INSTITUT FÜR INFORMATIK

A Case-Study on Manual Verification of State-based Source Code Generated by KIELER SCCharts

Steven Smyth, Sören Domrös, and
Reinhard von Hanxleden

Bericht Nr. 1905
December 2019
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

A Case-Study on Manual Verification of State-based Source Code Generated by KIELER SCCharts

Steven Smyth, Sören Domrös, and Reinhard von Hanxleden

Bericht Nr. 1905
December 2019
ISSN 2192-6247

E-mail: {ssm, sdo, rvh}@informatik.uni-kiel.de

Technical Report

Contents

List of Figures

Abstract

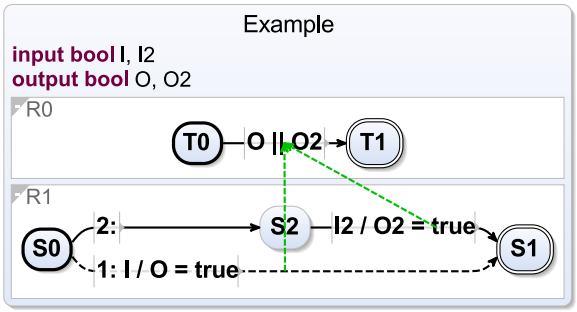
Statecharts-like languages, such as SCCharts, are commonly used to develop state-oriented reactive and critical systems. Code is often generated by automatic code generators, which employ different strategies. This paper presents the results of a second user study on manual user verification of different source codes, which were generated using a netlist-based, a priority-based, and a state-based code generation approach compiling SCCharts models to C. The evaluation shows that manual verification can be time-consuming and is error prone if the user has no clear mapping between states and transition of the original model and the generated code. The participants performed better if the generated code followed a state pattern that preserves original model structures and names.

1. Introduction

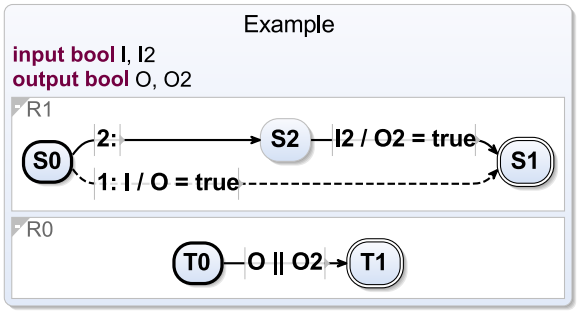
Harel introduced the Statecharts notation for visual languages in 1987 [Har87]. It has since become a popular means to model reactive and embedded systems, because state machines naturally describe the different modes of these systems. Since maintaining large projects becomes a tedious task, automatic code generators are commonly used. Nonetheless, authorities demand manual verification of the generated code before it can be deployed to safety-critical devices [Rie13]. *Readability* of the generated code and *mappability* to the original model are key factors for verifying.

SCCharts are a visual language that combines the Statecharts notation with sequentially constructive synchronous semantics [vHDM+14]. The official SCCharts compiler is included in the academic open-source project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). The tool chain linked to SCCharts is bundled under the name *KIELER SCCharts Tools*. By default, the compiler comes with three compilation strategies, namely the netlist-based, the priority-based, and the state-based compilation approaches. The first two strategies were part of the initial contribution on SCCharts [vHDM+14]. The latter one follows a common state pattern approach modified according to the structure of SCCharts models. It was presented in 2018 and compared to the previous approaches in a first case-study [SMvH18]. The case-study tested readability of the generated code by asking participants to reverse engineer an SCCharts model from the generated code. While the results were promising, code verification was of secondary concern. The follow-up case-study presented here explicitly aimed at verification. Participants were asked to find errors in the generated code of a steam boiler power plant model.

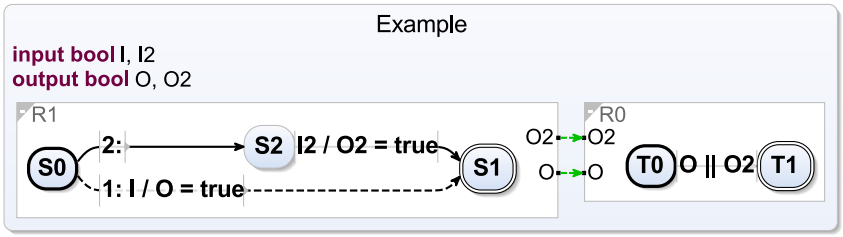
To increase readability of the generated source code even more, the state-based approach has been extended by a *lean mode*. The lean variant refrains from bi-directional concurrent broadcasting. Uni-directional communication between concurrent regions can be resolved by ordering the associated region functions statically without any synchronization code. This strategy is feasible if the models only comprise uni-directional communication or no concurrency at all. Fig. 1.1 shows the example model from the initial state-based contribution, but with swapped regions. Region R0 depends on region R1, because T0 is waiting for O or O2 to become true, which can be emitted by R1, as depicted in Fig. 1.1a. A region dependency processor inside the compilation performs a dependency analysis. The result is depicted in Fig. 1.1c. Afterwards, the regions are sorted accordingly in Fig. 1.1b. The source code can now be synthesized in the order of the regions. The complete compilation chain of the lean state-based approach is shown in Fig. 1.1d. More on the general capabilities of the KIELER compiler can be found elsewhere [SSRvH18b, SSRvH18a, Mot17, MSvH14].



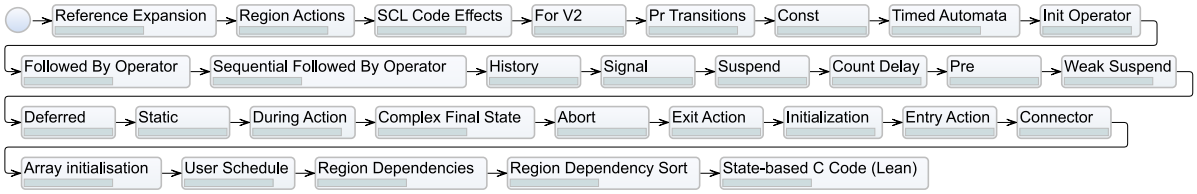
(a) Running example from the initial state-based contribution [SMvH18] with swapped regions.



(b) Regions are swapped automatically by the compilation chain due to the detected dependencies.



(c) The region dependency processor performs a dependency analysis. The result can be inspected in the diagram view.



(d) The complete compilation chain of the state-based lean approach. Intermediate results can be inspected interactively.

Figure 1.1.: Compilation system and region ordering of the lean state-based compilation approach

Chap. 2 presents the steam boiler model used for the verification study and describes its behavior. Chap. 3 describes the setup, the tasks, and the result of the case-study. Related work is presented in Chap. 4. Chap. 5 concludes the paper and presents future work on this topic.

2. The Steam Boiler Model

The steam boiler example is a well known model for cyber-physical systems, which has been implemented in several different languages [Abr96, BW96]. An SCCharts version of the core steam boiler part of that specification can be seen in Fig. 2.1. The inner behavior of the normal, degraded, and rescue state is depicted in Fig. 3.1a.

The steam boiler has four pumps that provide it with water. Each pump can be operated independently. In this example, however, they are always activated or deactivated at the same time to simplify the case-study. A water measuring system, which is not modeled here, calculates the quantity q of water that is currently in the steam boiler. The amount of water should be between the minimal normal quantity $N1$ and the maximal normal quantity $N2$. The water level has to be between the minimal limit quantity $M1$ and the maximal limit quantity $M2$ to prevent damage to the steam boiler. $M1 < N1 < N2 < M2$ must hold. To further simplify the generated code, boolean variables are used as signals to notify the system's components about external state changes, instead of broadcasting signals as in the original specification.

The behavior of the steam boiler can be expressed in five different execution modes:

- The initialization mode makes sure that the steam boiler is filled with an amount of water q that is within its normal quantity ($N1 \leq q \leq N2$). In this mode, the boiler is filled up or water is released by opening the valve if necessary. If all pumps work correctly, the initialization phase transitions in the normal execution phase. If they do not, the degraded state is activated.

- During normal execution, the steam boiler should only have a water level q between its normal, not safety-critical boundaries $N1$ and $N2$. The initial state in this mode assumes a valid water level (see initialization mode). Depending whether the water level falls below or above this range, pumps are activated or deactivated and a corresponding inner state is entered. For example, we take the transition from manageWaterLevel to tooLittleWater if $q < N1$. The inner state is left again if the water-level measured by the quantity q is within the normal range. The normal mode is left if parts of the pumping-system fail (degraded), the water measuring unit is defect (rescue), or the water level is no longer within its critical boundaries (emergency stop).

- In the degraded mode the system has the same behavior as in the normal mode as mentioned by the original specification. The system enters the degraded state if a physical unit, for example the pumping-system, fails. The degraded state is left if the broken parts are repaired (normal), a water measuring unit defect occurs (rescue), or the critical boundaries $M1$ and $M2$ are violated (emergency stop).

- During rescue mode only the calculation of q changes. The inner behavior is the same as in the normal and the degraded mode. In the rescue state the quantity q is calculated using v, the quantity of produced steam. This state is left if the corresponding broken systems are repaired (normal, degraded) or a defect of the steam measuring unit is detected (emergency stop).
- The emergency stop mode stops the whole system if it is no longer rescuable. The original paper specifies that the physical environment has to take appropriate actions to shut down the system. We model this by closing the pumps and releasing all water from the boiler by opening the valve.

The inner behavior of the normal, degraded, and rescue modes are modeled identically to make the task of identifying specific inner states non-trivial. This is also stated by the original specification.

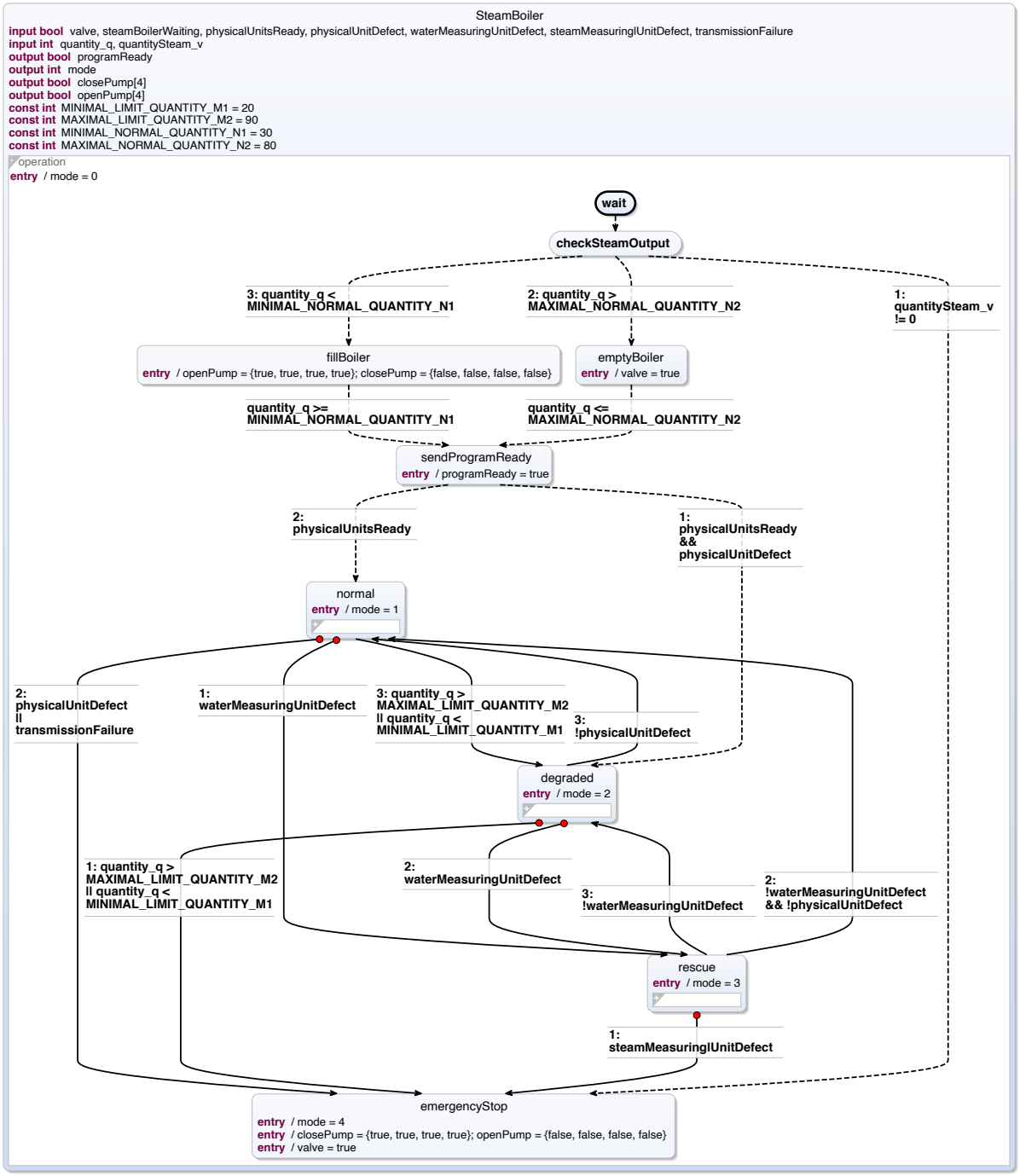


Figure 2.1.: Steam boiler [Abr96] core model implemented in SCCharts

3. Case-Study

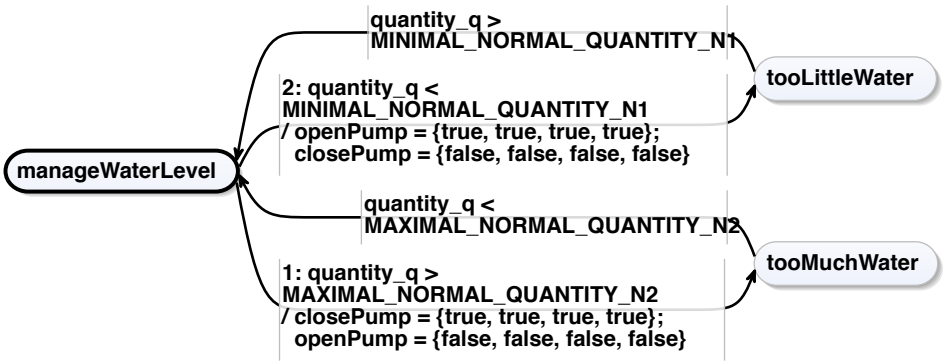
To test the hypothesis that the state-based code generation approach makes a program easier to comprehend, we conducted a verification study. Sec. 3.1 describes the case-study set-up. The results of the study are presented in Sec. 3.2.

3.1. Set-up

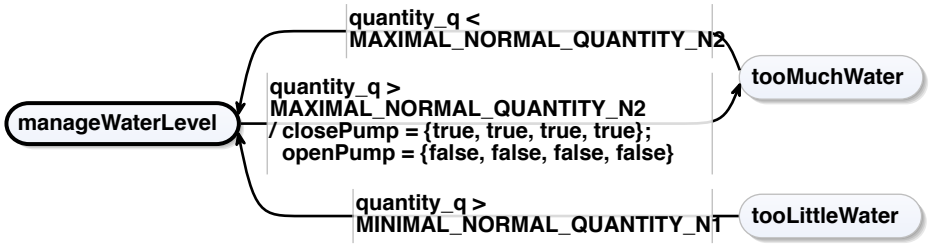
The diagram of the steam boiler model, as it is depicted in Fig. A.1, was given to the participants of the study in digital form as an SVG and also as A4 hard copy. They received a five minute introduction to the power plant set-up and a general explanation on the netlist-based, priority-based, and state-based code generation approach. Afterwards, the participants were asked to find a structural error within the generated C codes. For each approach, a structural error was included in the model. A structural error means that there is either

- an additional erroneous or missing state,
- an additional erroneous or missing transition,
- or a transition with wrong source or target state.

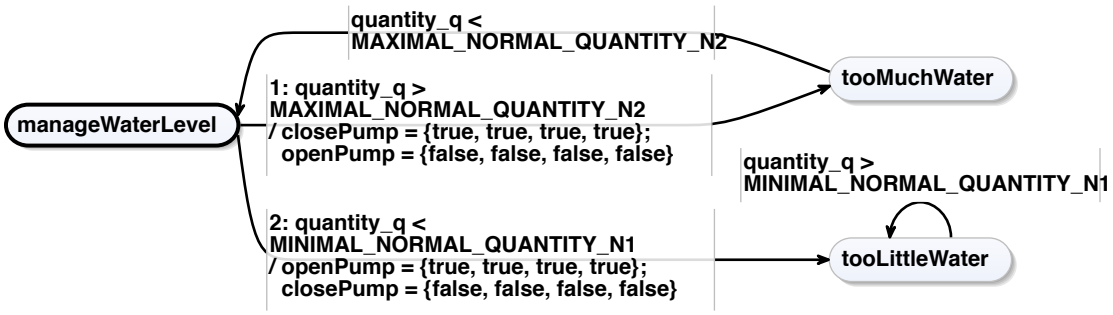
Moreover, the participants were informed that the wrong behavior has been located within the degraded state. It was not necessary to check outgoing transitions of the degraded superstate. The participants had 12 minutes to find the difference of the generated C code and the steam boiler model for each code generation approach. However, if they could not make any sense of the generated code, they were allowed to end the approach prematurely. To stay within the tight time constraints, each approach comprises exactly one structural error. The included errors are depicted as SCCharts in Fig. 3.1. Fig. 3.1a shows the original inner behavior of the degraded state. The participants faced erroneous code of the following alterations: In the first structural erroneous part (1) of the diagram, the transition from manageWaterLevel to tooLittleWater is missing, as seen in Fig. 3.1b. In the second faulty model part (2) in Fig. 3.1c, the transition from tooLittleWater to manageWaterLevel has the wrong target tooLittleWater. The third erroneous diagram (3), shown in Fig. 3.1d, has a new state pumpsOff. The pumpsOff state is always reached from manageWaterLevel since the transition to it has the highest priority and it is not guarded. Note that depending on the code generation approach, code optimization might throw away unreachable code. For example, error (1) (see Fig. 3.1b) results in a missing tooLittleWater state in the netlist-based code generation approach.



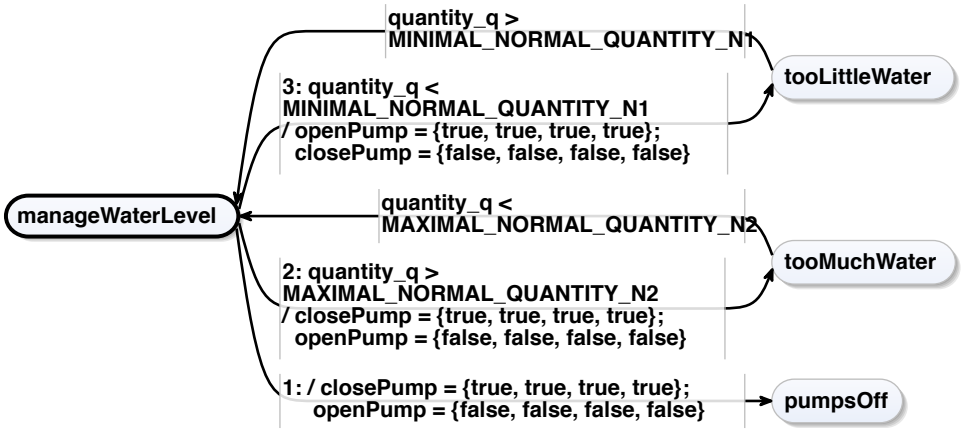
(a) Correct behavior



(b) Error 1



(c) Error 2



(d) Error 3

Figure 3.1.: Inner behavior of the degraded state

The participants were put into six different groups regarding the code generation approach they worked on first as seen in Tab. 3.1 to mitigate learning effects. Each group should include a reasonable number of people. With the expected number of participants, we did not vary the order of the errors, because shuffling the code generation approaches was more important.

	I	II	III
Group 1	netlist	prio	state-based
Group 2	prio	netlist	state-based
Group 3	state-based	netlist	prio
Group 4	prio	state-based	netlist
Group 5	state-based	prio	netlist
Group 6	netlist	state-based	prio

Table 3.1.: Order of code generation approaches for each group

42 students participated. They were all computer science students, either in the final terms of their bachelor's or in the master's degree programme. While having limited experience with SCCharts from attended lectures, they had only sparse knowledge about the different code generation approaches.

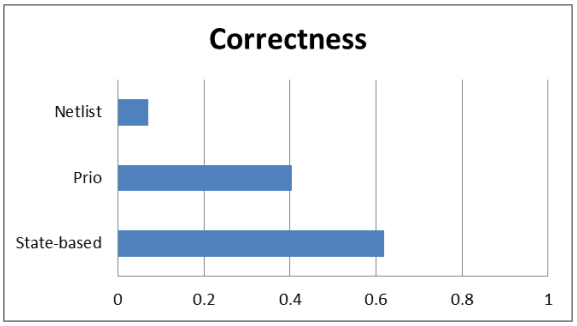
3.2. Results

Fig. 3.2 shows the results of the case-study in the categories *correctness*, *confidence*, and *time*.

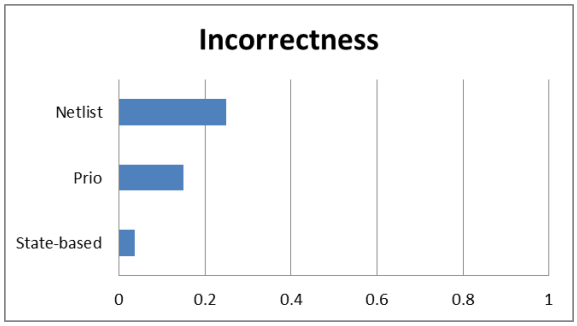
Correctness From the 42 participants, 61.9% found the diagram's issue in the state-based approach, 40.4% in the priority-based approach, and 7.1% in the netlist-based approach, which is depicted in Fig. 3.2a. Moreover, only 4, 20, and 27 participants named an issue within the netlist-, priority-, and state-based approaches respectively. In this order, 1, 3, and 1 answers were wrong, which is shown in Fig. 3.2b normalized to the answers given.

Confidence The participants should rate how confident they are with their answer on a scale from 0 (not confident) to 2 (confident). From the correct answers, the mean result of the confidence rating is depicted in Fig. 3.2c. While both, priority and state-based, are settled around a mean value of 1.7, the confidence in the netlist approach is lower at around 1.

The participants that did not find the issue were asked if they recognize any relevant parts of the diagram in the code, namely states, regions, or transitions relevant for the degraded state. The mean values of these answers are shown in Fig. 3.2d, counting the types of parts they recognized from 0 to 3. The results are 2.6, 2.32, and 1.41 for state-based, prio, and netlist respectively.

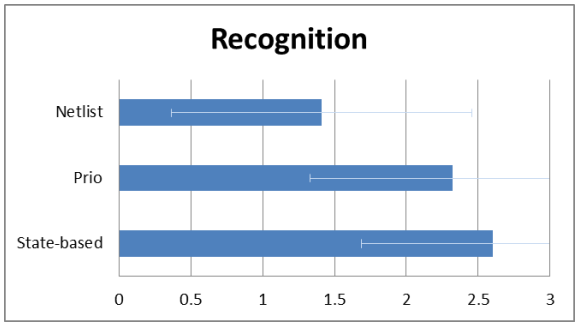


(a) Correct answers

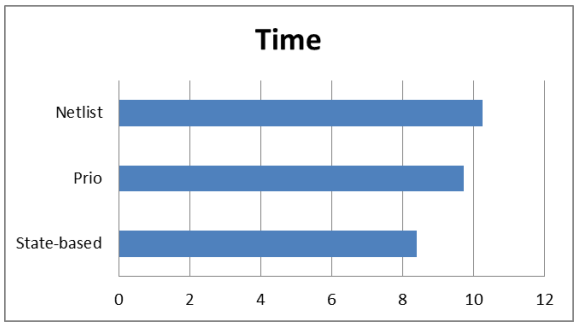


(b) Incorrect answers despite being confident

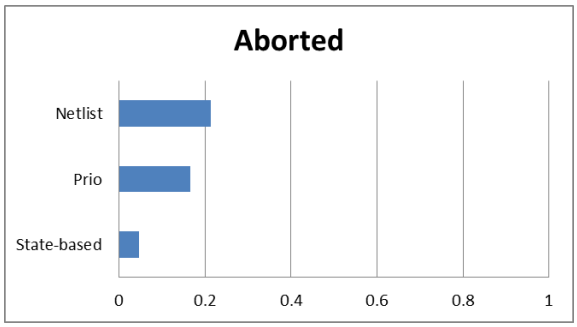
(c) Confidence rating



(d) Recognition of model elements



(e) Mean time needed



(f) Aborted prematurely

Figure 3.2.: Case-study results

Time The mean time values for the participants that gave correct answers are shown in Fig. 3.2e, which are 10:15 minutes for netlist, 09:44 minutes for prio, and 08:24 minutes for state-based. In this order of approaches, 9, 7, and 2 participants ended their try prematurely, because they could not make any sense of the generated code, which is shown in Fig. 3.2f normalized to the total number of participants.

Freeform Feedback All participants that gave feedback to the netlist approach state that it is not readable and not made to be read by humans. The priority approach was criticized since one has to read all code leading to the degraded state to find it. Moreover, the presence of many conditionals in this approach impairs the readability. The state-based approach was seen as the most readable. However, one has to focus on the code

relevant for the task, because of the increased code size. The state-based approach result in 1271 lines of generated code compared to 390 and 316 for priority and netlist (without header). The long variable names were also criticized: They proved a challenge for one participant with dyslexia, because they often have seemingly redundant names as a result of the transformation, such as "... regionoperation_stateoperation_regionoperation... ".

Results Overall, the results confirm the trends from the first case-study [SMvH18]. The code generated by the state-based approach is more readable and mappable to the original model. This facilitates manual verification of the generated code, increasing the rate of issues found in less time. While it is arguably not a big surprise that manually verifying netlists is hard, the state-based approach also scores better in this field compared to the priority-based approach despite the fact that both generate structures that resemble the original model. The priority-based approach uses nested macros for concurrent regions and jumps to labels to model transitions and states. The state-based approach follows the object-oriented state pattern, but creates dedicated functions for regions and states, which mimic the hierarchy of an SCChart. The current state is stored in a context object and is updated at state transitions. Also, the priority-based approach sometimes gave a false sense of confidence, which was less of a problem with the state-based approach.

4. Related Work

Pintér, Majzik [PM03], and Samek [Sam02] give overviews over common Statecharts compilation techniques, such as implementation of states via *nested switch* statements, *action-state tables* that store pointers to event functions, and approaches following the *state design pattern* for object-oriented languages [GHJV95], in which descendants of a common interface implement concrete states. SCCharts' state-based compilation approach is a variation of the state design pattern customized to the hierarchical state-region structure of the SCCharts meta-model. Niaz et al. present a code generation approach for UML Statecharts. Object-oriented Java code is generated by the JCode code generation system [NT05]. Niaz et al. argue that their approach produces maintainable code since generated Java objects can be mapped to states. However, they did not evaluate whether this is an improvement compared to other code generators.

For synchronous languages, Potop-Butucaru et al. [PBEB07] as well as Edwards and Zeng [EZ07] explain various different compilation approaches for compiling Esterel and similar languages. The approaches cover translations into classical finite state machines (FSMs), netlists, and graph code, a control-flow expansion developed by Potop-Butucaru [PB02], dynamic instruction lists, and code for virtual machine execution.

Amende [Ame10] describes translation rules for SyncCharts to SC [vH09], a lightweight synchronous macro-based extension to C. While the structural translation of a SyncCharts program is done straight-forwardly, concurrency is solved by assigning static priorities to the target code. Effectively, a light-weight scheduler implemented by the macros executes the program according to the pre-calculated priorities.

Biernacki et al. [BCHP08] presented a formalized modular code generation for synchronous dataflow languages, which is the foundation of the certified compiler used in SCADE. After type and clock checking, the program is translated into an minimal object-oriented intermediate language. This intermediate representation can be transformed easily into common general purpose languages, such as C or Java. A minimal, certified reference compiler was written in OCaml and COQ.

While all code generated for synchronous languages is founded in rigorous semantics, which make automatic code generation more trustworthy in the first place, readability and **manual** verification of the source code was not of primary concern in the previously mentioned approaches. Most code generation approaches transform input models into intermediate languages, which eases the downstream compilation, but code readability and therefore manual code verification suffers.

5. Conclusion

This study continues our works towards readability and verifiability of the state-based compilation approach [SMvH18]. The previous paper investigated how well SCCharts could be reverse engineered from their generated code. As proposed in the future work section of the initial contribution, we here focused on verification of SCCharts given the original diagram and the erroneous generated code. The results, as presented in Sec. 3.2, confirm the trend from the first study. The code generated by the state-based approach is more readable and mappable to the original model, increasing efficiency of manual verification in both, correctness and time needed.

As future work, readability can be improved even more by translating critical SCCharts elements, such as preemption, directly into state-based code. While this sacrifices some of the modularity of the general KIELER compilation approach, we expect readability and verifiability to increase even more. The main task here is to find a good balance between extra transformation effort needed to increase readability and reuse of modular compiler transformations that make compact SCCharts feasible in the first place.

Bibliography

[Abr96] Jean-Raymond Abrial. Steam-boiler control specification problem. In *Formal Methods for Industrial Applications*, pages 500–509. Springer, 1996.

[Ame10] Torsten Amende. Synthese von SC-Code aus SyncCharts. Diploma thesis, Kiel University, Department of Computer Science, May 2010. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tam-dt.pdf.

[BCHP08] Darek Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *Proceedings of the ACM SIGPLAN/SIGBED 2008 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM.

[BW96] Robert Büssow and Matthias Weber. A steam-boiler control specification with statecharts and z. In *Formal methods for industrial applications*, pages 109–128. Springer, 1996.

[EZ07] Stephen A. Edwards and Jia Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Mot17] Christian Motika. *SCCharts—Language and Interactive Incremental Implementation*. Number 2017/2 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.

[MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling SCCharts—A case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 461–480, Corfu, Greece, October 2014.

[NT05] Iftikhar Azim Niaz and Jiro Tanaka. An object-oriented approach to generate java code from uml statecharts. *International Journal of Computer & Information Science*, 6(2):83–98, 2005.

[PB02] Dumitru Potop-Butucaru. *Optimizations for faster simulation of Esterel programs*. PhD thesis, Ecole des Mines de Paris, France, November 2002.

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[PM03] Gergely Pintér and István Majzik. Program Code Generation based on UML Statechart Models. *Periodica Polytechnica*, pages 187–204, 2003.

[Rie13] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. Taylor & Francis Inc, 2013.

[Sam02] Miro Samek. *Practical Statecharts in C/C++*. CMP Books, 2002.

[SMvH18] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. Synthesizing manually verifiable code for statecharts. In *Proc. Reactive and Event-based Languages & Systems (REBLS '18), Workshop at the ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Boston, MA, USA, November 2018.

[SSRvH18a] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Guidance in model-based compilations. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '18), Doctoral Symposium*, Electronic Communications of the EASST, Limassol, Cyprus, November 2018. in press.

[SSRvH18b] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. Towards interactive compilation models. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, volume 11244 of *LNCS*, pages 246–260, Limassol, Cyprus, November 2018. Springer.

[vH09] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT '09)*, pages 225–234, Grenoble, France, October 2009. ACM.

[vHDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, pages 372–383, Edinburgh, UK, June 2014. ACM.

A. Steam Boiler Model Handout

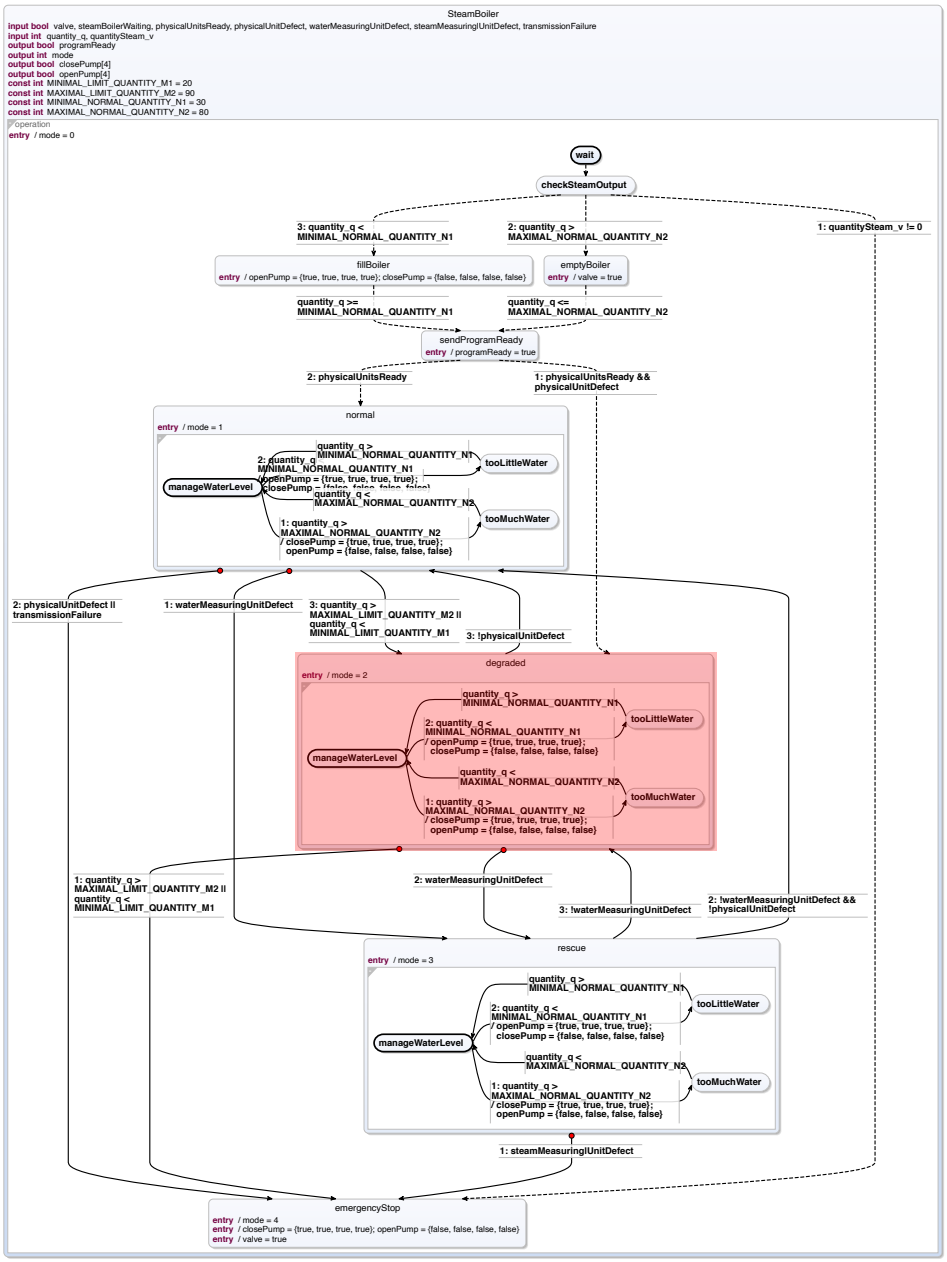


Figure A.1.: Steam boiler with highlighted degraded state