Graphical formalization and automated computing of safety constraints in robotics

Ludwig Nägele
Department of Software Engineering
University of Augsburg
86135 Augsburg, Germany
Email: mail@ludwig-naegele.de

Andreas Angerer
Department of Software Engineering
University of Augsburg
86135 Augsburg, Germany
Email: angerer@informatik.uni-augsburg.de

Bruce A. MacDonald
Department of Electrical Engineering
The University of Auckland
Auckland 1142, New Zealand
Email: b.macdonald@auckland.ac.nz

***Abstract*—Nowadays robots are no longer confined to only industrial use but increasingly assigned to jobs within human workspaces. Even the programming of robots is shifting from experienced software developers to the consumers themselves who need to customize the robot's behavior in order to match individual requirements. However both the human robot collaboration as well as the behavior 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 behavior specification by consumers.**

In this paper we present a visual language for the defining of safety constraints for state machine definition of robot behaviour. Our approach addresses mainly non-safety-experts and our abstraction from a mathematical temporal logic expression to a more intuitive visual representation is intended to enable a wider range of software developers to do model checking. Furthermore we propose a new concept of development support by providing automated constraint generation functionality. 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 demonstrates both visual constraint creation functionality as well as automated constraint generation.

I. Introduction

Safety critical robot applications require extensive testing or formal verification in order to achieve adequate safe and predictable behavior. 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 which guides a person through the process of medication intake, such as the healthcare robotics project at the University of Auckland, New Zealand [1]. 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 [2] which run on mobile robots 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 [1], [3]. The applications are intended to be developed by healthcare professionals using Robostudio [4], a visual programming environment for rapid authoring and customization of complex robot services. Generally these people don't have profound knowledge of the complex mathematical syntax used for formal temporal logic such as computation tree logic (CTL) or linear temporal logic (LTL). 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 notifications are sent to staff members whenever medication is not taken by the patient, or to avoid duplicate reminders for medications which have already been taken. So a mechanism is needed for the specification of such safety constraints; one that does not require special or expert knowledge and thus is easy to use for all kind of programmers. A visual language intended to fill this gap is presented in section IV-A.

Unfortunately the supply of an easy-to-use editor for constraint creation doesn't 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 development process. In Sect. IV-B we introduce a heuristic for finding reasonable safety constraints based on state machine descriptions that describe robots' behaviors.

II. Related work

A first step towards a graphical tool for designing safety constraints is taken by Sisiruca and Ionescu [5]. They developed an object-oriented graphical environment for visually creating temporal logic sentences and rules. The tool presents temporal operators as logical modules with boolean inputs and outputs which can be connected to one big expression

Preprint submitted to 2013 IEEE International Conference on Robotics and Automation. Received September 17, 2012.

graph. Del Bimbo et al. worked on a visual tool for temporal logic, but with a special focus on hierarchical representation of formulas [6]. Here each node of a tree is described by an abstract operator whose subformulas are branches to their replacement operators. Thus the root node represents the entire formula. They also explored the formula trees' 3D representation within a virtual space. Both approaches are different fundamental ideas of representing temporal formulas. Whereas having a graph seems not to be applicable for our approach, the idea of hirarchical operator representation of Del Bimbo et al. will be a good basis for our concept of constraint creation and understanding.

The step towards a more abstract level of development is taken by HomeTL [7], a visual formalism for the design of home based care. It allows healthcare professionals to specify rules and sequences of user 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. Even though this approach touches the subject of healthcare it barely matches our problem. HomeTL focuses more on monitoring temporal boundaries of a patient's behavior than on ensuring functional safety of an implemented program. It does depict how nontechnical constraint design can be realised and its graphical design is a good example of handling time dependencies.

III. Goals

We want to introduce a visual language for defining safety constraints in state machine definitions of robot behaviour, that is suitable for people who are not experts in formal methods. The visual language should be easy to use and intuitive to read, in order 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 we want to integrate some mechanism to support users in finding suitable constraints for a certain application. We assume that applications are modeled by state transition graphs. This modeling paradigm is used in many healthcare robots that employ dialogue systems for communicating with users [8]. Other possible fields might be shop assistant robots or entertainment robots. For such systems, our approach is suitable. It is based on heuristically analyzing state transition graphs and deriving constraints related to the graph structure. In this work, we want to investigate a particular heuristic as a first step towards user support.

Finally, we will present some results of applying this heuristics to the medication reminder example described before. The minimum goal that should be achieved was the identification of the following constraints:

- The application will always eventually check the database for new reminder jobs. Thus we ensure that pending reminders are processed.
- Whenever there is a pending reminder, 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 don't take it. As our approach validates state machine behaviour rather than human attitudes we assume there is either cooperation from the patient or a separate tool that can verify medication intake. Furthermore we assume the correctness of external services used by the medication reminder such as the database module holding all reminders.

IV. Design

A. Visual formalism

The presented visual language is designed to express constraints for state machines. To provide the required functionality, the fundamental logical operators are required, as shown in (a) though (e) below. In addition the visual language should be capable of expressing constraints about future steps, both (h) any future state and (g) the next state, that (f) events should always happen, and that (i) a property must be true until some future event. Like in Del Bimbo et al.'s formula representation [6], a constraint consists of nested operators and propositions in a hierarchy. However we surrender the three dimensional idea and focus on two dimensional blocks and their compositions instead, what is also used by HomeTL [7]. Each block represents either an operator or a proposition and has its specified semantics. Thus a visual formula can be recursively translated to a linear temporal logic (LTL) sentence and be checked using any ordinary model checker.

The visual language has support for eight operator types and one proposition as shown in figure 1.

(a) *AND* operator: $\varphi \wedge \psi$.

(b) *OR* Operator: $\varphi \vee \psi$.

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

(d) *NOT* operator: $\neg\varphi$.

(e) Proposition: 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 our healthcare subject.

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

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

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

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

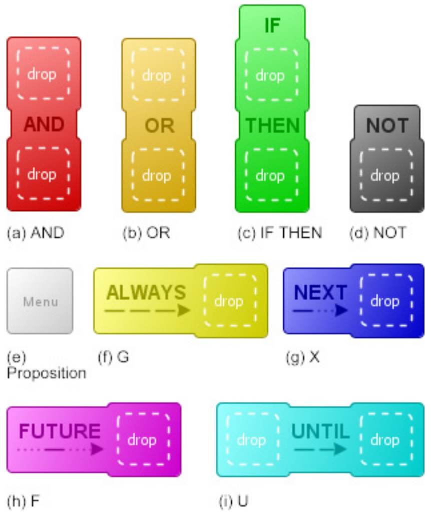

Fig. 1. The visual language consists of eight unary or binary operator types and one state proposition, each of them having its own color flavor. Logical operators are arranged in a vertical row wereas time relevant statements are aligned horizontally.

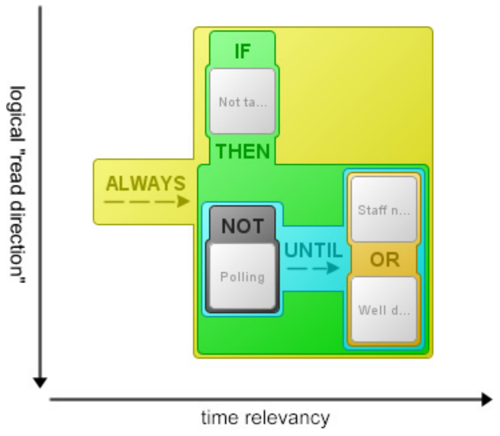

Fig. 2. Easy understanding of visual constraints due to intuitive read directions: "ALWAYS is true: IF state *'Not taken yet'* is active THEN state *'Polling'* can NOT be visited UNTIL state *'Staff notified'* OR state *'Well done!'* is reached."

To meet the requirements the visual language should be very easy to use, especially for non-experts. Furthermore it should provide an intuitive application which is ideally learnable within little time. So there are some design decisions this visual language must fulfill [9]:

- Each operator has a different significant visual appearance; in this case we have used color. This kind of classification enables a faster subconscious identification.
- The layout of constraints (nested operators) has two dimensions with a certain meaning. Figure 2 demonstrates the "logical" vertical read direction as well as the time releveant horizontal line.
- Multiple nested operators of the same kind which have no order priorities such as *OR* or *AND* appear unnecessarily complex. These operators can be merged.

Also an editor for this visual language 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 there is a trash can for deleting existing operators.
- In order to simplify the reading of constraints an operator can be hovered over by the mouse pointer. All higher operators become bleached out so that the hovered operator and its children appear highlighted. It's intended to be similar to line coloring when reading a digital text.
- Every edit 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 / editing except operator and proposition providers, which let the developer create new instances. Having no additional buttons reduces complexity and keeps it simple.

B. Automated constraint generation

In order to support the developer in specifying safety constraints, automated generation and suggestion of constraints might be helpful. But it's helpful only if the suggested constraints are relevant and reasonable. This however depends on the domain and use case. Regarding the important constraints postulated in Sect. III, all the relevant states 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, we deal with subgraphs defined as follows:

- There is a start state (α-state) with at least two outgoing transitions.
- There is one end state (β-state) in which all possible paths starting from start state come together again.
- All states between the start and end states 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 incoming and outgoing transitions from and to states not contained in the subgraph.

A graphical example of such a subgraph is shown in Fig. 3. Using this specification all possible such subgraphs can be detected within a state machine. If the subgraph contains neither inner (infinite) loops nor final states (states having no outgoing transition) we can derive the following valid LTL formulas for each subgraph (with α as proposition for "α is current state" and similarly for β and γ):

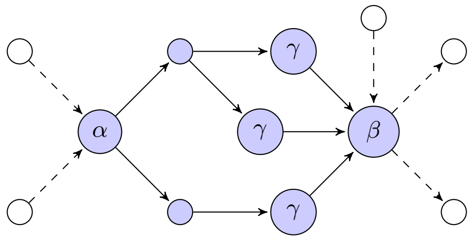

Fig. 3. Example for a subgraph.

$$\models \Box(\alpha \Rightarrow \Diamond\beta) \tag{1}$$

$$\models \Box(\alpha \Rightarrow \Diamond(\bigvee_{\gamma} \gamma)) \tag{2}$$

$$\models \Box(\alpha \Rightarrow [\neg\beta\,\mathcal{U} \bigvee_{\gamma} \gamma]) \tag{3}$$

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:

$$\models \Box(\alpha \Rightarrow [(\bigwedge_{\delta} \neg\delta)\,\mathcal{U}\beta]) \tag{4}$$

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:

$$\models \Box\Diamond\alpha \tag{5}$$

These five constraints apply to each subgraph matching the conditions mentioned above. Formula 1 says that an α-state is always eventually followed by a β-state. Formula 2 ensures that there will be always at least one γ-state be visited after each visit of an α-state. The quite similar but stricter formula 3 demands additionally a visit of at least one γ-state before the β-state can be reached. Formula 4 simply says that a state not contained in the subgraph can't be visited between α- and β-states. The constraint that the α-state will always eventually be visited again is expressed by formula 5.

All formulas can be translated to the visual formalism presented in section IV-A and shown to the user. The appropriate LTL formulas for the constraints postulated in section III relate to number 2 and 5 of the subgraph based formula templates and are as follows:

$$\models \Box\Diamond\ \text{'Polling'} \tag{6}$$

$$\models \Box(\text{'Not taken yet'} \Rightarrow \Diamond(\text{'Well done!'} \vee \text{'Staff notified'})) \tag{7}$$

V. Prototype and Evaluation

As stated in Sect. III, appropriate support by a graphical editor is important for the usefulness of the developed visual language. We therefore designed and implemented a prototype of such an editor. The proposed constraint generation heuristics is integrated in this prototype as well and could thus be evaluated. The implementation is a java swing component, making it easy to integrate the visual language into other java applications. An API enables an underlying state machine which also influences the state propositions available for constraint editing. Furthermore a particular model checker can be specified for constraint validation. As a default, the symbolic model checker NuSMV [10], [11] is used.

Fig. 4 shows a snapshot of the editor's environment: All functionality needed for constraint editing is provided in the tool bar on the right. It contains draggable elements for creating all operator and proposition types as well as a trashcan for deleting. Constraints can be composed in the dashboard in the center of the editor by drag&drop. The tab functionality on the left allows multiple constraints to be managed. Each tab shows a small thumbnail of the constraint and a symbol indicating its validity (a warning shield for "incomplete", a green shield for "valid", a red cross for "invalid" or an animated ring for "validation in progress.") The magic wand button within the tab pane triggers the constraint generation.

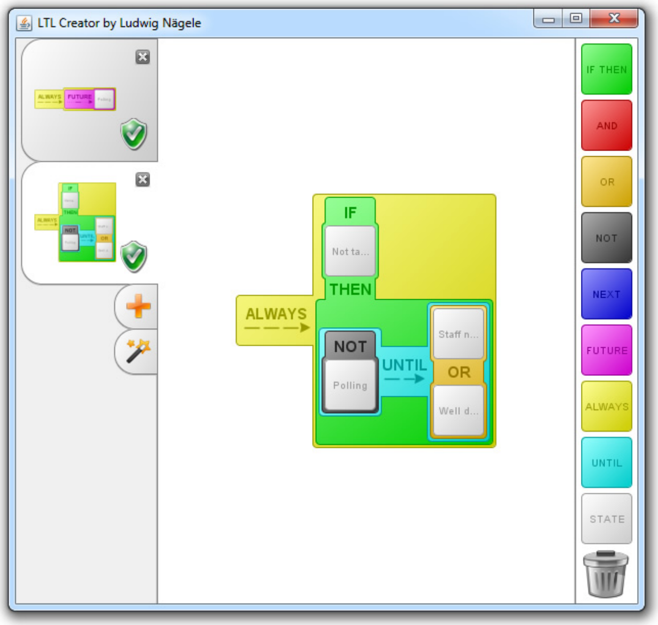

Fig. 4. Snapshot of the visual editor.

In order to evaluate the proposed safety functionality the tool was integrated into Robostudio [4]. This Visual Programming Environment allows the specification of state machine based programs whose behaviour can then be examined by visual constraints. As an example we consider the simplified medication reminder application whose underlying screen-flow is explained in Fig. 5. 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 we want to focus only on the medication reminder part. In “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 he or she may give a reason for it and a caregiver is notified about this incident. Otherwise it will result in “Well done!” after completing intake.

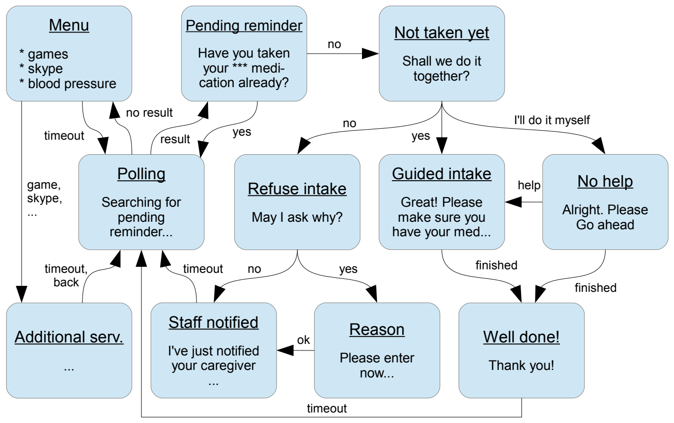

Fig. 5. Simplified screen-flow of the medication reminder application.

A. Visual constraints

Before we can start creating constraints it might be useful to think about a reasonable term we want to express. 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 refusing medication intake.” given in the requirements in section III.

In order to find a corresponding graphical constraint the sentence has to be analyzed just as you read it. Since “whenever” is a semantic equivalent to “always if” first of all a *ALWAYS* operator gets dragged to the dashboard directly followed by an *IF* as shown in figure 6. 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. The resulting visual constraint can now automatically be translated to the corresponding LTL formula by the editor:

$$\models \Box(\text{'Not taken yet'} \Rightarrow \Diamond(\text{'Well done!'} \vee \text{'Staff notified'})) \tag{8}$$

After each change in the dashboard, the constraint is automatically recompiled and revalidated by using the underlying

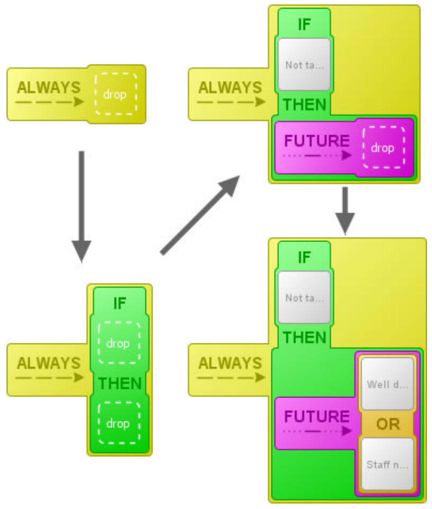

Fig. 6. Visual constraint creation for LTL formula 8.

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.

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 we avoid waiting times and disruptions during constraint development, which would be likely with conventional model checking where constraint development and constraint validation are alternating processes. Thus we consider our tool an improvement for the user experience.

B. 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 found constraint, and validation is initiated immediately. For the medication reminder application six constraints are found, including *a)* and *b)* shown in Fig. 7 which match the postulations in the requirements in Sect. III. 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.

We 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.

VI. Conclusion and Future Work

In this paper we presented a visual formalism for the defining of safety constraints and first steps towards a con-

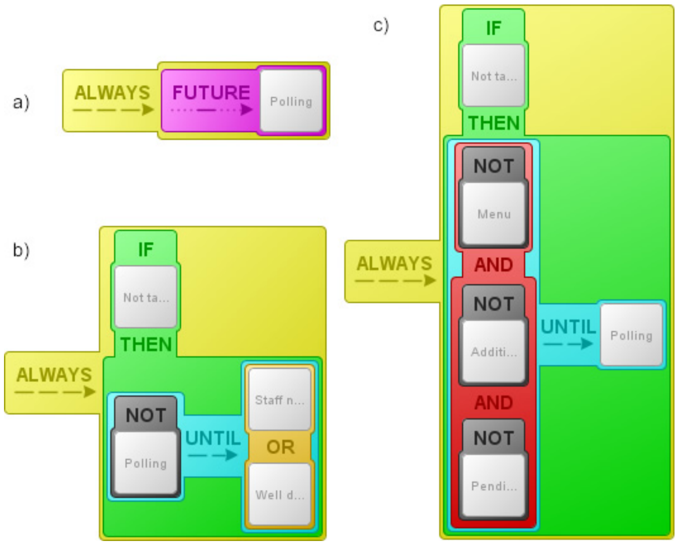

Fig. 7. Automatically generated constraints which match the postulations in the requirements.

cept 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 our use case, the medication reminder application. All postulated constraints were found and we even discovered a new reasonable constraint. 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.

Perhaps the most important next step we 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 heuristic 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 user support. A study among a group of users 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 mumerical equations might be necessary. 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 be considered to extend the visual language with support for bug identification by a visualization of counter examples, for example.

Acknowledgment

This work evolved from a master thesis project provided by the Elite Graduate Program Software Engineering. For the received support the authors would like to thank the people of this program and the three patron universities: University of Augsburg, Technical University Munich and Ludwig-Maximilians-University Munich.

References

[1] 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*, Rome, Italy, June 24-27 2012, pp. 1837–1842.

[2] T. P, W. J, D. K, M. BA, J. C, K. T, I. A, and D. C, "Feasibility study of a robotic medication assistant for the elderly," in *Australasian User Interface Conference (AUIC)*, 17–20 January 2011.

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

[4] C. Datta, C. Jayawardena, I. Kuo, and B. 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.

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

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

[7] A. Rugnone, E. Vicario, C. Nugent, M. 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*, sept. 2007, pp. 747 –752.

[8] T. Bickmore and T. Giorgino, "Health dialog systems for patients and consumers," *Journal of Biomedical Informatics*, vol. 39, no. 5, pp. 556 – 571, 2006, ¡ce:title¿Dialog Systems for Health Communications¡/ce:title¿. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046405001413

[9] D. Moody, "The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering," *Software Engineering, IEEE Transactions on*, vol. 35, no. 6, pp. 756 –779, nov.-dec. 2009.

[10] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, pp. 410–425, 2000, 10.1007/s100090050046. [Online]. Available: http://dx.doi.org/10.1007/s100090050046

[11] 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 *Computer Aided Verification*, ser. Lecture Notes in Computer Science, E. Brinksma and K. Larsen, Eds. Springer Berlin / Heidelberg, 2002, vol. 2404, pp. 241–268, 10.1007/3-540-45657-0_29. [Online]. Available: http://dx.doi.org/10.1007/3-540-45657-0_29