people. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ InternationalConference on*, pages 5990 –5995, oct. 2010.

[8] M. Medoff, R. Faller, and R. Smith. *Functional Safety - An IEC 61508 SIL 3 Compliant Development Process, 2nd edition.* exida.com LLC, sep 2012.

[9] Dr. C. Harper. Current activities in international robotics standardisation. http://www.europeanrobotics12.eu/media/15090/2_Harper_Current_Activities_in_International_Robotics_Standardisation.pdf, 2012. Accessed: 15/11/2012.

[10] R. Bostelman. International standards efforts towards safe accessibility technology for persons with disabilities: Cross-industry activities. http://www.nist.gov/el/isd/upload/Stnd_Eng_Journal.pdf, aug 2010. Accessed: 15/11/2012.

[11] Statistisches Bundesamt Wiesbaden. Alter im Wandel - Ältere Menschen in Deutschland und der EU. https://www.destatis.de/DE/Publikationen/Thematisch/Bevoelkerung/Bevoelkerungsstand/AlterimWandel.html, jan 2012. Accessed: 15/11/2012.

[12] Statistisches Bundesamt Wiesbaden. Bevölkerung Deutschlands bis 2060 - 12. koordinierte Bevölkerungsvorausberechnung. https://www.destatis.de/DE/PresseService/Presse/Pressekonferenzen/2009/Bevoelkerung/pressebroschuere_bevoelkerungsentwicklung2009.html, nov 2009. Accessed: 15/11/2012.

[13] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications.* Springer-Verlag Berlin Heidelberg New York, 2004.

[14] R. Kusterer. Netbeans ide - netbeans rich-client platform development (rcp).

http://netbeans.org/features/platform/index.html, oct 2012. Accessed: 15/11/2012.

[15] F. Wagner. *Modeling Software with Finite State Machines: A Practical Approach.* Auerbach Publications, 2006.

[16] J. Carroll and D. Long. *Theory of Finite Automata with an Introduction to Formal Languages.* Prentice Hall, Englewood Cliffs, 1989.

[17] C. Baier and J.-P. Katoen. *Principles of model checking.* The MIT Press, may 2008.

[18] T. Bickmore and T. Giorgino. Health dialog systems for patients and consumers. *Journal of Biomedical Informatics*, 39(5):556 – 571, 2006. Dialog Systems for Health Communications.

[19] D. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *Software Engineering, IEEE Transactions on*, 35(6):756 –779, nov.-dec. 2009.

[20] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:410–425, 2000. 10.1007/s100090050046.

[21] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 241–268. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45657-0_29.