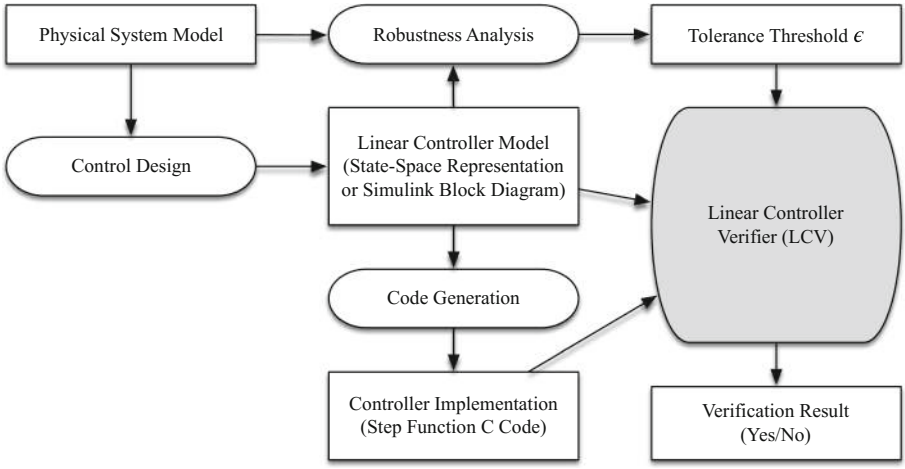


Fig. 1. LCV in the model-based development process.

diagram (i.e., controller model) and the C code (i.e., controller implementation). LCV checks the input-output equivalence relation between the two LTI models by similarity checking. The contribution of this work compared to the previous work [24] is as follows: As controller specifications are often given in the form of block diagrams, LCV extends the preliminary prototype [24] to take not only the state-space representation of an LTI system but also the Simulink block diagram as an input specification model. As a result, a real-world case study, where the controller specification of a quadrotor called Erle-Copter [11] is given as a Simulink block diagram, was conducted using LCV and demonstrated in this paper. In the case study with a proportional-integral-derivative (PID) controller, we demonstrate that LCV successfully detects a known (reproduced) bug of Embedded Coder as well as an unknown bug of Salsa [8], a code transformation method/tool for numerical accuracy.[2] Moreover, LCV has been enhanced in many ways such as improving in scalability, supporting fully automatic verification procedures, providing informative output messages and handling customized user inputs.

2 Related Work

To ensure the correctness of the controller implementation against the controller model, a typically used method in practice is equivalence testing (or back-to-back testing) [6,7,28] which compares the outputs of the executable model and code for the common input sequence. The limitation of this testing-based method is that it does not provide a thorough verification. Static analysis-based approaches [3,12,14] have been used to analyze the controller code,

[2] This bug has been confirmed by the author of the tool.

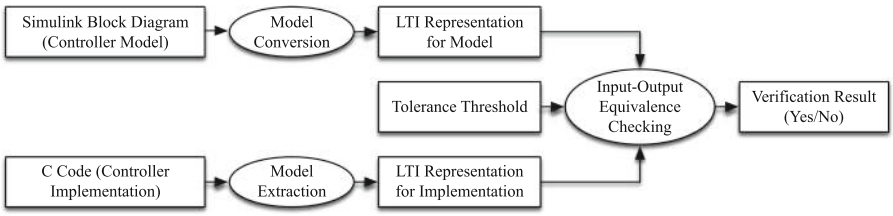


Fig. 2. The verification flow of LCV.

but focuses on checking common properties such as numerical stability, the absence of buffer overflow or arithmetic exceptions rather than verifying the code against the model. The work of [17,27] proposes translation validation techniques for Simulink diagrams and the generated codes. The verification relies on the structure of the block diagram and the code, thus being sensitive to the controller state while our method verifies code against the model from the input-output perspective, not being sensitive to the controller state. Due to optimization and transformation during a code generation process, a generated code which is correct, may have a different state representation than the model's. In this case, our method can verify that the code is correct w.r.t. the model, but the state-sensitive methods [17,27] cannot. [13,16,37,38] present a control software verification approach based on the concept of proof-carrying code. In their approach, the code annotation based on the Lyapunov function and its proof are produced at the time of code generation. The annotation asserts control theory related properties such as stability and convergence, but not equivalence between the controller specifications and the implementations. In addition, their approach requires the internal knowledge and control of the code generator to use, and may not be applicable to the off-the-shelf black-box code generators. The work of [19,24,25] presents methods to verify controller implementations against LTI models, but does not relate the block diagram models with the implementation code.

3 Verification Flow of Linear Controller Verifier

The goal of LCV is to verify linear controller software. Controllers are generally specified as a function that, given the current state of the controller and a set of input sensor values, computes control output that is sent to the system actuators and the new state of the controller. In this work, we focus on linear-time invariant (LTI) controllers [26], since these are the most commonly used controllers in control systems. In software, controllers are implemented as a subroutine (or a function in the C language). This function is known as the *step function* (see [23] for an example). The step function is invoked by the control system periodically, or upon arrival of new sensor data (i.e., measurements).

This section describes the verification flow (shown in Fig. 2) and the implementation details of LCV. LCV takes as input a Simulink block diagram

(i.e., controller model), a C code (i.e., controller implementation) and a tolerance threshold as a real number. In addition, LCV requires the following information to be given as input: the name of the step function and the interface of the step function. LCV assumes that the step function interfaces through the given input and output global variables. In other words, the input and output variables are declared in the global scope, and the input variables are written (or set) before the execution (or entrance) of the step function. Likewise, the output variables are read (or used) after the execution (or exit) of the step function.[3] Thus, the step function interface comprises the list of input (and output) variables of the step function in the same order of the corresponding input (and output) ports of the block diagram model. Since LCV verifies controllers from the input-output perspective, LCV does not require any state related information (i.e., the dimension of the controller state, or the list of state variables of the step function). Instead, LCV automatically obtains such information about the controller state from the analysis of the input C code and the input Simulink block diagram.

A restriction on this work is that LCV only focuses on verifying linear controller software. Thus, the scope of inputs of LCV is limited as follows: the input C program is limited to be a step function that only has a deterministic and finite execution path for a symbolic input, which is often found to be true for many embedded linear controllers. Moreover, the input Simulink block diagram is limited to be essentially an LTI system model (i.e., satisfying the superposition property). The block diagram that LCV can handle may include basic blocks (e.g., constant block, gain block, sum block), subsystem blocks (i.e., hierarchy) and series/parallel/feedback connections of those blocks. Extending LCV to verify a broader class of controllers is an avenue for future work.

The key idea in the verification flow (shown in Fig. 2) is that LCV represents both the Simulink block diagram and the C code in the same form of mathematical representation (i.e., the state space representation of an LTI system), and compares the two LTI models from the input-output perspective. Thus, the first step of the verification is to transform the Simulink block diagram into a state space representation of an LTI system, which is defined as follows:

$$\begin{aligned} \mathbf{z}_{k+1} &= \mathbf{A}\mathbf{z}_k + \mathbf{B}\mathbf{u}_k \\ \mathbf{y}_k &= \mathbf{C}\mathbf{z}_k + \mathbf{D}\mathbf{u}_k. \end{aligned} \tag{1}$$

where $\mathbf{u}_k$, $\mathbf{y}_k$ and $\mathbf{z}_k$ are the input vector, the output vector and the state vector at time k respectively. The matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{D}$ are controller parameters. We convert the Simulink block diagram into the LTI model employing the 'exact linearization' (or block-by-block linearization) feature of Simulink Control Design [33] which is implemented in the built-in Matlab function `linearize`. In this step, each individual block is linearized first and then combined together with others to produce the overall block diagram's LTI model.

[3] This convention is used by Embedded Coder, a code generation toolbox for Matlab/Simulink.

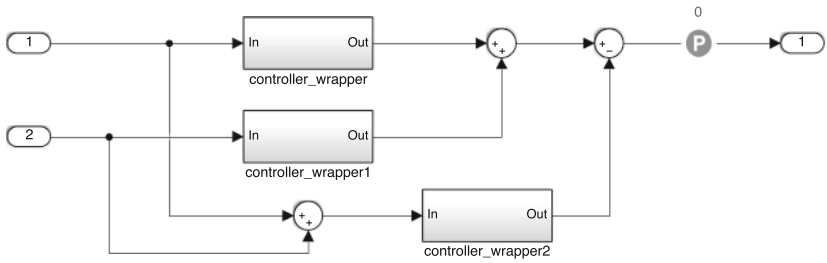


Fig. 3. The simulink block diagram for checking the additivity of the controller

This step assumes that the block diagram represents a linear controller model. A systematic procedure[4] can remove this assumption: one can check whether a given Simulink block diagram is linear (i.e., both additive and homogeneous) using Simulink Design Verifier [34], a model checker for Simulink. For example, to check if a controller block in Simulink is additive or not, as shown in Fig. 3, one can create two additional duplicates of the controller block, generate two different input sequences, and exhaustively check if the output of the controller in response to the sum of two inputs is equal to the sum of two outputs of the controllers in response the two inputs respectively. In Fig. 3, `controller_wrapper` wraps the actual controller under test, and internally performs multiplexing and demultiplexing to handle the multiple inputs and outputs of the controller. Simulink Design Verifier serves checking if this holds for all possible input sequences. However, a limitation of the current version of Simulink Design Verifier is that it does not support all Simulink blocks and does not properly handle non-linear cases. In these cases, alternatively, one can validate the linearity of controllers using simulation-based testing instead of model checking, which can be systematically done by Simulink Test [35]. This method is not limited by any types of Simulink blocks, and can effectively disprove the linearity of controllers for non-linear cases. However, this alternative method using Simulink Test may not be as rigorous as the model-checking based method using Simulink Design Verifier because not all possible input cases are considered.

The next step in the LCV's verification flow is to extract the LTI model from the controller implementation C code. The idea behind this step is to exploit the fact that linear controller codes (i.e., step function) used for embedded systems generally have simple control flows for the sake of deterministic real-time behaviors (e.g., fixed upper bound of loops). Thus, the semantics of such linear controller codes can be represented as a set of mathematical functions that are loop-free, which can be further transformed into the form of an LTI model. To do this specifically, LCV uses the symbolic execution technique which is capable of

[4] This procedure is currently not implemented in LCV because the required tools such as Simulink Design Verifier and Simulink Test mostly provide their features through GUIs rather than APIs. Thus, this procedure will be implemented in the future work once such APIs are available. Until then, this procedure can be performed manually.

identifying the computation of the step function (i.e., C function which implements the controller). By the computation, we mean the big-step transition relation on global states between before and after the execution of the step function. The big-step transition relation is represented as symbolic formulas that describe how the global variables change as the effect of the step function execution. The symbolic formulas associate each global variable representing the controller's state and output with the symbolic expression to be newly assigned to the global variable, where the symbolic expression consists of the old values of the global variables representing the controller's state and input. Then, LCV transforms the set of equations (i.e., symbolic formulas) that represent the transition relation into a form of matrix equation, from which an LTI model for the controller implementation is extracted [24]. LCV employs the off-the-shelf symbolic execution tool PathCrawler [39], which outputs the symbolic execution paths and the path conditions of a given C program in an extensible markup language (XML) file format.

Finally, LCV performs the input-output equivalence checking between the LTI model obtained from the block diagram and the LTI model extracted from the C code implementation. To do this, we employ the notion of similarity transformation [26], which implies that two minimal LTI models $\mathbf{\Sigma}(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D})$ and $\hat{\mathbf{\Sigma}}(\hat{\mathbf{A}}, \hat{\mathbf{B}}, \hat{\mathbf{C}}, \hat{\mathbf{D}})$ are input-output equivalent if and only if they are *similar* to each other, meaning that there exists a non-singular matrix $\mathbf{T}$ such that

$$\hat{\mathbf{A}} = \mathbf{T}\mathbf{A}\mathbf{T}^{-1}, \qquad \hat{\mathbf{B}} = \mathbf{T}\mathbf{B}, \qquad \hat{\mathbf{C}} = \mathbf{C}\mathbf{T}^{-1}, \qquad \text{and} \qquad \hat{\mathbf{D}} = \mathbf{D} \tag{2}$$

where $\mathbf{T}$ is referred to as the *similarity transformation matrix* [26].

Given the extracted LTI model (from the C Code) and the original LTI model (obtained from the Simulink block diagram), we first minimize both LTI models via Kalman Decomposition [26] (Matlab function `minreal`). Then, the input-output equivalence checking problem is reduced to the problem of finding the existence of $\mathbf{T}$ (i.e., similarity checking problem). LCV formulates the similarity checking problem as a convex optimization problem[5], and employs CVX [15], a convex optimization solver to find $\mathbf{T}$. In the formulation, the equality relation is relaxed up to a given tolerance threshold ϵ in order to tolerate the numerical errors that come from multiple sources (e.g., the controller parameters, the computation of the implementation, the verification process). We assume that the tolerance threshold ϵ is given by a control engineer as the result of robustness analysis so that the verified controller implementation preserves the certain desired properties of the original controller model (e.g., stability). ϵ is chosen to be 10^{-5} for the case study that we performed in the next section.

The output of LCV is as follows: First of all, when LCV fails to extract an LTI model from code, it tells the reason (e.g., non-deterministic execution paths for a symbolic input due to branching over a symbolic expression condition, non-linear arithmetic computation due to the use of trigonometric functions). Moreover, for the case of non-equivalent model and code, LCV provides the LTI models obtained from the Simulink block diagram model and the C code respectively, so that the user can simulate both of the models and easily find

[5] Please refer [24] for the details of the formulation.

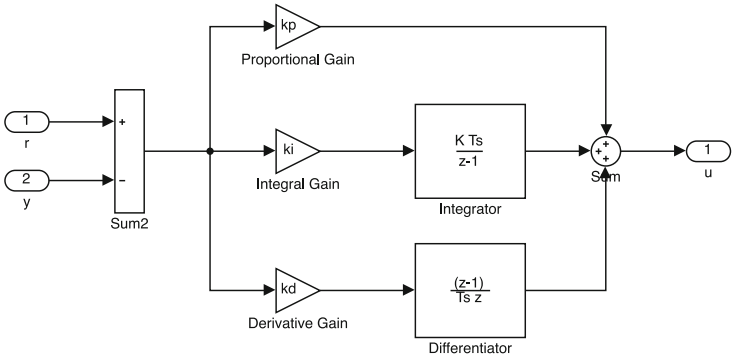


Fig. 4. The block diagram of the PID controller.

an input sequence that leads to a discrepancy between their output behaviors.[6] Finally, for the case of equivalent model and code, LCV additionally provides a similarity transformation matrix $\mathbf{T}$ between the two LTI models, which is the key evidence to prove the input-output equivalence between the model and code.

4 Evaluation

We evaluate LCV through conducting a case study using a standard PID controller and a controller used in a quadrotor. We also evaluate the scalability of LCV in the subsequent subsection.

4.1 Case Study

PID Controller. In our case study, we first consider a proportional-integral-derivative (PID) controller, which is a closed-loop feedback controller commonly used in various control systems (e.g., industrial control systems, robotics, automotive). A PID controller attempts to minimize the error value e_t over time which is defined as the difference between a reference point r_t (i.e., desired value) and a measurement value y_t (i.e., $e_t = r_t - y_t$). To do this, the PID controller adjusts a control input u_t computing the sum of the proportion term $k_p e_t$, integral term $k_i T \sum_{i=1}^{t} e_t$ and derivative term $k_d \frac{e_t - e_{t-1}}{T}$ so that

$$u_t = k_p e_t + k_i T \sum_{i=1}^{t} e_t + k_d \frac{e_t - e_{t-1}}{T}. \tag{3}$$

where k_p, k_i and k_d are gain constants for the corresponding term, and T is the sampling time. Figure 4 shows the Simulink block diagram for the PID controller, where the gain constants are defined as $k_p = 9.4514, k_i = 0.69006, k_d = 2.8454$, and the sampling period is 0.2 s.

[6] This feature to generate counterexamples will be implemented in a future version of LCV.

Table 1. Summary of the case study with the PID controller (Fig. 4) and its different versions of implementation

Impl.	Description	Buggy?	LCV output
`PID1`	Generated by Embedded Coder	No	Equivalent
`PID2`	Optimized from PID1 by Salsa (Level 1)	No	Equivalent
`PID3`	Optimized from PID1 by Salsa (Level 2)	Yes (due to a bug in Salsa)	Not equivalent
`PID3'`	Corrected from PID3 manually	No	Equivalent
`PID4`	Generated by Embedded Coder with a buggy option triggered	Yes (due to a bug in Embedded Coder)	Not equivalent

For the PID controller model, we check various different versions of implementations such as `PID1`, `PID2`, `PID3`, `PID3'` and `PID4` (summarized in Table 1). `PID1` is obtained by code generation from the model using Embedded Coder. `PID2` is obtained from `PID1` by the transformation (or optimization) of Salsa [8] to improve the numerical accuracy (using the first transformation technique (referred to as Level 1) presented in [8]). In a similar way, `PID3` is obtained by the transformation from `PID1` for an even better numerical accuracy (following the second transformation technique (referred to as Level 2) as Listing 3 in [8]). However, this transformation for `PID3` contains an unintended bug by mistake that has been confirmed by the authors of the paper (i.e., variable `s` is not computed correctly, and the integral term is redundantly added to the output), which makes `PID3` incorrect. `PID3'` is an implementation that manually corrects `PID3`. Using LCV, we can verify that `PID1`, `PID2` and `PID3'` are correct implementations, but `PID3` is not (see the verification result for `PID3` [21]).

Moreover, `PID4` is obtained by injecting a known bug of Embedded Coder into the implementation `PID1`. The bug with the ID 1658667 [29] that exists in the Embedded Coder version from 2015a through 2017b (7 consecutive versions) causes the generated code to have state variable declarations in a wrong scope. The state variables which are affected by the bug are mistakenly declared as local variables inside the step function instead of being declared as global variables. Thus, those state variables affected by the bug are unable to preserve their values throughout the consecutive step function executions. LCV can successfully detect the injected bug by identifying that the extracted model from the controller code does not match the original controller model (see the verification result for `PID4` [22]).

Quadrotor Controller. The second and more complex application in our case study is a controller of the quadrotor called Erle-Copter. The quadrotor controller controls the quadrotor to be in certain desired angles in roll, yaw and pitch. The quadrotor uses the controller software from the open source project Ardupilot [1]. Inspired by the controller software, we obtained the Simulink block diagram shown in Fig. 5. In the names of the Inport blocks, the suffix `_d` indicates the desired angle, `_d`, the measured angle, and `_rate_y`, the angular speed. Each component of the coordinate of the quadrotor is separately controlled by its own

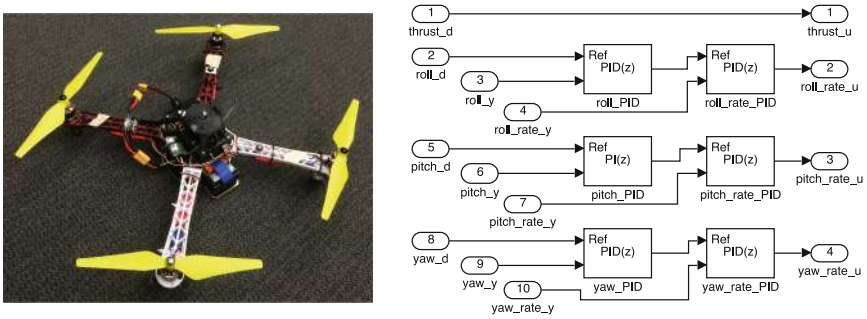


Fig. 5. Our quadrotor platform (Left). The quadrotor controller block diagram (Right).

cascade PID controller [18]. A cascade of PID controller is a sequential connection of two PID controllers such that one PID controller controls the reference point of another. In Fig. 5, there are three cascade controllers for the controls of roll, pitch and yaw. For example, for the roll control, `roll_pid` controls the angle of roll, while `roll_rate_PID` controls the rate of roll using the output of `roll_PID` as the reference point. The sampling time T of each PID controller is 2.5 ms. This model uses the built-in PID controller block of Simulink to enable the PID auto-tuning software in Matlab (i.e., `pidtune()`). The required physical quantities for controlling roll and pitch are identified by physical experiments [10]. We use Embedded Coder to generate the controller code for the model, and verify that the generated controller code correctly implements the controller model using LCV (see the verification result for the quadrotor controller [20]).

4.2 Scalability

To evaluate the scalability of LCV, we measure the running time of LCV verifying the controllers of different dimensions (i.e., the size of the LTI model). We randomly generate LTI controller models using Matlab function `drss` varying the controller dimension n from 2 to 50. The range of controller sizes was chosen based on our observation of controller systems in practice. We construct Simulink models with LTI system blocks that contain the generated LTI models, and use Embedded Coder to generate the implementations for the controllers. The running time of LCV for verifying the controllers with different dimensions is presented in Fig. 6, which shows that LCV is scalable for the realistic size of controller dimension. Compared to the previous version (or the preliminary prototype) of LCV [24], the new version of LCV has been much improved in scalability by tighter integration with the symbolic execution engine PathCrawler (i.e., in the model extraction phase, the invocation of constraint solver along with symbolic execution has been significantly reduced).

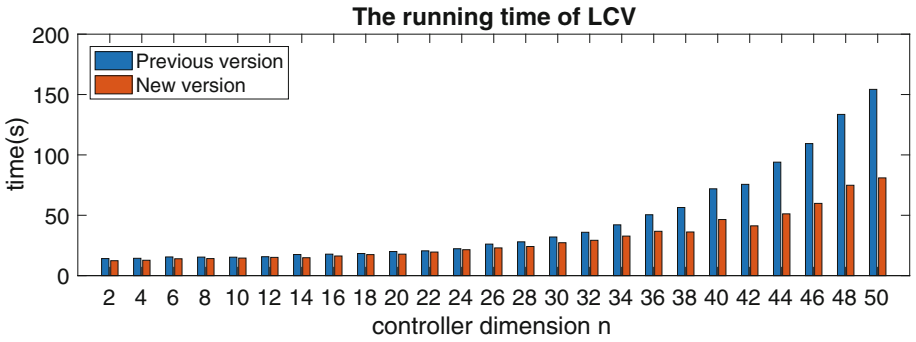


Fig. 6. The running time of LCV for verifying controllers with dimension n.

5 Conclusion

We have presented our tool LCV which verifies the equivalence between a given Simulink block diagram and a given C implementation from the input-output perspective. Through an evaluation, we have demonstrated that LCV is applicable to the verification of a real-world system's controller and scalable for the realistic controller size. Our current/future development work includes: relating the equivalence precision and the controller's performance, and handling nonlinear controllers.

Acknowledgments. This work is sponsored in part by the ONR under agreement N00014-17-1-2504, as well as the NSF CNS-1652544 grant. This research was supported in part by ONR N000141712012, Global Research Laboratory Program (2013K1A1A2A02078326) through NRF, and the DGIST Research and Development Program (CPS Global Center) funded by the Ministry of Science, ICT & Future Planning, and NSF CNS-1505799 and the Intel-NSF Partnership for Cyber-Physical Systems Security and Privacy.

References

1. Ardupilot Dev Team: Ardupilot, September 2018. http://ardupilot.org/
2. Behera, C.K., Bhaskari, D.L.: Different obfuscation techniques for code protection. Procedia Comput. Sci. **70**, 757–763 (2015)
3. Blanchet, B., et al.: A static analyzer for large safety-critical software. In: ACM SIGPLAN Notices, vol. 38, pp. 196–207. ACM (2003)
4. Cappaert, J.: Code obfuscation techniques for software protection, pp. 1–112. Katholieke Universiteit Leuven (2012)
5. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand (1997)
6. Conrad, M.: Testing-based translation validation of generated code in the context of IEC 61508. Form Methods Syst. Des. **35**(3), 389–401 (2009)

7. Conrad, M.: Verification and validation according to ISO 26262: a workflow to facilitate the development of high-integrity software. Embedded Real Time Software and Systems (ERTS2 2012) (2012)
8. Damouche, N., Martel, M., Chapoutot, A.: Transformation of a PID controller for numerical accuracy. Electron. Notes Theor. Comput. Sci. **317**, 47–54 (2015)
9. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. Int. J. Softw. Tools Technol. Transfer **19**(4), 427–448 (2017). https://doi.org/10.1007/s10009-016-0435-0
10. Derafa, L., Madani, T., Benallegue, A.: Dynamic modelling and experimental identification of four rotors helicopter parameters. In: 2006 IEEE International Conference on Industrial Technology (2006)
11. Erle Robotics: Erle-copter, September 2018. http://erlerobotics.com/blog/erle-copter/
12. Feret, J.: Static analysis of digital filters. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 33–48. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24725-8_4
13. Feron, E.: From control systems to control software. IEEE Control Syst. **30**(6), 50–71 (2010)
14. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
15. Grant, M., Boyd, S.: CVX: Matlab software for disciplined convex programming, version 2.1, March 2014. http://cvxr.com/cvx
16. Herencia-Zapana, H., et al.: PVS linear algebra libraries for verification of control software algorithms in C/ACSL. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 147–161. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_15
17. Majumdar, R., Saha, I., Ueda, K., Yazarel, H.: Compositional equivalence checking for models and code of control systems. In: 52nd Annual IEEE Conference on Decision and Control (CDC), pp. 1564–1571 (2013)
18. Michael, N., Mellinger, D., Lindsey, Q., Kumar, V.: The GRASP multiple micro-UAV test bed. IEEE Robot. Autom. Mag. **17**(3), 56–65 (2010)
19. Pajic, M., Park, J., Lee, I., Pappas, G.J., Sokolsky, O.: Automatic verification of linear controller software. In: 12th International Conference on Embedded Software (EMSOFT), pp. 217–226. IEEE Press (2015)
20. Park, J.: Erle-copter verification result. https://doi.org/10.5281/zenodo.2565035
21. Park, J.: Pid3 verification result. https://doi.org/10.5281/zenodo.2565023
22. Park, J.: Pid4 verification result. https://doi.org/10.5281/zenodo.2565030
23. Park, J.: Step function example. https://doi.org/10.5281/zenodo.44338
24. Park, J., Pajic, M., Lee, I., Sokolsky, O.: Scalable verification of linear controller software. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 662–679. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_43
25. Park, J., Pajic, M., Sokolsky, O., Lee, I.: Automatic verification of finite precision implementations of linear controllers. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 153–169. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_9
26. Rugh, W.J.: Linear System Theory. Prentice Hall, London (1996)
27. Ryabtsev, M., Strichman, O.: Translation validation: from simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 696–701. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02658-4_57

28. Stuermer, I., Conrad, M., Doerr, H., Pepper, P.: Systematic testing of model-based code generators. IEEE Trans. Software Eng. **33**(9), 622–634 (2007)
29. The Mathworks, Inc.: Bug reports for incorrect code generation. http://www.mathworks.com/support/bugreports/?product=ALL&release=R2015b&keyword=Incorrect+Code+Generation
30. The Mathworks, Inc.: Embedded coder, September 2017. https://www.mathworks.com/products/embedded-coder.html
31. The Mathworks, Inc.: Simulink, September 2018. https://www.mathworks.com/products/simulink.html
32. The Mathworks, Inc.: Simulink coder, September 2018. https://www.mathworks.com/products/simulink-coder.html
33. The Mathworks, Inc.: Simulink control design, September 2018. https://www.mathworks.com/products/simcontrol.html
34. The Mathworks, Inc.: Simulink design verifier, September 2018. https://www.mathworks.com/products/sldesignverifier.html
35. The Mathworks, Inc.: Simulink test, September 2018. https://www.mathworks.com/products/simulink-test.html
36. The Mathworks, Inc.: Stateflow, September 2018. https://www.mathworks.com/products/stateflow.html
37. Wang, T., et al.: From design to implementation: an automated, credible autocoding chain for control systems. arXiv preprint arXiv:1307.2641 (2013)
38. Wang, T.E., Ashari, A.E., Jobredeaux, R.J., Feron, E.M.: Credible autocoding of fault detection observers. In: American Control Conference (ACC), pp. 672–677 (2014)
39. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: Dal Cin, M., Kaâniche, M., Pataricza, A. (eds.) EDCC 2005. LNCS, vol. 3463, pp. 281–292. Springer, Heidelberg (2005). https://doi.org/10.1007/11408901_21

Semantic Fault Localization and Suspiciousness Ranking

Maria Christakis[1(✉)], Matthias Heizmann[2(✉)], Muhammad Numair Mansur[1(✉)], Christian Schilling[3(✉)], and Valentin Wüstholz[4(✉)]

[1] MPI-SWS, Kaiserslautern, Saarbrücken, Germany
{maria,numair}@mpi-sws.org
[2] University of Freiburg, Freiburg im Breisgau, Germany
heizmann@informatik.uni-freiburg.de
[3] IST Austria, Klosterneuburg, Austria
christian.schilling@ist.ac.at
[4] ConsenSys Diligence, Berlin, Germany
valentin.wustholz@consensys.net

Abstract. Static program analyzers are increasingly effective in checking correctness properties of programs and reporting any errors found, often in the form of error traces. However, developers still spend a significant amount of time on debugging. This involves processing long error traces in an effort to localize a bug to a relatively small part of the program and to identify its cause. In this paper, we present a technique for automated fault localization that, given a program and an error trace, efficiently narrows down the cause of the error to a few statements. These statements are then ranked in terms of their suspiciousness. Our technique relies only on the semantics of the given program and does not require any test cases or user guidance. In experiments on a set of C benchmarks, we show that our technique is effective in quickly isolating the cause of error while out-performing other state-of-the-art fault-localization techniques.

1 Introduction

In recent years, program analyzers are increasingly applied to detect errors in real-world software. When detecting an error, static (or dynamic) analyzers often present the user with an error trace (or a failing test case), which shows how an assertion can be violated. Specifically, an error trace refers to a sequence of statements through the program that leads to the error. The user then needs to process the error trace, which is often long for large programs, in order to localize the problem to a manageable number of statements and identify its actual cause. Therefore, despite the effectiveness of static program analyzers in detecting errors and generating error traces, users still spend a significant amount of time on debugging.

Our Approach. To alleviate this situation, we present a technique for automated fault localization, which significantly reduces the number of statements

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 226–243, 2019.
https://doi.org/10.1007/978-3-030-17462-0_13

that might be responsible for a particular error. Our technique takes as input a program and an error trace, generated by a static analyzer, and determines which statements along the error trace are potential causes of the error. We identify the potential causes of an error by checking, for each statement along the error trace, whether there exists a local fix such that the trace verifies. We call this technique *semantic fault localization* because it exclusively relies on the semantics of the given program, without for instance requiring any test cases or guidance from the user.

Although there is existing work that also relies on program semantics for fault localization, our technique is the first to semantically rank the possible error causes in terms of suspiciousness. On a high level, we compute a *suspiciousness score* for a statement by taking into account how much code would become unreachable if we were to apply a local fix to the statement. Specifically, suspiciousness is inversely proportional to the amount of unreachable code. The key insight is that developers do not intend to write unreachable code, and thus the cause of the error is more likely to be a statement that, when fixed, renders fewer parts of the program unreachable.

Our experimental evaluation compares our technique to six fault-localization approaches from the literature on the widely-used TCAS benchmarks (of the Siemens test suite [19]). We show that in 30 out of 40 benchmarks, our technique narrows down the cause of the error more than any of the other approaches and is able to pin-point the faulty statement in 14 benchmarks. In addition, we evaluate our technique on several seeded bugs in SV-COMP benchmarks [5].

Contributions. We make the following contributions:

- We present an automated fault-localization technique that is able to quickly narrow down the error cause to a small number of suspicious statements.
- We describe an effective ranking mechanism for the suspicious statements.
- We implement this technique in a tool architecture for localizing and ranking suspicious statements along error traces reported by the Ultimate Automizer [16] software model checker.
- We evaluate the effectiveness of our technique on 51 benchmarks.

2 Guided Tour

This section uses a motivating example to give an overview of our technique for semantic fault localization and suspiciousness ranking.

Example. Let us consider the simple program on the left of Fig. 1. The statement on line 2 denotes that `y` is assigned a non-deterministic value, denoted by $\star$. The conditional on line 3 has a non-deterministic predicate, but in combination with the `assume` statements (lines 4 and 7), it is equivalent to a conditional of the following form:

```
if (y < 3) { x := 0; } else { x := 1; }
```

```
1  x := 2;
2  y := ⋆;
3  if (⋆) {
4    assume y < 3;
5    x := 0;
6  } else {
7    assume 3 ≤ y;
8    x := 1;
9  }
10 assert 0 < x;
```

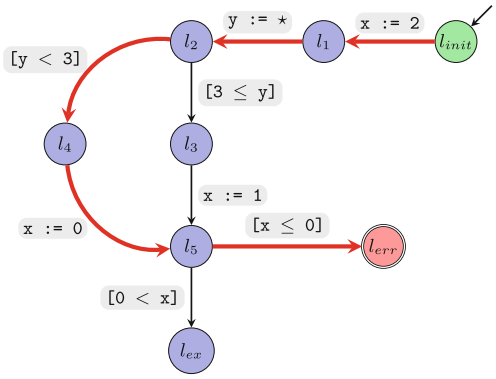


Fig. 1. The running example (left) and its control-flow graph (right).

The assertion on line 10 fails when program execution takes the `then` branch of the `if` statement, and thus, it cannot be verified by a (sound) static analyzer. The error trace that is generated by a static analyzer for this program is

`x := 2; y := ⋆; assume y < 3; x := 0; assume x ≤ 0; assert false`

and we mark it in bold on the control-flow graph of the program, which is shown on the right of Fig. 1. Statement `assume x ≤ 0` indicates that the error trace takes the failing branch of the assertion on line 10 of the program. The `assert false` statement denotes that this trace through the program results in an error.

Fault Localization. From such an error trace, our technique is able to determine a set of suspicious assignment statements, which we call *trace-aberrant statements*. Intuitively, these are statements along the error trace for which there exists a *local fix* that makes the trace verify. An assignment statement has a local fix if there exists an expression that may replace the right-hand side of the assignment such that the error becomes unreachable along this error trace. (In Sect. 5, we explain how our technique is able to identify other suspicious statements, apart from assignments.)

For example, statement `x := 0` of the error trace is trace-aberrant because there exists a value that may be assigned to variable `x` such that the trace verifies. In particular, when `x` is assigned a positive value, `assume x ≤ 0` terminates execution before reaching the error. Statement `y := ⋆` is trace-aberrant for similar reasons. These are all the trace-aberrant statements along this error trace. For instance, `x := 2` is not trace-aberrant because the value of `x` is over-written by the second assignment to the variable, thus making the error reachable along this trace, regardless of the initial value of `x`.

Suspiciousness Ranking. So far, we have seen that, when given the above error trace and the program of Fig. 1, semantic fault localization detects two trace-aberrant statements, `y := ⋆` and `x := 0`. Since for real programs there can be

many trace-aberrant statements, our technique computes a semantic suspiciousness score for each of them. Specifically, the computed score is inversely proportional to how much code would become unreachable if we applied a local fix to the statement. This is because developers do not intentionally write unreachable code. Therefore, when they make a mistake, they are more likely to fix a statement that renders fewer parts of the program unreachable relatively to other suspicious statements.

For example, if we were to fix statement `x := 0` as discussed above (that is, by assigning a positive value to `x`), no code would become unreachable. As a result, this statement is assigned the highest suspiciousness score. On the other hand, if we were to fix statement `y := ⋆` such that $3 \leq y$ holds and the trace verifies, the `then` branch of the `if` statement would become unreachable. Consequently, `y := ⋆` is assigned a lower suspiciousness score than `x := 0`.

As we show in Sect. 6, this ranking mechanism is very effective at narrowing down the cause of an error to only a few lines in the program.

Program $\mathcal{P} ::= s$

Statement $s ::= s_1; s_2 \mid v := \star \mid v := e \mid \texttt{assert}\ p \mid \texttt{assume}\ p \mid \texttt{if (}\star\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}} \mid \texttt{while (}\star\texttt{) \{}s\texttt{\}}$

Expression $e ::= v \mid c \mid e_1 \oplus e_2 \quad (\oplus \in \{+, -, \times\})$

Predicate $p ::= e_1 \oslash e_2 \quad (\oslash \in \{<, >, \leq, \geq, =\})$

Fig. 2. A simple programming language.

3 Semantic Fault Localization

As mentioned in the previous section, our technique consists of two steps, where we determine (trace-)aberrant statements in the first step and compute their suspiciousness ranks in the next one.

3.1 Programming Language

To precisely describe our technique, we introduce a small programming language, shown in Fig. 2. As shown in the figure, a program consists of a statement, and statements include sequencing, assignments, assertions, assumptions, conditionals, and loops. Observe that conditionals and loops have non-deterministic predicates, but note that, in combination with assumptions, they can express any conditional or loop with a predicate p. To simplify the discussion, we do not introduce additional constructs for procedure definitions and calls.

We assume that program execution terminates as soon as an assertion or assumption violation in encountered (that is, when the corresponding predicate evaluates to false). For simplicity, we also assume that our technique is applied to one failing assertion at a time.

3.2 Trace-Aberrant Statements

Recall from the previous section that trace-aberrant statements are assignments along the error trace for which there exists a local fix that makes the trace verify (that is, the error becomes unreachable along this trace):

Definition 1 (Trace aberrance). *Let τ be a feasible error trace and s an assignment statement of the form $v := \star$ or $v := e$ along τ. Statement s is* trace-aberrant *iff there exists an expression e' that may be assigned to variable v such that the trace verifies.*

To determine which assignments along an error trace are trace-aberrant, we first compute, in the post-state of each assignment, the weakest condition that ensures that the trace verifies. We, therefore, define a predicate transformer *WP* such that, if $WP(S, Q)$ holds in a state along the error trace, then the error is unreachable and Q holds after executing statement S. The definition of this weakest-precondition transformer is standard [9] for all statements that may appear in an error trace:

- $WP(s_1; s_2, Q) \equiv WP(s_1, WP(s_2, Q))$
- $WP(v := \star, Q) \equiv \forall v'.Q[v := v']$, where $v' \notin \mathit{freeVars}(Q)$
- $WP(v := e, Q) \equiv Q[v := e]$
- $WP(\texttt{assert false}, Q) \equiv \mathit{false}$
- $WP(\texttt{assume } p, Q) \equiv p \Rightarrow Q$

In the weakest precondition of the non-deterministic assignment, v' is fresh in Q and $Q[v := v']$ denotes the substitution of v by v' in Q.

To illustrate, we compute this condition in the post-state of each assignment along the error trace of Sect. 2. Weakest precondition

$$WP(\texttt{assume x} \leq \texttt{0;assert false}, \mathit{true}) \equiv 0 < \texttt{x}$$

should hold in the pre-state of statement `assume x` $\leq$ `0`, and thus in the post-state of assignment `x := 0`, for the trace to verify. Similarly, 3 $\leq$ `y` and *false* should hold in the post-state of assignments `y = ⋆` and `x := 2`, respectively. Note that condition *false* indicates that the error is always reachable after assignment `x := 2`.

Second, we compute, in the pre-state of each assignment along the error trace, the strongest condition that holds when executing the error trace until that state. We define a predicate transformer *SP* such that condition $SP(P, S)$ describes the post-state of statement S for an execution of S that starts from an initial state satisfying P. The definition of this strongest-postcondition transformer is also standard [10] for all statements that may appear in an error trace:

- $SP(P, s_1; s_2) \equiv SP(SP(P, s_1), s_2)$
- $SP(P, v := \star) \equiv \exists v'.P[v := v']$, where $v' \notin \mathit{freeVars}(P)$
- $SP(P, v := e) \equiv \exists v'.P[v := v'] \wedge v = e[v := v']$, where $v' \notin \mathit{freeVars}(P) \cup \mathit{freeVars}(e)$

- $SP(P, \texttt{assert false}) \equiv \mathit{false}$
- $SP(P, \texttt{assume } p) \equiv P \wedge p$

In the strongest postcondition of the assignment statements, v' represents the previous value of v.

For example, strongest postcondition

$$SP(\mathit{true}, \texttt{x := 2}) \equiv \texttt{x} = 2$$

holds in the post-state of assignment `x := 2`, and therefore in the pre-state of `y := ⋆`. Similarly, the strongest such conditions in the pre-state of assignments `x := 2` and `x := 0` along the error trace are *true* and $\texttt{x} = 2 \wedge \texttt{y} < 3$, respectively.

Third, our technique determines if an assignment a (of the form $v := \star$ or $v := e$) along the error trace is trace-aberrant by checking whether the Hoare triple [17] $\{\phi\}\ v := \star\ \{\psi\}$ is valid. Here, ϕ denotes the strongest postcondition in the pre-state of assignment a, v the left-hand side of a, and ψ the negation of the weakest precondition in the post-state of a. If this Hoare triple is *invalid*, then assignment statement a is trace-aberrant, otherwise it is not.

Intuitively, the validity of the Hoare triple implies that, when starting from the pre-state of a, the error is always reachable no matter which value is assigned to v. In other words, there is no local fix for statement a that would make the trace verify. Consequently, assignment a is not trace-aberrant since it cannot possibly be the cause of the error. As an example, consider statement `x := 2`. For this assignment, our technique checks the validity of the Hoare triple $\{\mathit{true}\}$ `x := ⋆` $\{\mathit{true}\}$. Since any value for `x` satisfies the true postcondition, assignment `x := 2` is not trace-aberrant.

If, however, the Hoare triple is invalid, there exists a value for variable v such that the weakest precondition in the post-state of a holds. This means that there is a local fix for a that makes the error unreachable. As a result, statement a is found to be trace-aberrant. For instance, for statement `x := 0`, we construct the following Hoare triple: $\{\texttt{x} = 2 \wedge \texttt{y} < 3\}$ `x := ⋆` $\{\texttt{x} \leq 0\}$. This Hoare triple is invalid because there are values that may be assigned to `x` such that $\texttt{x} \leq 0$ does not hold in the post-state. Assignment `x := 0` is, therefore, trace-aberrant. Similarly, for `y := ⋆`, the Hoare triple $\{\texttt{x} = 2\}$ `y := ⋆` $\{\texttt{y} < 3\}$ is invalid.

3.3 Program-Aberrant Statements

We now define *program-aberrant statements*; these are assignments for which there exists a local fix that makes *every* trace through them verify:

Definition 2 (Program aberrance). *Let τ be a feasible error trace and s an assignment statement of the form $v := \star$ or $v := e$ along τ. Statement s is* program-aberrant *iff there exists an expression e' that may be assigned to variable v such that all traces through s verify.*

Based on the above definition, the trace-aberrant assignments in the program of Fig. 1 are also program-aberrant. This is because there is only one error trace through these statements.

As another example, let us replace the assignment on line 8 of the program by `x := -1`. In this modified program, the assertion on line 10 fails when program execution takes either branch of the `if` statement. Now, assume that a static analyzer, which clearly fails to verify the assertion, generates the same error trace that we described in Sect. 2 (for the `then` branch of the `if` statement). Like before, our technique determines that statements `y := ⋆` and `x := 0` along this error trace are trace-aberrant. However, although there is still a single error trace through statement `x := 0`, there are now two error traces through `y := ⋆`, one for each branch of the conditional. We, therefore, know that `x := 0` is program-aberrant, but it is unclear whether assignment `y := ⋆` is.

To determine which trace-aberrant assignments along an error trace are also program-aberrant, one would need to check if there exists a local fix for these statements such that all traces through them verify. Recall that there exists a fix for a trace-aberrant assignment if there exists a right-hand side that satisfies the weakest precondition in the post-state of the assignment *along the error trace*. Therefore, checking the existence of a local fix for a program-aberrant statement involves computing the weakest precondition in the post-state of the statement *in the program*, which amounts to program verification and is undecidable.

Identifying which trace-aberrant statements are also program-aberrant is desirable since these are precisely the statements that can be fixed for the program to verify. However, determining these statements is difficult for the reasons indicated above. Instead, our technique uses the previously-computed weakest preconditions to decide which trace-aberrant assignments *must* also be program-aberrant, in other words, it can under-approximate the set of program-aberrant statements. In our experiments, we find that many trace-aberrant statements are *must-program-aberrant*.

To compute the must-program-aberrant statements, our technique first identifies the trace-aberrant ones, for instance, `y := ⋆` and `x := 0` for the modified program. In the following, we refer to the corresponding error trace as ϵ.

As a second step, our technique checks whether all traces through the trace-aberrant assignments verify with the most permissive local fix that makes ϵ verify. To achieve this, we instrument the faulty program as follows. We replace a trace-aberrant statement a_i of the form $v := e$ by a non-deterministic assignment $v := \star$ with the same left-hand side v. Our technique then introduces an `assume` statement right after the non-deterministic assignment. The predicate of the assumption corresponds to the weakest precondition that is computed in the post-state of the assignment along error trace ϵ[1]. We apply this instrumentation separately for each trace-aberrant statement a_i, where $i = 0, \ldots, n$, and we refer to the instrumented program that corresponds to trace-aberrant statement a_i as P_{a_i}. (Note that our technique uses this instrumentation for ranking aberrant statements in terms of suspiciousness, as we explain in Sect. 4.) Once we obtain a program P_{a_i}, we instrument it further to add a flag that allows the error to

[1] Any universal quantifier appearing in the weakest precondition can be expressed within the language of Fig. 2 by using non-deterministic assignments.

manifest itself only for traces through statement a_i (as per Definition 2). We denote each of these programs by $P^{\dagger}_{a_i}$.

For example, the light gray boxes on the right show the instrumentation for checking whether statement `x := 0` of the modified program is program-aberrant (the darker boxes should be ignored). Lines 8–9 constitute the instrumentation that generates program $P_{\texttt{x := 0}}$. As explained in Sect. 3.2, when the computed weakest precondition holds on line 9, it implies that trace ϵ verifies. Consequently, this instrumentation represents a hypothetical local fix for assignment `x := 0`. Lines 1, 10, and 15 block any program execution that does not go through statement `x := 0`. As a result, the assertion may fail only due to failing executions through this statement. Similarly, when considering the dark gray boxes in addition to lines 1 and 15 (and ignoring all other light boxes), we obtain $P^{\dagger}_{\texttt{y := }\star}$. Line 4 alone constitutes the instrumentation that generates program $P_{\texttt{y := }\star}$.

```
1  flag := 0;
2  x := 2;
3  y := ⋆;
4  assume 3 ≤ y;
5  flag := 1;
6  if (⋆)
7    assume y < 3;
8    x := ⋆;
9    assume 0 < x;
10   flag := 1;
11 } else {
12   assume 3 ≤ y;
13   x := -1;
14 }
15 assume flag = 1;
16 assert 0 < x;
```

Third, our technique runs the static analyzer on each of the n instrumented programs $P^{\dagger}_{a_i}$. If the analyzer does not generate a new error trace, then statement a_i must be program-aberrant, otherwise we do not know. For instance, when running the static analyzer on $P^{\dagger}_{\texttt{x := 0}}$ from above, no error is detected. Statement `x := 0` is, therefore, program-aberrant. However, an error trace is reported for program $P^{\dagger}_{\texttt{y := }\star}$ (through the `else` branch of the conditional). As a result, our technique cannot determine whether `y := ⋆` is program-aberrant. Notice, however, that this statement is, in fact, not program-aberrant because there is no fix that we can apply to it such that both traces verify.

3.4 *k*-Aberrance

So far, we have focused on (trace- or program-) aberrant statements that may be fixed to single-handedly make one or more error traces verify. The notion of aberrance, however, may be generalized to sets of statements that make the corresponding error traces verify only when fixed together:

Definition 3 (k-Trace aberrance). *Let τ be a feasible error trace and $\bar{s}$ a set of assignment statements of the form $v := \star$ or $v := e$ along τ. Statements $\bar{s}$ are $|\bar{s}|$-*trace-aberrant*, where $|\bar{s}|$ is the cardinality of $\bar{s}$, iff there exist local fixes for all statements in $\bar{s}$ such that trace τ verifies.*

Definition 4 (k-Program aberrance). *Let $\bar{\tau}$ be the set of all feasible error traces through any assignment statement s in a set $\bar{s}$. Each statement s is of the form $v := \star$ or $v := e$ along an error trace τ in $\bar{\tau}$. Statements $\bar{s}$ are $|\bar{s}|$-*program-aberrant*, where $|\bar{s}|$ is the cardinality of $\bar{s}$, iff there exist local fixes for all statements in $\bar{s}$ such that all traces $\bar{\tau}$ verify.*

For example, consider the modified version of the program in Fig. 1 that we discussed above. Assignments `x := 0` and `x := -1` are 2-program-aberrant because their right-hand side may be replaced by a positive value such that both traces through these statements verify.

Our technique may be adjusted to compute k-aberrant statements by exploring all combinations of k assignments along one or more error traces.

4 Semantic Suspiciousness Ranking

In this section, we present how aberrant statements are ranked in terms of their suspiciousness. As mentioned earlier, the suspiciousness score of an aberrant statement is inversely proportional to how much code would become unreachable if we applied a local fix to the statement.

First, for each aberrant statement a_i, where $i = 0, \ldots, n$, our technique generates the instrumented program P_{a_i} (see Sect. 3.3 for the details). Recall that the trace-aberrant statements for the program of Fig. 1 are `y := ⋆` and `x := 0`.

Second, we check reachability of the code in each of these n instrumented programs P_{a_i}. Reachability may be simply checked by converting all existing assertions into assumptions and introducing an `assert false` at various locations in the program. An instrumentation for checking reachability in $P_{\texttt{x := 0}}$ is shown on the right; all changes are highlighted. In particular, we detect whether the injected assertion on line 7 is reachable by passing the above program (without the dark gray box) to an off-the-shelf analyzer. We can similarly check reachability of the other assertion. In the above program, both assertions are reachable, whereas in the corresponding program for assignment `y := ⋆`, only one of them is. The number of reachable assertions in a program P_{a_i} constitutes the suspiciousness score of statement a_i.

```
1  x := 2;
2  y := ⋆;
3  if (⋆)
4    assume y < 3;
5    x := ⋆;
6    assume 0 < x;
7    assert false;
8  } else {
9    assume 3 ≤ y;
10   x := 1;
11   assert false;
12 }
13 assume 0 < x;
```

As a final step, our technique ranks the aberrant statements in order of decreasing suspiciousness. Intuitively, this means that, by applying a local fix to the higher-ranked statements, less code would become unreachable in comparison to the statements that are ranked lower. Since developers do not typically intend to write unreachable code, the cause of the error in P is more likely to be a higher-ranked aberrant statement. For our running example, trace-aberrant statement `x := 0` is ranked higher than `y := ⋆`.

As previously discussed, when modifying the program of Fig. 1 to replace assignment `x := 1` by `x := -1`, our technique determines that only `x := 0` must be program-aberrant. For the error trace through the other branch of the conditional, we would similarly identify statement `x := -1` as must-program-aberrant. Note that, for this example, must program aberrance does not miss any program-aberrant statements. In fact, in our experiments, must program aberrance does not miss any error causes, despite its under-approximation.

5 Implementation

We have implemented our technique in a toolchain for localizing and ranking suspicious statements in C programs. We used UAutomizer in the Ultimate analysis framework to obtain error traces (version `0.1.23`). UAutomizer is a software model checker that translates C programs to Boogie [4] and then employs an automata-based verification approach [16]. Our implementation extends UAutomizer to identify (trace- or program-) aberrant statements along the generated error traces, as we describe in Sect. 3. Note that, due to abstraction (for instance, of library calls), UAutomizer may generate spurious error traces. This is an orthogonal issue that we do not address in this work.

To also identify aberrant expressions, for instance, predicates of conditionals or call arguments, we pre-process the program by first assigning these expressions to temporary variables, which are then used instead. This allows us to detect error causes relating to statements other than assignments.

Once the aberrant statements have been determined, we instrument the Boogie code to rank them (see Sect. 4). Specifically, our implementation inlines procedures and injects an `assert false` statement at the end of each basic block (one at a time). Instead of extending the existing support for "smoke checking" in Boogie, we implemented our own reachability checker in order to have more control over where the assertions are injected. While this might not be as efficient due to the larger number of Boogie queries (each including the time for parsing, pre-processing, and SMT solving), one could easily optimize or replace this component.

6 Experimental Evaluation

We evaluate the effectiveness of our technique in localizing and ranking suspicious statements by applying our toolchain to several faulty C programs. In the following, we introduce our set of benchmarks (Sect. 6.1), present the experimental setup (Sect. 6.2), and investigate four research questions (Sect. 6.3).

6.1 Benchmark Selection

For our evaluation, we used 51 faulty C programs from two independent sources. On the one hand, we used the faulty versions of the TCAS task from the Siemens test suite [19]. The authors of the test suite manually introduced faults in several tasks while aiming to make these bugs as realistic as possible. In general, the Siemens test suite is widely used in the literature (e.g., [14,20,22,25–28]) for evaluating and comparing fault-localization techniques.

The TCAS task implements an aircraft-collision avoidance system and consists of 173 lines of C code; there are no specifications. This task also comes with 1608 test cases, which we used to introduce assertions in the faulty program versions. In particular, in each faulty version, we specified the correct behavior as this was observed by running the tests against the original, correct version of the

code. This methodology is commonly used in empirical studies with the Siemens test suite, and it was necessary for obtaining an error trace from UAutomizer.

On the other hand, we randomly selected 4 correct programs (with over 250 lines of code) from the SV-COMP software-verification competition [5], which includes standard benchmarks for evaluating program analyzers. We automatically injected faults in each of these programs by randomly mutating statements within the program. All SV-COMP benchmarks are already annotated with assertions, so faults manifest themselves by violating the existing assertions.

6.2 Experimental Setup

We ran all experiments on an Intel® Core i7 CPU @ 2.67 GHz machine with 16 GB of memory, running Linux. Per analyzed program, we imposed a timeout of 120 s and a memory limit of 6 GB to UAutomizer.

To inject faults in the SV-COMP benchmarks, we developed a mutator that randomly selects an assignment statement, mutates the right-hand side, and checks whether the assertion in the program is violated. If it is, the mutator emits a faulty program version. Otherwise, it generates up to two additional mutations for the same assignment before moving on to another.

6.3 Experimental Results

To evaluate our technique, we consider the following research questions:

- **RQ1:** How effective is our technique in narrowing down the cause of an error to a small number of suspicious statements?
- **RQ2:** How efficient is our technique?
- **RQ3:** How does under-approximating program-aberrant statements affect fault localization?
- **RQ4:** How does our technique compare against state-of-the-art approaches for fault localization in terms of effectiveness and efficiency?

RQ1 (Effectiveness). Tables 1 and 2 summarize our experimental results on the TCAS and SV-COMP benchmarks, respectively. The first column of Table 1 shows the faulty versions of the program, and the second column the number of trace-aberrant statements that were detected for every version. Similarly, in Table 2, the first column shows the program version[2], the second column the lines of source code in every version, and the third column the number of trace-aberrant statements. For all benchmarks, the actual cause of each error is always

[2] A version is denoted by <*correct-program-id*>.<*faulty-version-id*>. We mutate the following correct programs from SV-COMP: 1. `mem_slave_tlm.1_true-unreach-call_false-termination.cil.c` (4 faulty versions), 2. `kundu_true-unreach-call_false-termination.cil.c` (4 faulty versions), 3. `mem_slave_tlm.2_true-unreach-call_false-termination.cil.c` (2 faulty versions), and 4. `pc_sfifo_1_true-unreach-call_false-termination.cil.c` (1 faulty version). All versions are at: https://github.com/numairmansur/SemanticFaultLocalization_Benchmarks.

Table 1. Our experimental results for the TCAS benchmarks.

Prg	Abr stmts		Rank		Time (m:s)		Program reduction (%)							
ver	trc	prg	trc	prg	trc	prg	trc	prg	A	B	C	D	E	F
1	16	11	5	3	2:24	1:17	8.1	5.7	1.7	1.7	1.7	1.7	1.7	8.6
2	15	13	1	1	2:33	1:46	1.7	1.7	6.3	5.7	5.7	6.3	5.7	4.6
3	4	4	1	1	0:29	0:31	1.7	1.7	12.7	12.1	10.4	12.7	12.1	9.8
4	16	14	4	4	2:59	1:53	7.5	7.5	1.7	1.7	1.7	1.7	1.7	9.2
5	4	4	1	1	0:28	0:33	1.7	1.7	14.4	10.9	9.8	14.4	10.9	8.6
6	13	13	5	5	1:51	1:53	7.5	7.5	2.8	2.8	2.8	2.8	2.8	8.6
7	15	13	5	5	2:28	1:57	6.9	6.3	12.1	12.1	9.8	12.1	12.1	9.2
8	14	13	5	5	2:14	1:51	6.9	5.7	12.7	12.7	12.7	12.7	12.7	8.6
9	16	14	1	1	2:21	1:55	1.7	1.7	7.5	7.5	7.5	7.5	7.5	5.2
10	15	14	2	2	2:12	1:53	3.4	2.8	11.5	11.5	15.0	11.5	16.1	9.2
11	15	7	2	1	1:58	0:56	2.8	1.7	2.8	2.8	2.3	2.8	2.8	6.3
12	16	14	3	3	2:38	1:46	5.7	5.2	12.7	12.1	9.8	12.7	12.1	9.2
13	17	15	5	5	2:40	1:56	6.3	5.2	15.6	12.1	10.4	15.6	12.1	9.2
14	4	4	1	1	0:30	0:29	1.7	1.7	5.7	5.7	5.7	5.7	5.7	8.1
15	17	15	4	4	2:52	2:02	7.5	6.3	15.0	13.2	10.4	15.0	13.2	7.5
16	16	14	5	5	2:22	1:57	6.3	6.3	12.1	12.1	12.1	12.1	12.1	9.2
17	16	14	5	5	2:37	2:03	8.0	7.5	12.1	12.1	9.8	12.1	12.1	9.2
18	16	14	5	5	2:02	1:51	8.0	7.5	14.4	11.5	9.8	14.4	11.5	6.9
19	16	14	5	5	2:20	1:55	8.0	7.5	12.1	12.1	9.8	12.1	12.1	9.2
20	15	11	1	1	2:16	1:23	1.7	1.7	7.5	7.5	7.5	7.5	7.5	9.2
21	16	12	1	1	2:16	1:27	1.7	1.7	7.5	7.5	7.5	7.5	7.5	8.6
22	17	8	1	1	2:45	1:03	2.3	1.7	7.5	7.5	7.5	7.5	7.5	5.7
23	14	11	1	1	2:22	1:30	1.7	1.7	7.5	7.5	7.5	7.5	7.5	6.3
24	17	13	1	1	2:39	1:57	2.3	1.7	7.5	7.5	7.5	7.5	7.5	8.6
25	16	14	3	3	1:44	1:51	9.2	8.1	1.1	1.1	1.1	1.1	1.1	6.9
26	15	13	3	3	2:17	1:52	2.8	1.7	13.8	12.7	10.4	13.8	12.7	9.2
27	15	13	4	4	2:36	1:58	8.6	8.1	14.4	10.9	9.8	14.4	10.9	10.9
28	15	13	1	1	2:03	1:50	1.7	1.7	6.3	5.7	4.0	6.3	5.7	5.7
29	12	8	3	2	1:46	1:09	1.7	1.7	6.3	6.3	5.7	6.3	5.7	5.7
30	14	8	2	2	1:47	1:06	1.7	1.7	6.3	5.7	5.7	6.3	5.7	5.7
31	17	15	4	4	2:32	1:57	8.1	6.3	2.8	2.8	2.8	2.8	2.8	10.9
32	14	13	5	5	1:55	1:54	6.3	6.3	2.8	2.8	2.8	2.8	2.8	10.9
33	17	15	4	4	2:48	2:05	2.8	2.8	14.4	12.7	10.9	14.4	12.7	—
34	16	14	4	4	2:43	2:13	8.0	6.9	13.2	12.7	10.4	13.2	12.7	8.6
35	14	8	2	2	1:59	0:59	2.3	1.7	6.3	5.7	4.0	6.3	5.7	5.7
36	6	3	1	1	0:43	0:26	3.4	1.7	13.2	13.2	10.4	13.2	13.2	2.9
37	17	8	6	3	2:43	1:04	9.8	4.6	1.1	1.1	0.5	1.1	0.5	8.6
39	16	7	5	1	2:45	0:55	9.2	4.6	1.1	1.1	1.1	1.1	1.1	6.9
40	16	14	4	4	2:48	1:46	6.3	6.3	15.0	15.0	15.0	15.0	15.6	6.3
41	17	15	4	4	2:50	2:00	9.8	8.6	1.7	1.7	1.7	1.7	1.7	8.6
Avg	14.4	11.5	3.1	2.9	2:12	1:34	5.1	4.3	8.6	8.0	7.3	8.6	8.1	7.9

included in the statements that our technique identifies as trace-aberrant. This is to be expected since the weakest-precondition and strongest-postcondition transformers that we use for determining trace aberrance are known to be sound.

The fourth column of Table 1 and the fifth column of Table 2 show the suspiciousness rank that our technique assigns to the actual cause of each error. For both sets of benchmarks, the average rank of the faulty statement is 3, and all faulty statements are ranked in the top 6. A suspiciousness rank of 3 means that

users need to examine *at least* three statements to identify the problem; they might have to consider more in case multiple statements have the same rank.

To provide a better indication of how much code users have to examine to identify the bug, the eighth column of Table 1 and the seventh column of Table 2 show the percentage reduction in the program size. On average, our technique reduces the code size down to 5% for TCAS and less than 1% for SV-COMP.

RQ2 (Efficiency). The sixth column of Table 1 shows the time that our technique requires for identifying the trace-aberrant statements in a given error trace as well as for ranking them in terms of suspiciousness. This time does not include the generation of the error trace by UAutomizer. As shown in the table, our technique takes only a little over 2 min on average to reduce a faulty program to about 5% of its original size.

Table 2. Our experimental results for the SV-COMP benchmarks.

Prg ver	LoSC	Abr stmts		Rank		Rdc (%)	
		trc	prg	trc	prg	trc	prg
1.1	1336	34	29	4	2	0.4	0.2
1.2	1336	34	28	4	2	0.4	0.2
1.3	1336	34	31	2	1	0.2	0.1
1.4	1336	34	30	3	1	0.4	0.1
2.1	630	23	10	3	2	1.3	0.3
2.2	630	16	8	1	1	0.3	0.3
2.3	630	22	9	3	2	1.3	0.3
2.4	630	27	25	4	3	1.2	1.1
3.1	1371	37	33	3	1	0.3	0.2
3.2	1371	37	32	3	1	0.3	0.1
4.1	360	18	8	4	2	3.3	0.8
Average		28.7	22.0	3.0	1.6	0.8	0.3

Note that most of this time (98.5% on average) is spent on the suspiciousness ranking. The average time for determining the trace-aberrant statements in an error trace is only 1.7 s. Recall from Sect. 5 that our reachability analysis, which is responsible for computing the suspiciousness score of each aberrant statement, is not implemented as efficiently as possible (see Sect. 5 for possible improvements).

RQ3 (Program aberrance). In Sect. 3.3, we discussed that our technique can under-approximate the set of program-aberrant statements along an error trace. The third, fifth, seventh, and ninth columns of Table 1 as well as the fourth, sixth, and eighth columns of Table 2 show the effect of this under-approximation.

There are several observations to be made here, especially in comparison to the experimental results for trace aberrance. First, there are fewer aberrant statements, which is to be expected since (must-)program-aberrant statements may only be a subset of trace-aberrant statements. Perhaps a bit surprisingly, the actual cause of each error is always included in the must-program-aberrant statements. In other words, the under-approximation of program-aberrant statements does not miss any error causes in our benchmarks. Second, the suspiciousness rank assigned to the actual cause of each error is slightly higher, and all faulty

statements are ranked in the top 5. Third, our technique requires about 1.5 min for fault localization and ranking, which is faster due to the smaller number of aberrant statements. Fourth, the code is reduced even more, down to 4.3 for TCAS and 0.3% for SV-COMP.

RQ4 (Comparison). To compare our technique against state-of-the-art fault-localization approaches, we evaluated how five of the most popular [35] spectrum-based fault-localization (SBFL) techniques [20,24,34] perform on our benchmarks. In general, SBFL is the most well-studied and evaluated fault-localization technique in the literature [26]. SBFL techniques essentially compute suspiciousness scores based on statement-execution frequencies. Specifically, the more frequently a statement is executed by failing test cases and the less frequently it is executed by successful tests, the higher its suspiciousness score. We also compare against an approach that reduces fault localization to the maximal satisfiability problem (MAX-SAT) and performs similarly to SBFL.

The last eight columns of Table 1 show the comparison in code reduction across different fault-localization techniques. Columns A, B, C, D, and E refer to the SBFL techniques, and in particular, to Tarantula [20], Ochiai [2], Op2 [24], Barinel [1], and DStar [34], respectively. The last column (F) corresponds to BugAssist [21,22], which uses MAX-SAT. To obtain these results, we implemented all SBFL techniques and evaluated them on TCAS using the existing test suite. For BugAssist, we used the published percentages of code reduction for these benchmarks [22]. Note that we omit version 38 in Table 1 as is common in experiments with TCAS. The fault is in a non-executable statement (array declaration) and its frequency cannot be computed by SBFL.

The dark gray boxes in the table show which technique is most effective with respect to code reduction for each version. Our technique for must program aberrance is the most effective for 30 out of 40 versions. The light gray boxes in the trace-aberrance column denote when this technique is the most effective in comparison with columns A–F (that is, without considering program aberrance). As shown in the table, our technique for trace aberrance outperforms approaches A–F in 28 out of 40 versions. In terms of lines of code, users need to inspect 7–9 statements when using our technique, whereas they would need to look at 13–15 statements when using other approaches. This is a reduction of 4–8 statements, and every statement that users may safely ignore saves them valuable time.

Regarding efficiency, our technique is comparable to SBFL (A–E); we were not able to run BugAssist (F), but it should be very lightweight for TCAS. SBFL techniques need to run the test suite for every faulty program. For the TCAS tests, this takes 1 min 11 s on average on our machine. Parsing the statement-execution frequencies and computing the suspiciousness scores takes about 5 more seconds. Therefore, the average difference with our technique ranges from a few seconds (for program aberrance) to a little less than a minute (for trace aberrance). There is definitely room for improving the efficiency of our technique, but despite it being slightly slower than SBFL for these benchmarks, it saves the user the effort of inspecting non-suspicious statements. Moreover, note that the larger the test suite, the higher the effectiveness of SBFL, and the

longer its running time. Thus, to be as effective as our technique, SBFL would require more test cases, and the test suite would take longer to run. We do not consider the time for test case generation just like we do not consider the running time of the static analysis that generates the error traces.

7 Related Work

Among the many fault-localization techniques [35], SBFL [20,24,34] is the most well-studied and evaluated. Mutation-based fault localization (MBFL) [23,25] is almost as effective as SBFL but significantly more inefficient [26]. In general, MBFL extends SBFL by considering, not only how frequently a statement is executed in tests, but also whether a mutation to the statement affects the test outcomes. So, MBFL generates many mutants per statement, which requires running the test suite per mutant, and not per faulty program as in SBFL. Our local fixes resemble mutations, but they are performed *symbolically* and can be seen as applying program-level abductive reasoning [6,11,12] or angelic verification [8] for fault localization.

The use of error invariants [7,13,18,29] is a closely-related fault-localization technique. Error invariants are computed from Craig interpolants along an error trace and capture which states will produce the error from that point on. They are used for slicing traces by only preserving statements whose error invariants before and after the statement differ. Similarly, Wang et al. [32] use a syntactic-level weakest-precondition computation for a given error trace to produce a minimal set of word-level predicates, which explain why the program fails. In contrast, we use the novel notion of trace aberrance for this purpose and compute a suspiciousness ranking to narrow down the error cause further.

Griesmayer et al. [14] use an error trace from a bounded model checker to instrument the program with "abnormal predicates". These predicates allow expressions in the program to take arbitrary values, similarly to how our technique replaces a statement $v := e$ by a non-deterministic one. Unlike our technique, their approach may generate a prohibitively large instrumentation, requires multiple calls to the model checker, and does not rank suspicious statements.

Several fault-localization algorithms leverage the differences between faulty and successful traces [3,15,27,36]. For instance, Ball et al. [3] make several calls to a model checker and compare any generated counterexamples with successful traces. In contrast, we do not require successful traces for comparisons.

Zeller [36] uses delta-debugging, which identifies suspicious parts of the input by running the program multiple times. Slicing [31,33] removes statements that are definitely not responsible for the error based on data and control dependencies. Shen et al. [30] use unsatisfiable cores for minimizing counterexamples. Our technique is generally orthogonal to these approaches, which could be run as a pre-processing step to reduce the search space.

8 Conclusion

We have presented a novel technique for fault localization and suspiciousness ranking of statements along an error trace. We demonstrated its effectiveness in narrowing down the error cause to a small fraction of the entire program.

As future work, we plan to evaluate the need for k-aberrance by analyzing software patches and to combine our technique with existing approaches for program repair to improve their effectiveness.

Acknowledgments. This work was supported by the German Research Foundation (DFG) as part of CRC 248 (https://www.perspicuous-computing.science), the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award), and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 754411.

References

1. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: Spectrum-based multiple fault localization. In: ASE, pp. 88–99. IEEE Computer Society (2009)
2. Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C.: A practical evaluation of spectrum-based fault localization. JSS **82**, 1780–1792 (2009)
3. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL, pp. 97–105. ACM (2003)
4. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006). https://doi.org/10.1007/11804192_17
5. Beyer, D.: Competition on software verification (SV-COMP) (2017). https://sv-comp.sosy-lab.org
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL, pp. 289–300. ACM (2009)
7. Christ, J., Ermis, E., Schäf, M., Wies, T.: Flow-sensitive fault localization. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 189–208. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35873-9_13
8. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: precise verification modulo unknowns. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 324–342. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_19
9. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. CACM **18**, 453–457 (1975)
10. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, New York (1990). https://doi.org/10.1007/978-1-4612-3228-5
11. Dillig, I., Dillig, T.: Explain: a tool for performing abductive inference. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 684–689. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_46
12. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI, pp. 181–192. ACM (2012)

13. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_17
14. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. ENTCS **174**, 95–111 (2007)
15. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**, 229–247 (2006)
16. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
17. Hoare, C.A.R.: An axiomatic basis for computer programming. CACM **12**, 576–580 (1969)
18. Holzer, A., Schwartz-Narbonne, D., Tabaei Befrouei, M., Weissenbacher, G., Wies, T.: Error invariants for concurrent traces. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 370–387. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_23
19. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.J.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: ICSE, pp. 191–200. IEEE Computer Society/ACM (1994)
20. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: ASE, pp. 273–282. ACM (2005)
21. Jose, M., Majumdar, R.: Bug-assist: assisting fault localization in ANSI-C programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 504–509. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_40
22. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI, pp. 437–446. ACM (2011)
23. Moon, S., Kim, Y., Kim, M., Yoo, S.: Ask the mutants: mutating faulty programs for fault localization. In: ICST, pp. 153–162. IEEE Computer Society (2014)
24. Naish, L., Lee, H.J., Ramamohanarao, K.: A model for spectra-based software diagnosis. TOSEM **20**, 11:1–11:32 (2011)
25. Papadakis, M., Le Traon, Y.: Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. **25**, 605–628 (2015)
26. Pearson, S., et al.: Evaluating and improving fault localization. In: ICSE, pp. 609–620. IEEE Computer Society/ACM (2017)
27. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39. IEEE Computer Society (2003)
28. Santelices, R.A., Jones, J.A., Yu, Y., Harrold, M.J.: Lightweight fault-localization using multiple coverage types. In: ICSE, pp. 56–66. IEEE Computer Society (2009)
29. Schäf, M., Schwartz-Narbonne, D., Wies, T.: Explaining inconsistent code. In: ESEC/FSE, pp. 521–531. ACM (2013)
30. Shen, S.Y., Qin, Y., Li, S.K.: Minimizing counterexample with unit core extraction and incremental SAT. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 298–312. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_20
31. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**, 121–189 (1995)
32. Wang, C., Yang, Z., Ivančić, F., Gupta, A.: Whodunit? Causal analysis for counterexamples. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 82–95. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_9
33. Weiser, M.: Program slicing. In: ICSE, pp. 439–449. IEEE Computer Society (1981)

34. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. Trans. Reliab. **63**, 290–308 (2014)
35. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. TSE **42**, 707–740 (2016)
36. Zeller, A.: Isolating cause-effect chains from computer programs. In: FSE, pp. 1–10. ACM (2002)

Computing Coupled Similarity

Benjamin Bisping(✉) and Uwe Nestmann

Technische Universität Berlin, Berlin, Germany
{benjamin.bisping,uwe.nestmann}@tu-berlin.de

Abstract. *Coupled similarity* is a notion of equivalence for systems with internal actions. It has outstanding applications in contexts where internal choices must transparently be distributed in time or space, for example, in process calculi encodings or in action refinements. No tractable algorithms for the computation of coupled similarity have been proposed up to now. Accordingly, there has not been any tool support.

We present a *game-theoretic algorithm to compute coupled similarity*, running in cubic time and space with respect to the number of states in the input transition system. We show that one cannot hope for much better because deciding the coupled simulation preorder is at least as hard as deciding the weak simulation preorder.

Our results are backed by an *Isabelle/HOL* formalization, as well as by a parallelized implementation using the *Apache Flink* framework. Data or code related to this paper is available at: [2].

1 Introduction

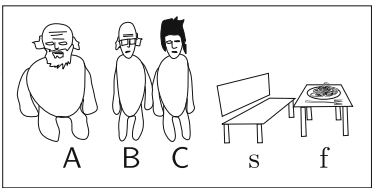
Coupled similarity hits a sweet spot within the *linear-time branching-time spectrum* [9]. At that spot, one can encode between brands of process calculi [14,22,25], name a branching-time semantics for Communicating Sequential Processes [10], distribute synchronizations [23], and refine atomic actions [5,28]. Weak bisimilarity is too strong for these applications due to the occurrence of situations with *partially commited states* like in the following example.

Example 1 (Gradually committing philosophers). Three philosophers A, B, and C want to eat pasta. To do so, they must first sit down on a bench s and grab a fork f. Unfortunately, only either A alone or the thinner B and C together can fit on the bench, and there is just one fork. From the outside, we are only interested in the fact which of them gets to eat. So we consider the whole bench-and-fork business internal to the system. The following CCS structure models the situation in the notation of [21]. The resources correspond to output actions (which can be consumed only once) and obtaining the resources corresponds to input actions.

$$\mathsf{P}_g \stackrel{\text{def}}{=} \left(\bar{\mathsf{s}} \mid \bar{\mathsf{f}} \mid \mathsf{s.f.A} \mid \mathsf{s.}(\mathsf{f.B} \mid \mathsf{f.C})\right) \setminus \{\mathsf{s},\mathsf{f}\}$$

$$\mathsf{A} \stackrel{\text{def}}{=} \mathsf{aEats.A} \qquad \mathsf{B} \stackrel{\text{def}}{=} \mathsf{bEats.B}$$

$$\mathsf{C} \stackrel{\text{def}}{=} \mathsf{cEats.C}$$



T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 244–261, 2019.
https://doi.org/10.1007/978-3-030-17462-0_14

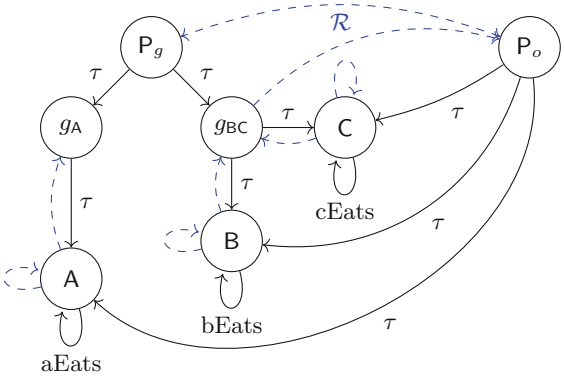


Fig. 1. A non-maximal weak/coupled simulation $\mathcal{R}$ on the philosopher system from Example 1. (Color figure online)

One might now be inclined to ponder that exactly one of the philosophers will get both resources and that we thus could merge s and f into a single resource sf:

$$\mathsf{P}_o \stackrel{\text{def}}{=} \left(\overline{\mathsf{sf}} \quad | \quad \mathsf{sf.A} \mid \mathsf{sf.B} \mid \mathsf{sf.C}\right) \setminus \{\mathsf{sf}\}$$

The structure of P_g and P_o has the transition system in Fig. 1 as its semantics. Notice that the internal communication concerning the resource allocation turns into internal τ-actions, which in P_g, g_A, and g_BC *gradually decide* who is going to eat the pasta, whereas P_o *decides in one step.*

P_g and P_o are mutually related by a weak simulation (blue dashed lines in Fig. 1) and hence *weakly similar*. However, there cannot be a *symmetric* weak simulation relating them because $\mathsf{P}_g \xrightarrow{\tau} g_\mathsf{BC}$ cannot be matched symmetrically by P_o as no other reachable state shares the weakly enabled actions of g_BC. Thus, they are *not weakly bisimilar*. This counters the intuition that weak bisimilarity ignores how much internal behavior happens between visible actions. There seems to be no good argument how an outside observer should notice the difference whether an internal choice is made in one or two steps.

So how to fix this overzealousness of weak bisimilarity? Falling back to weak similarity would be too coarse for many applications because it lacks the property of weak bisimilarity to coincide with strong bisimilarity on systems without internal behavior. This property, however, is present in notions that refine *contrasimilarity* [31]. There is an easy way to having the cake and eating it, here: *Coupled similarity* is precisely the intersection of contrasimilarity and weak similarity (Fig. 2). It can be defined by adding a weak form of symmetry (*coupling*) to weak simulation. The weak simulation in Fig. 1 fulfills coupling and thus is a coupled simulation. This shows that coupled similarity is coarse enough for situations with gradual commitments. At the same time, it is a close fit for weak bisimilarity, with which it coincides for many systems.

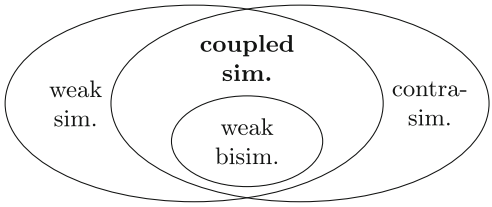


Fig. 2. Notions of equivalence for systems with internal actions.

Up to now, no algorithms and tools have been developed to enable a wider use of coupled similarity in automated verification settings. Parrow and Sjödin [24] have only hinted at an exponential-space algorithm and formulated as an open research question whether coupled similarity can be decided in **P**. For similarity and bisimilarity, polynomial algorithms exist. The best algorithms for weak bisimilarity [3,19,26] are slightly sub-cubic in time, $\mathcal{O}(|S|^2 \log |S|)$ for transition systems with $|S|$ states. The best algorithms for similarity [15,27], adapted for weak similarity, are cubic. Such a slope between similarity and bisimilarity is common [18]. As we show, coupled similarity inherits the higher complexity of weak similarity. Still, the closeness to weak bisimilarity can be exploited to speed up computations.

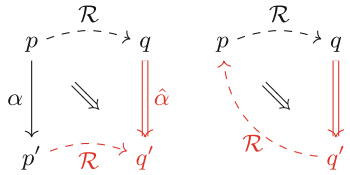
Contributions. This paper makes the following contributions.

- We prove that action-based single-relation *coupled similarity can be defined in terms of coupled delay simulation* (Subsect. 2.2).
- We *reduce weak similarity to coupled similarity*, thereby showing that deciding coupled similarity inherits the complexity of weak similarity (Subsect. 2.4).
- We present and verify a simple polynomial-time *coupled simulation fixed-point algorithm* (Sect. 3).
- We *characterize the coupled simulation preorder by a game and give an algorithm*, which runs in cubic time and can be nicely optimized (Sect. 4)
- We *implement the game algorithm for parallel computation using Apache Flink* and benchmark its performance (Sect. 5).

Technical details can be found in the first author's Master's thesis [1]. Isabelle/HOL [32] proofs are available from https://coupledsim.bbisping.de/isabelle/.

2 Coupled Similarity

This section characterizes the coupled simulation preorder for transition systems with silent steps in terms of coupled delay simulation. We prove properties that are key to the correctness of the following algorithms.



A: Weak simulation **B:** Coupling

Fig. 3. Illustration of weak simulation and coupling on transition systems (Definition 4, black part implies red part). (Color figure online)

2.1 Transition Systems with Silent Steps

Labeled transition systems capture a discrete world view, where there is a current state and a branching structure of possible state changes ("transitions") to future states.

Definition 1 (Labeled transition system). *A* labeled transition system *is a tuple* $\mathcal{S} = (S, \Sigma_\tau, \rightarrow)$ *where* S *is a set of* states, Σ_τ *is a set of* actions *containing a special* internal action $\tau \in \Sigma_\tau$, *and* $\rightarrow \subseteq S \times \Sigma_\tau \times S$ *is the* transition relation. *We call* $\Sigma \coloneqq \Sigma_\tau \setminus \{\tau\}$ *the* visible actions.

The weak transition relation $\hat{\Rightarrow}$ *is defined as the reflexive transitive closure of internal steps* $\stackrel{\hat{\tau}}{\Rightarrow} \coloneqq \stackrel{\tau}{\rightarrow}^*$ *combined with* $\stackrel{\hat{a}}{\Rightarrow} \coloneqq \stackrel{\hat{\tau}}{\Rightarrow}\stackrel{a}{\rightarrow}\stackrel{\hat{\tau}}{\Rightarrow}$ $(a \in \Sigma)$.

As a shorthand for $\stackrel{\hat{\tau}}{\Rightarrow}$, we also write just $\Rightarrow$. We call an $\stackrel{\hat{\alpha}}{\Rightarrow}$-step "weak" whereas an $\stackrel{\alpha}{\rightarrow}$-step is referred to as "strong" ($\alpha \in \Sigma_\tau$). A visible action $a \in \Sigma$ is said to be *weakly enabled* in p iff there is some p' such that $p \stackrel{\hat{a}}{\Rightarrow} p'$.

Definition 2 (Stability and divergence). *A state* p *is called* stable *iff it has no* τ*-transitions,* $p \stackrel{\tau}{\not\rightarrow}$. *A state* p *is called* divergent *iff it is possible to perform an infinite sequence of* τ*-transitions beginning in this state,* $p \stackrel{\tau}{\rightarrow}^\omega$.

2.2 Defining Coupled Similarity

Coupled simulation is often defined in terms of two *weak simulations*, but it is more convenient to use just a single one [10], which extends weak simulation with a weak form of symmetry, we shall call *coupling* (Fig. 3).

Definition 3 (Weak simulation). *A* weak simulation *is a relation* $\mathcal{R} \subseteq S \times S$ *such that, for all* $(p, q) \in \mathcal{R}$, $p \stackrel{\alpha}{\rightarrow} p'$ *implies that there is a* q' *such that* $q \stackrel{\hat{\alpha}}{\Rightarrow} q'$ *and* $(p', q') \in \mathcal{R}$.

Definition 4 (Coupled simulation). *A* coupled simulation *is a weak simulation* $\mathcal{R} \subseteq S \times S$ *such that, for all* $(p, q) \in \mathcal{R}$, *there exists a* q' *such that* $q \Rightarrow q'$ *and* $(q', p) \in \mathcal{R}$ (coupling).

The coupled simulation preorder *relates two processes,* $p \sqsubseteq_{CS} q$*, iff there is a coupled simulation* $\mathcal{R}$ *such that* $(p, q) \in \mathcal{R}$. Coupled similarity *relates two processes,* $p \equiv_{CS} q$*, iff* $p \sqsubseteq_{CS} q$ *and* $q \sqsubseteq_{CS} p$.

Adapting words from [10], $p \sqsubseteq_{CS} q$ intuitively does not only mean that "p is *ahead* of q" (weak simulation), but also that "q can *catch up* to p" (coupling). The weak simulation on the philosopher transition system from Example 1 is coupled.

Coupled similarity can also be characterized employing an effectively stronger concept than weak simulation, namely *delay simulation*. Delay simulations [11, 28] are defined in terms of a "shortened" weak step relation $\overset{\alpha}{\Rightarrow}$ where $\overset{\tau}{\Rightarrow} := \mathrm{id}$ and $\overset{a}{\Rightarrow} := \Rightarrow\overset{a}{\rightarrow}$. So the difference between $\overset{a}{\Rightarrow}$ and $\overset{\hat{a}}{\Rightarrow}$ lies in the fact that the latter can move on with τ-steps after the strong $\overset{a}{\rightarrow}$-step in its construction.

Definition 5 (Coupled delay simulation). *A* coupled delay simulation *is a relation* $\mathcal{R} \subseteq S \times S$ *such that, for all* $(p, q) \in \mathcal{R}$,

- $p \overset{\alpha}{\rightarrow} p'$ *implies there is a* q' *such that* $q \overset{\alpha}{\Rightarrow} q'$ *and* $(p', q') \in \mathcal{R}$ *(*delay simulation*),*
- *and there exists a* q' *such that* $q \Rightarrow q'$ *and* $(q', p) \in \mathcal{R}$ *(*coupling*).*

The only difference to Definition 4 is the use of $\overset{\alpha}{\Rightarrow}$ instead of $\overset{\hat{\alpha}}{\Rightarrow}$. Some coupled simulations are no (coupled) delay simulations, for example, consider $\mathcal{R} = \{(\mathrm{c}.\tau, \mathrm{c}.\tau), (\tau, \mathbf{0}), (\mathbf{0}, \tau), (\mathbf{0}, \mathbf{0})\}$ on CCS processes. Still, the *greatest* coupled simulation $\sqsubseteq_{CS}$ *is* a coupled delay simulation, which enables the following characterization:

Lemma 1. $p \sqsubseteq_{CS} q$ *precisely if there is a coupled delay simulation* $\mathcal{R}$ *such that* $(p, q) \in \mathcal{R}$.

2.3 Order Properties and Coinduction

Lemma 2. $\sqsubseteq_{CS}$ *forms a preorder, that is, it is reflexive and transitive. Coupled similarity* $\equiv_{CS}$ *is an equivalence relation.*

Lemma 3. *The coupled simulation preorder can be characterized coinductively by the rule:*

$$\frac{\forall p', \alpha.\ p \xrightarrow{\alpha} p' \longrightarrow \exists q'.\ q \overset{\alpha}{\Rightarrow} q' \ \wedge\ p' \sqsubseteq_{CS} q' \qquad \exists q'.\, q \Rightarrow q' \wedge q' \sqsubseteq_{CS} p}{p \sqsubseteq_{CS} q}.$$

This coinductive characterization motivates the fixed-point algorithm (Sect. 3) and the game characterization (Sect. 4) central to this paper.

Lemma 4. *If* $q \Rightarrow p$*, then* $p \sqsubseteq_{CS} q$.

Corollary 1. *If* p *and* q *are on a* τ*-cycle, that means* $p \Rightarrow q$ *and* $q \Rightarrow p$*, then* $p \equiv_{CS} q$.

Ordinary coupled simulation is blind to divergence. In particular, it cannot distinguish two states whose outgoing transitions only differ in an additional τ-loop at the second state:

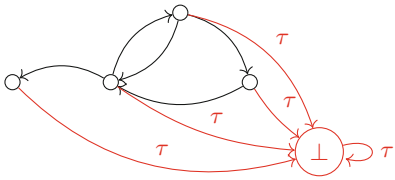


Fig. 4. Example for $\mathcal{S}^{\perp}$ from Theorem 1 ($\mathcal{S}$ in black, $\mathcal{S}^{\perp}\backslash\mathcal{S}$ in red). (Color figure online)

Lemma 5. *If* $p \xrightarrow{\alpha} p' \longleftrightarrow q \xrightarrow{\alpha} p' \vee p' = p \wedge \alpha = \tau$ *for all* α, p'*, then* $p \equiv_{CS} q$*.*

Due to the previous two results, finite systems with divergence can be transformed into $\equiv_{CS}$-equivalent systems without divergence. This connects the original notion of stability-coupled similarity [23,24] to our modern formulation and motivates the usefulness of the next lemma.

Coupling can be thought of as "weak symmetry." For a relation to be symmetric, $\mathcal{R}^{-1} \subseteq \mathcal{R}$ must hold whereas coupling means that $\mathcal{R}^{-1} \subseteq \Rightarrow\mathcal{R}$. This weakened symmetry of coupled similarity can guarantee weak bisimulation on steps to stable states:

Lemma 6. *Assume* $\mathcal{S}$ *is finite and has no* τ*-cycles. Then* $p \sqsubseteq_{CS} q$ *and* $p \stackrel{\hat{\alpha}}{\Rightarrow} p'$ *with stable* p' *imply there is a stable* q' *such that* $q \stackrel{\hat{\alpha}}{\Rightarrow} q'$ *and* $p' \equiv_{CS} q'$*.*

2.4 Reduction of Weak Simulation to Coupled Simulation

Theorem 1. *Every decision algorithm for the coupled simulation preorder in a system* $\mathcal{S}$*,* $\sqsubseteq^{\mathcal{S}}_{CS}$*, can be used to decide the weak simulation preorder,* $\sqsubseteq^{\mathcal{S}}_{WS}$*, (without relevant overhead with respect to space or time complexity).*

Proof. Let $\mathcal{S} = (S, \Sigma_\tau, \rightarrow)$ be an arbitrary transition system and $\perp \notin S$. Then

$$\mathcal{S}^{\perp} := \Big(S \cup \{\perp\}, \quad \Sigma_\tau, \quad \dot{\rightarrow} \cup \{(p, \tau, \perp) \mid p \in S \cup \{\perp\}\}\Big)$$

extends $\mathcal{S}$ with a sink $\perp$ that can be reached by a τ-step from everywhere. For an illustration see Fig. 4. Note that for $p, q \neq \perp$, $p \sqsubseteq^{\mathcal{S}}_{WS} q$ exactly if $p \sqsubseteq^{\mathcal{S}^{\perp}}_{WS} q$. On $\mathcal{S}^{\perp}$, coupled simulation preorder and weak simulation preorder coincide, $\sqsubseteq^{\mathcal{S}^{\perp}}_{WS} = \sqsubseteq^{\mathcal{S}^{\perp}}_{CS}$, because $\perp$ is τ-reachable everywhere, and, for each p, $\perp \sqsubseteq^{\mathcal{S}^{\perp}}_{CS} p$ discharges the coupling constraint of coupled simulation.

Because $\sqsubseteq^{\mathcal{S}}_{WS}$ can be decided by deciding $\sqsubseteq^{\mathcal{S}^{\perp}}_{CS}$, a decision procedure for $\sqsubseteq_{CS}$ also induces a decision procedure for $\sqsubseteq_{WS}$. The transformation has linear time in terms of state space size $|S|$ and adds only one state to the problem size.

```
def fp_step_{(S,Σ_τ,→)}(R):
    return {(p,q) ∈ R |
        (∀p', α. p -α-> p' ⟶ ∃q'. (p',q') ∈ R ∧ q =α=> q')
        ∧ (∃q'. q ⇒ q' ∧ (q',p) ∈ R)}
def fp_compute_cs(S = (S, Σ_τ, →)):
    R := S × S
    while fp_step_S(R) ≠ R:
        R := fp_step_S(R)
    return R
```

Algorithm 1: Fixed-point algorithm for the coupled simulation preorder.

3 Fixed-Point Algorithm for Coupled Similarity

The coinductive characterization of $\sqsubseteq_{CS}$ in Lemma 3 induces an extremely simple polynomial-time algorithm to compute the coupled simulation preorder as a *greatest fixed point*. This section introduces the algorithm and proves its correctness.

3.1 The Algorithm

Roughly speaking, the algorithm first considers the universal relation between states, $S \times S$, and then proceeds by removing every pair of states from the relation that would contradict the coupling or the simulation property. Its pseudo code is depicted in Algorithm 1.

fp_step plays the role of removing the tuples that would immediately violate the simulation or coupling property from the relation. Of course, such a pruning might invalidate tuples that were not rejected before. Therefore, fp_compute_cs repeats the process until $\mathsf{fp\_step}_{\mathcal{S}}(\mathcal{R}) = \mathcal{R}$, that is, until $\mathcal{R}$ is a fixed point of $\mathsf{fp\_step}_{\mathcal{S}}$.

3.2 Correctness and Complexity

It is quite straight-forward to show that Algorithm 1 indeed computes $\sqsubseteq_{CS}$ because of the resemblance between fp_step and the coupled simulation property itself, and because of the monotonicity of fp_step.

Lemma 7. *If $\mathcal{R}$ is the greatest fixed point of* fp_step*, then $\mathcal{R} = \sqsubseteq_{CS}$.*

On finite labeled transition systems, that is, with finite S and $\rightarrow$, the while loop of fp_compute_cs is guaranteed to terminate at the greatest fixed point of fp_step (by a dual variant of the Kleene fixed-point theorem).

Lemma 8. *For finite $\mathcal{S}$,* fp_compute_cs$(\mathcal{S})$ *computes the greatest fixed point of* $\mathsf{fp\_step}_{\mathcal{S}}$.

Theorem 2. *For finite* $\mathcal{S}$, fp_compute_cs($\mathcal{S}$) *returns* $\sqsubseteq_{CS}^{\mathcal{S}}$.

We verified the proof using Isabelle/HOL. Due to its simplicity, we can trust implementations of Algorithm 1 to faithfully return sound and complete $\sqsubseteq_{CS}$-relations. Therefore, we use this algorithm to generate reliable results within test suites for the behavior of other $\sqsubseteq_{CS}$-implementations.

The space complexity, given by the maximal size of $\mathcal{R}$, clearly is in $\mathcal{O}(|S|^2)$. Time complexity takes some inspection of the algorithm. For our considerations, we assume that $\dot{\Rightarrow}$ has been pre-computed, which can slightly increase the space complexity to $\mathcal{O}(|\Sigma|\,|S|^2)$.

Lemma 9. *The running time of* fp_compute_cs *is in* $\mathcal{O}(|\Sigma|\,|S|^6)$.

Proof. Checking the simulation property for a tuple $(p,q) \in \mathcal{R}$ means that for all $\mathcal{O}(|\Sigma|\,|S|)$ outgoing $p\dot{\longrightarrow}$-transitions, each has to be matched by a $q\dot{\Rightarrow}$-transition with identical action, of which there are at most $|S|$. So, simulation checking costs $\mathcal{O}(|\Sigma|\,|S|^2)$ time per tuple. Checking the coupling can be approximated by $\mathcal{O}(|S|)$ per tuple. Simulation dominates coupling. The amount of tuples that have to be checked is in $\mathcal{O}(|S|^2)$. Thus, the overall complexity of one invocation of fp_step is in $\mathcal{O}(|\Sigma|\,|S|^4)$.

Because every invocation of fp_step decreases the size of $\mathcal{R}$ or leads to termination, there can be at most $\mathcal{O}(|S|^2)$ invocations of fp_step in fp_compute_cs. Checking whether fp_step changes $\mathcal{R}$ can be done without notable overhead. In conclusion, we arrive at an overall time complexity of $\mathcal{O}(|\Sigma|\,|S|^6)$.

Now, it does not take much energy to spot that applying the filtering in fp_step to each and every tuple in $\mathcal{R}$ in every step, would not be necessary. Only after a tuple (p,q) has been removed from $\mathcal{R}$, the algorithm does really need to find out whether this was the last witness for the $\exists$-quantification in the clause of another tuple. While this observation could inspire various improvements, let us fast-forward to the game-theoretic approach in the next section, which elegantly explicates the witness structure of a coupled similarity problem.

4 Game Algorithm for Coupled Similarity

Checking whether two states are related by a (bi-)simulation preorder $\sqsubseteq_X$ can be seen as a *game* along the lines of coinductive characterizations [30]. One player, the *attacker*, challenges that $p \sqsubseteq_X q$, while the other player, the *defender*, has to name witnesses for the existential quantifications of the definition.

Based on the coinductive characterization from Lemma 3, we here define such a game for the coupled simulation preorder and transform it into an algorithm, which basically only amounts to a more clever way of computing the fixed point of the previous section. We show how this additional layer of abstraction enables optimizations.

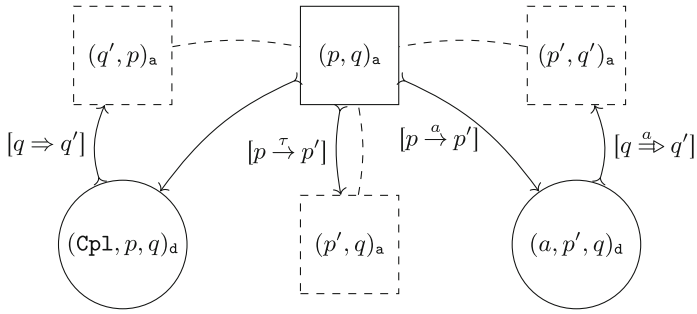


Fig. 5. Schematic coupled simulation game. Boxes stand for attacker nodes, circles for defender nodes, arrows for moves. From the dashed boxes, the moves are analogous to the ones of the solid box.

4.1 The Coupled Simulation Game

The *coupled simulation game* proceeds as follows: For $p \sqsubseteq_{CS} q$, the attacker may question that simulation holds by selecting p' and $a \in \Sigma$ with $p \xrightarrow{a} p'$. The defender then has to name a q' with $q \overset{a}{\Rightarrow} q'$, whereupon the attacker may go on to challenge $p' \sqsubseteq_{CS} q'$. If $p \xrightarrow{\tau} p'$, the attacker can directly skip to question $p' \sqsubseteq_{CS} q$. For coupled simulation, the attacker may moreover demand the defender to name a coupling witness q' with $q \Rightarrow q'$ whereafter $q' \sqsubseteq_{CS} p$ stands to question. If the defender runs out of answers, they lose; if the game continues forever, they win. This can be modeled by a simple game, whose schema is given in Fig. 5, as follows.

Definition 6 (Games). *A* simple game $\mathcal{G}[p_0] = (G, G_d, \rightarrowtail, p_0)$ *consists of*

- *a (countable) set of* game positions G,
  - *partitioned into a set of* defender positions $G_d \subseteq G$
  - *and* attacker positions $G_a := G \setminus G_d$,
- *a graph of* game moves $\rightarrowtail \subseteq G \times G$, *and*
- *an* initial position $p_0 \in G$.

Definition 7 ($\sqsubseteq_{CS}$ game). *For a transition system* $\mathcal{S} = (S, \Sigma_\tau, \rightarrow)$, *the* coupled simulation game $\mathcal{G}^{\mathcal{S}}_{CS}[p_0] = (G, G_d, \rightarrowtail, p_0)$ *consists of*

- attacker nodes $(p, q)_{\mathtt{a}} \in G_a$ *with* $p, q \in S$,
- simulation defender nodes $(a, p, q)_{\mathtt{d}} \in G_d$ *for situations where a simulation challenge for* $a \in \Sigma$ *has been formulated, and*
- coupling defender nodes $(\mathtt{Cpl}, p, q)_{\mathtt{d}} \in G_d$ *when coupling is challenged,*

and five kinds of moves

- simulation challenges $(p, q)_{\mathtt{a}} \rightarrowtail (a, p', q)_{\mathtt{d}}$ *if* $p \xrightarrow{a} p'$ *with* $a \neq \tau$,
- simulation internal moves $(p, q)_{\mathtt{a}} \rightarrowtail (p', q)_{\mathtt{a}}$ *if* $p \xrightarrow{\tau} p'$,

- simulation answers $(a, p', q)_{\mathtt{d}} \rightarrowtail (p', q')_{\mathtt{a}}$ *if* $q \stackrel{a}{\Rightarrow} q'$,
- coupling challenges $(p, q)_{\mathtt{a}} \rightarrowtail (\mathtt{Cpl}, p, q)_{\mathtt{d}}$, *and*
- coupling answers $(\mathtt{Cpl}, p, q)_{\mathtt{d}} \rightarrowtail (q', p)_{\mathtt{a}}$ *if* $q \Rightarrow q'$.

Definition 8 (Plays and wins). *We call the paths* $p_0 p_1 \ldots \in G^\infty$ *with* $p_i \rightarrowtail p_{i+1}$ plays *of* $\mathcal{G}[p_0]$. *The defender* wins *all infinite plays. If a finite play* $p_0 \ldots p_n$ *is stuck, that is, if* $p_n \not\rightarrowtail$, *then the stuck player loses: The defender wins if* $p_n \in G_a$, *and the attacker wins if* $p_n \in G_d$.

Definition 9 (Strategies and winning strategies). *A* defender strategy *is a (usually partial) mapping from initial play fragments to next moves* $f \subseteq \{(p_0 \ldots p_n, p') \mid p_n \in G_d \wedge p_n \rightarrowtail p'\}$. *A play* p *follows a strategy* f *iff, for each move* $p_i \rightarrowtail p_{i+1}$ *with* $p_i \in G_d$, $p_{i+1} = f(p_0 \ldots p_i)$. *If every such play is won by the defender,* f *is a* winning strategy *for the defender. The player with a winning strategy for* $\mathcal{G}[p_0]$ *is said to* win $\mathcal{G}[p_0]$.

Definition 10 (Winning regions and determinacy). *The* winning region W_σ *of player* $\sigma \in \{a, d\}$ *for a game* $\mathcal{G}$ *is the set of states* p_0 *from which player* σ *wins* $\mathcal{G}[p_0]$.

Let us now see that the defender's winning region of $\mathcal{G}^{\mathcal{S}}_{CS}$ indeed corresponds to $\sqsubseteq^{\mathcal{S}}_{CS}$. To this end, we first show how to construct winning strategies for the defender from a coupled simulation, and then establish the opposite direction.

Lemma 10. *Let* $\mathcal{R}$ *be a coupled delay simulation and* $(p_0, q_0) \in \mathcal{R}$. *Then the defender wins* $\mathcal{G}^{\mathcal{S}}_{CS}[(p_0, q_0)_{\mathtt{a}}]$ *with the following positional strategy:*

- *If the current play fragment ends in a simulation defender node* $(a, p', q)_{\mathtt{d}}$, *move to some attacker node* $(p', q')_{\mathtt{a}}$ *with* $(p', q') \in \mathcal{R}$ *and* $q \stackrel{a}{\Rightarrow} q'$;
- *if the current play fragment ends in a coupling defender node* $(\mathtt{Cpl}, p, q)_{\mathtt{d}}$, *move to some attacker node* $(q', p)_{\mathtt{a}}$ *with* $(q', p) \in \mathcal{R}$ *and* $q \Rightarrow q'$.

Lemma 11. *Let* f *be a winning strategy for the defender in* $\mathcal{G}^{\mathcal{S}}_{CS}[(p_0, q_0)_{\mathtt{a}}]$. *Then* $\{(p, q) \mid$ *some* $\mathcal{G}^{\mathcal{S}}_{CS}[(p_0, q_0)_{\mathtt{a}}]$*-play fragment consistent with* f *ends in* $(p, q)_{\mathtt{a}}\}$ *is a coupled delay simulation.*

Theorem 3. *The defender wins* $\mathcal{G}^{\mathcal{S}}_{CS}[(p, q)_{\mathtt{a}}]$ *precisely if* $p \sqsubseteq_{CS} q$.

4.2 Deciding the Coupled Simulation Game

It is well-known that the winning regions of finite simple games can be computed in linear time. Variants of the standard algorithm for this task can be found in [12] and in our implementation [1]. Intuitively, the algorithm first assumes that the defender wins everywhere and then sets off a chain reaction beginning in defender deadlock nodes, which "turns" all the nodes won by the attacker. The algorithm runs in linear time of the game moves because every node can only turn once.

```
1 def game_compute_cs(S):
2 |   G^S_CS = (G, G_a, ↣) := obtain_cs_game(S)
3 |   win := compute_winning_region(G^S_CS)
4 |   R := {(p, q) | (p, q)_a ∈ G_a ∧ win[(p, q)_a] = d}
5 |   return R
```

Algorithm 2: Game algorithm for the coupled simulation preorder $\sqsubseteq_{CS}$.

With such a winning region algorithm for simple games, referred to as compute_winning_region in the following, it is only a matter of a few lines to determine the coupled simulation preorder for a system $\mathcal{S}$ as shown in game_compute_cs in Algorithm 2. One starts by constructing the corresponding game $\mathcal{G}^{\mathcal{S}}_{CS}$ using a function obtain_cs_game, we consider given by Definition 7. Then, one calls compute_winning_region and collects the attacker nodes won by the defender for the result.

Theorem 4. *For a finite labeled transition systems* $\mathcal{S}$, game_compute_cs($\mathcal{S}$) *from Algorithm 2 returns* $\sqsubseteq^{\mathcal{S}}_{CS}$.

Proof. Theorem 3 states that the defender wins $\mathcal{G}^{\mathcal{S}}_{CS}[(p,q)_a]$ exactly if $p \sqsubseteq^{\mathcal{S}}_{CS} q$. As compute_winning_region($\mathcal{G}^{\mathcal{S}}_{CS}$), according to [12], returns where the defender wins, line 4 of Algorithm 2 precisely assigns $\mathcal{R} = \sqsubseteq^{\mathcal{S}}_{CS}$.

The complexity arguments from [12] yield linear complexity for deciding the game by compute_winning_region.

Proposition 1. *For a game* $\mathcal{G} = (G, G_a, \rightarrowtail)$, compute_winning_region *runs in* $\mathcal{O}(|G| + |\rightarrowtail|)$ *time and space.*

In order to tell the overall complexity of the resulting algorithm, we have to look at the size of $\mathcal{G}^{\mathcal{S}}_{CS}$ depending on the size of $\mathcal{S}$.

Lemma 12. *Consider the coupled simulation game* $\mathcal{G}^{\mathcal{S}}_{CS} = (G, G_a, \rightarrowtail)$ *for varying* $\mathcal{S} = (S, \Sigma_\tau, \rightarrow)$. *The growth of the game size* $|G| + |\rightarrowtail|$ *is in* $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$.

Proof. Let us reexamine Definition 7. There are $|S|^2$ attacker nodes. Collectively, they can formulate $\mathcal{O}(|\dot{\rightarrow}|\,|S|)$ simulation challenges including internal moves and $|S|^2$ coupling challenges. There are $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$ simulation answers and $\mathcal{O}(|\Rightarrow|\,|S|)$ coupling answers. Of these, $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$ dominates the others.

Lemma 13. game_compute_cs *runs in* $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$ *time and space.*

Proof. Proposition 1 and Lemma 12 already yield that line 3 is in $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$ time and space. Definition 7 is completely straight-forward, so the complexity of building $\mathcal{G}^{\mathcal{S}}_{CS}$ in line 2 equals its output size $\mathcal{O}(|\dot{\Rightarrow}|\,|S|)$, which coincides with the complexity of computing $\dot{\Rightarrow}$. The filtering in line 4 is in $\mathcal{O}(|S|^2)$ (upper bound for attacker nodes) and thus does not influence the overall complexity.

4.3 Tackling the τ-closure

We have mentioned that there can be some complexity to computing the τ-closure $\Rightarrow = \xrightarrow{\tau}{}^{*}$ and the derived $\dot{\Rightarrow}$. In theory, both the weak delay transition relation $\dot{\Rightarrow}$ and the conventional transition relation $\dot{\rightarrow}$ are bounded in size by $|\Sigma_\tau|\,|S|^2$. But for most transition systems, the weak step relations tend to be much bigger in size. Sparse $\dot{\rightarrow}$-graphs can generate dense $\dot{\Rightarrow}$-graphs. The computation of the transitive closure also has significant time complexity. Algorithms for transitive closures usually are cubic, even though the theoretical bound is a little lower.

There has been a trend to skip the construction of the transitive closure in the computation of weak forms of bisimulation [3,13,19,26]. With the game approach, we can follow this trend. The transitivity of the game can emulate the transitivity of $\dot{\Rightarrow}$ (for details see [1, Sec. 4.5.4]). With this trick, the game size, and thus time and space complexity, reduces to $\mathcal{O}(|\Sigma_\tau|\,|\xrightarrow{\tau}|\,|S| + |\dot{\rightarrow}|\,|S|)$. Though this is practically better than the bound from Lemma 13, both results amount to cubic complexity $\mathcal{O}(|\Sigma|\,|S|^3)$, which is in line with the reduction result from Theorem 1 and the time complexity of existing similarity algorithms.

4.4 Optimizing the Game Algorithm

The game can be downsized tremendously once we take additional over- and under-approximation information into account.

Definition 11. *An* over-approximation *of $\sqsubseteq_{CS}$ is a relation $\mathcal{R}_O$ of that we know that $\sqsubseteq_{CS} \subseteq \mathcal{R}_O$. Conversely, an* under-approximation *of $\sqsubseteq_{CS}$ is a relation $\mathcal{R}_U$ where $\mathcal{R}_U \subseteq \sqsubseteq_{CS}$.*

Regarding the game, over-approximations tell us where the defender *can* win, and under-approximations tell us where the attacker is doomed to lose. They can be used to eliminate "boring" parts of the game. Given an over-approximation $\mathcal{R}_O$, when unfolding the game, it only makes sense to add moves from defender nodes to attacker nodes $(p, q)_{\mathtt{a}}$ if $(p, q) \in \mathcal{R}_O$. There just is no need to allow the defender moves we already know cannot be winning for them. Given an under-approximation $\mathcal{R}_U$, we can ignore all the outgoing moves of $(p, q)_{\mathtt{a}}$ if $(p, q) \in \mathcal{R}_U$. Without moves, $(p, q)_{\mathtt{a}}$ is sure to be won by the defender, which is in line with the claim of the approximation.

Corollary 2. *$\Rightarrow^{-1}$ is an under-approximation of $\sqsubseteq_{CS}$. (Cf. Lemma 4)*

Lemma 14. *$\{(p, q) \mid$ all actions weakly enabled in p are weakly enabled in $q\}$ is an over-approximation of $\sqsubseteq_{CS}$.*

The fact that coupled simulation is "almost bisimulation" on steps to stable states in finite systems (Lemma 6) can be used for a comparably cheap and precise over-approximation. The idea is to compute strong bisimilarity for the system $\mathcal{S}_{\Rightarrow|} = (S, \Sigma_\tau, \Rightarrow|)$, where *maximal weak steps*, $p \overset{\alpha}{\Rightarrow}| p'$, exist iff $p \overset{\hat{\alpha}}{\Rightarrow} p'$ and p' is stable, that is, $p' \overset{\tau}{\not\rightarrow}$. Let $\equiv_{\Rightarrow|}$ be the biggest symmetric relation where $p \equiv_{\Rightarrow|} q$ and $p \overset{\alpha}{\Rightarrow}| p'$ implies there is q' such that $p' \equiv_{\Rightarrow|} q'$ and $q \overset{\alpha}{\Rightarrow}| q'$.

Lemma 15. *$\mathcal{R}_{\Rightarrow|} = \{(p,q) \mid \forall p'.\, p \overset{\alpha}{\Rightarrow}| p' \longrightarrow q \overset{\alpha}{\Rightarrow}| \equiv_{\Rightarrow|} p'\}$ is an over-approximation of $\sqsubseteq_{CS}$ on finite systems.*

Computing $\equiv_{\Rightarrow|}$ can be expected to be cheaper than computing weak bisimilarity $\equiv_{WB}$. After all, $\Rightarrow|$ is just a subset of $\hat{\Rightarrow}$. However, filtering $S \times S$ using subset checks to create $\mathcal{R}_{\Rightarrow|}$ might well be *quartic,* $\mathcal{O}(|S|^4)$, or worse. Nevertheless, one can argue that with a reasonable algorithm design and for many real-world examples, $\overset{\alpha}{\Rightarrow}|\equiv_{\Rightarrow|}$ will be sufficiently bounded in branching degree, in order for the over-approximation to do more good than harm.

For everyday system designs, $\mathcal{R}_{\Rightarrow|}$ is a tight approximation of $\sqsubseteq_{CS}$. On the philosopher system from Example 1, they even coincide. In some situations, $\mathcal{R}_{\Rightarrow|}$ degenerates to the shared enabledness relation (Lemma 14), which is to say it becomes comparably useless. One example for this are the systems created by the reduction from weak simulation to coupled simulation in Theorem 1 after τ-cycle removal. There, all $\Rightarrow|$-steps are bound to end in the same one τ-sink state $\perp$.

5 A Scalable Implementation

The experimental results by Ranzato and Tapparo [27] suggest that their simulation algorithm and the algorithm by Henzinger, Henzinger, and Kopke [15] only work on comparably small systems. The necessary data structures quickly consume gigabytes of RAM. So, the bothering question is not so much whether some highly optimized C++-implementation can do the job in milliseconds for small problems, but how to implement the algorithm such that large-scale systems are feasible at all.

To give first answers, we implemented a scalable and distributable prototype of the coupled simulation game algorithm using the stream processing framework *Apache Flink* [4] and its *Gelly* graph API, which enable computations on large data sets built around a universal data-flow engine. Our implementation can be found on https://coupledsim.bbisping.de/code/flink/.

5.1 Prototype Implementation

We base our implementation on the game algorithm and optimizations from Sect. 4. The implementation is a vertical prototype in the sense that every feature to get from a transition system to its coupled simulation preorder is present, but there is no big variety of options in the process. The phases are:

Import Reads a CSV representation of the transition system $\mathcal{S}$.

Minimize Computes an equivalence relation under-approximating $\equiv_{CS}$ on the transition system and builds a quotient system $\mathcal{S}_M$. This stage should at least compress τ-cycles if there are any. The default minimization uses a parallelized signature refinement algorithm [20,33] to compute delay bisimilarity ($\equiv_{DB}^{\mathcal{S}}$).

Table 1. Sample systems, sizes, and benchmark results.

system	S	$\dot{\rightarrow}$	$\dot{\Rightarrow}$	$S_{/\equiv_{DB}}$	$\rightarrowtail$	$\rightarrowtail_{\sigma.}$	$S_{/\equiv_{CS}}$	$\sqsubseteq_{CS}^{\mathcal{S}_{/\equiv_{CS}}}$	time/s
`phil`	10	14	86	6	234	201	5	11	5.1
`ltbts`	88	98	2,599	27	4,100	399	25	38	5.5
`vasy_0_1`	289	1,224	52,641	9	543	67	9	9	5.7
`vasy_1_4`	1,183	4,464	637,585	4	73	30	4	4	5.3
`vasy_5_9`	5,486	9,676	1,335,325	112	63,534	808	112	112	6.0
`cwi_1_2`	1,952	2,387	593,734	67	29,049	1,559	67	137	6.9
`cwi_3_14`	3,996	14,552	15,964,021	2	15	10	2	2	7.8
`vasy_8_24`	8,879	24,411	2,615,500	170	225,555	3,199	169	232	6.7
`vasy_8_38`	8,921	38,424	46,232,423	193	297,643	2,163	193	193	6.7
`vasy_10_56`	10,849	56,156	842,087	2,112	o.o.m.	72,617	2,112	3,932	13.8
`vasy_25_25`	25,217	25,216	50,433	25,217	o.o.m.	126,083	25,217	25,217	117.4

Compute over-approximation Determines an equivalence relation over-approximating $\equiv_{CS}^{\mathcal{S}_M}$. The result is a mapping σ from states to *signatures* (sets of colors) such that $p \sqsubseteq_{CS}^{\mathcal{S}_M} q$ implies $\sigma(p) \subseteq \sigma(q)$. The prototype uses the maximal weak step equivalence $\equiv_{\Rightarrow|}$ from Subsect. 4.4.

Build game graph Constructs the τ-closure-free coupled simulation game $\mathcal{G}_{CS}^{\mathcal{S}_M}$ for $\mathcal{S}_M$ with attacker states restricted according to the over-approximation signatures σ.

Compute winning regions Decides for $\mathcal{G}_{CS}^{\mathcal{S}_M}$ where the attacker has a winning strategy following the scatter-gather scheme [16]. If a game node is discovered to be won by the attacker, it *scatters* the information to its predecessors. Every game node *gathers* information on its winning successors. Defender nodes count down their degrees of freedom starting at their game move outdegrees.

Output Finally, the results can be output or checked for soundness. The winning regions directly imply $\sqsubseteq_{CS}^{\mathcal{S}_M}$. The output can be de-minimized to refer to the original system $\mathcal{S}$.

5.2 Evaluation

Experimental evaluation shows that the approach can cope with the smaller examples of the "Very Large Transition Systems (VLTS) Benchmark Suite" [6] (`vasy_*` and `cwi_*` up to 50,000 transitions). On small examples, we also tested that the output matches the return values of the verified fixed-point $\sqsubseteq_{CS}$-algorithm from Sect. 3. These samples include, among others, the philosopher system `phil` containing P_g and P_o from Example 1 and `ltbts`, which consists of the finitary separating examples from the linear-time branching-time spectrum [9, p. 73].

Table 1 summarizes the results for some of our test systems with preminimization by delay bisimilarity and over-approximation by maximal weak step equivalence. The first two value columns give the system sizes in number of states

S and transitions $\dot{\rightarrow}$. The next two columns present derived properties, namely an upper estimate of the size of the (weak) delay step relation $\dot{\Rightarrow}$, and the number of partitions with respect to delay bisimulation $S_{/\equiv_{DB}}$. The next columns list the sizes of the game graphs without and with maximal weak step over-approximation ($\rightarrowtail$ and $\rightarrowtail_\sigma$, some tests without the over-approximation trick ran out of memory, "o.o.m."). The following columns enumerate the sizes of the resulting coupled simulation preorders represented by the partition relation pair $(S_{/\equiv_{CS}}, \sqsubseteq_{CS}^{\mathcal{S}/\equiv_{CS}})$, where $S_{/\equiv_{CS}}$ is the partitioning of S with respect to coupled similarity $\equiv_{CS}$, and $\sqsubseteq_{CS}^{\mathcal{S}/\equiv_{CS}}$ the coupled simulation preorder projected to this quotient. The last column reports the running time of the programs on an Intel i7-8550U CPU with four threads and 2 GB Java Virtual Machine heap space.

The systems in Table 1 are a superset of the VLTS systems for which Ranzato and Tapparo [27] report their algorithm *SA* to terminate. Regarding complexity, SA is the best simulation algorithm known. In the [27]-experiments, the C++ implementation ran out of 2 GB RAM for `vasy_10_56` and `vasy_25_25` but finished much faster than our setup for most smaller examples. Their time advantage on small systems comes as no surprise as the start-up of the whole Apache Flink pipeline induces heavy overhead costs of about 5 s even for tiny examples like `phil`. However, on bigger examples such as `vasy_18_73` their and our implementation both fail. This is in stark contrast to *bi*-simulation implementations, which usually cope with much larger systems single-handedly [3,19].

Interestingly, for all tested VLTS systems, the weak bisimilarity quotient system $S_{/\equiv_{WB}}$ equals $S_{/\equiv_{CS}}$ (and, with the exception of `vasy_8_24`, $S_{/\equiv_{DB}}$). The preorder $\sqsubseteq_{CS}^{\mathcal{S}/\equiv_{CS}}$ also matches the identity in 6 of 9 examples. This observation about the effective closeness of coupled similarity and weak bisimilarity is two-fold. On the one hand, it brings into question how meaningful coupled similarity is for minimization. After all, it takes a lot of space and time to come up with the output that the cheaper delay bisimilarity already minimized everything that could be minimized. On the other hand, the observation suggests that the considered VLTS samples are based around models that do not need—or maybe even do avoid—the expressive power of weak bisimilarity. This is further evidence for the case from the introduction that coupled similarity has a more sensible level of precision than weak bisimilarity.

6 Conclusion

The core of this paper has been to present a game-based algorithm to compute coupled similarity in cubic time and space. To this end, we have formalized coupled similarity in Isabelle/HOL and merged two previous approaches to defining coupled similarity, namely using single relations with weak symmetry [10] and the relation-pair-based coupled delay simulation from [28], which followed the older tradition of two weak simulations [24,29]. Our characterization seems to be the most convenient. We used the entailed coinductive characterization to devise a game characterization and an algorithm. Although we could show that deciding

coupled similarity is as hard as deciding weak similarity, our Apache Flink implementation is able to exploit the closeness between coupled similarity and weak bisimilarity to at least handle slightly bigger systems than comparable similarity algorithms. Through the application to the VLTS suite, we have established that coupled similarity and weak bisimilarity match for the considered systems. This points back to a line of thought [11] that, for many applications, branching, delay and weak bisimilarity will coincide with coupled similarity. Where they do not, usually coupled similarity or a coarser notion of equivalence is called for. To gain deeper insights in that direction, real-world case studies—and maybe an embedding into existing tool landscapes like FDR [8], CADP [7], or LTSmin [17]—would be necessary.

References

1. Bisping, B.: Computing coupled similarity. Master's thesis, Technische Universität Berlin (2018). https://coupledsim.bbisping.de/bisping_computingCoupledSimilarity_thesis.pdf
2. Bisping, B.: Isabelle/HOL proof and Apache Flink program for TACAS 2019 paper: Computing Coupled Similarity (artifact). Figshare (2019). https://doi.org/10.6084/m9.figshare.7831382.v1
3. Boulgakov, A., Gibson-Robinson, T., Roscoe, A.W.: Computing maximal weak and other bisimulations. Formal Aspects Comput. **28**(3), 381–407 (2016). https://doi.org/10.1007/s00165-016-0366-2
4. Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., Tzoumas, K.: Apache Flink: stream and batch processing in a single engine. In: Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, vol. 36, no. 4 (2015)
5. Derrick, J., Wehrheim, H.: Using coupled simulations in non-atomic refinement. In: Bert, D., Bowen, J.P., King, S., Waldén, M. (eds.) ZB 2003. LNCS, vol. 2651, pp. 127–147. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44880-2_10
6. Garavel, H.: The VLTS benchmark suite (2017). https://doi.org/10.18709/perscido.2017.11.ds100. Jointly created by CWI/SEN2 and INRIA/VASY as a CADP resource
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. Int. J. Softw. Tools Technol. Transfer **15**(2), 89–107 (2013). https://doi.org/10.1007/s10009-012-0244-z
8. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — a modern refinement checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 187–201. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13
9. van Glabbeek, R.J.: The linear time — branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57208-2_6
10. van Glabbeek, R.J.: A branching time model of CSP. In: Gibson-Robinson, T., Hopcroft, P., Lazić, R. (eds.) Concurrency, Security, and Puzzles. LNCS, vol. 10160, pp. 272–293. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-51046-0_14

11. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM (JACM) **43**(3), 555–600 (1996). https://doi.org/10.1145/233551.233556
12. Grädel, E.: Finite model theory and descriptive complexity. In: Grädel, E., et al. (eds.) Finite Model Theory and Its Applications. Texts in Theoretical Computer Science an EATCS Series, pp. 125–130. Springer, Heidelberg (2007). https://doi.org/10.1007/3-540-68804-8_3
13. Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.J.: An $\mathcal{O}(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. ACM Trans. Comput. Logic (TOCL) **18**(2), 13:1–13:34 (2017). https://doi.org/10.1145/3060140
14. Hatzel, M., Wagner, C., Peters, K., Nestmann, U.: Encoding CSP into CCS. In: Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS, pp. 61–75 (2015). https://doi.org/10.4204/EPTCS.190.5
15. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, pp. 453–462 (1995). https://doi.org/10.1109/SFCS.1995.492576
16. Kalavri, V., Vlassov, V., Haridi, S.: High-level programming abstractions for distributed graph processing. IEEE Trans. Knowl. Data Eng. **30**(2), 305–324 (2018). https://doi.org/10.1109/TKDE.2017.2762294
17. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
18. Kučera, A., Mayr, R.: Why is simulation harder than bisimulation? In: Brim, L., Křetínský, M., Kučera, A., Jančar, P. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 594–609. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45694-5_39
19. Li, W.: Algorithms for computing weak bisimulation equivalence. In: Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 2009. TASE 2009, pp. 241–248. IEEE (2009). https://doi.org/10.1109/TASE.2009.47
20. Luo, Y., de Lange, Y., Fletcher, G.H.L., De Bra, P., Hidders, J., Wu, Y.: Bisimulation reduction of big graphs on MapReduce. In: Gottlob, G., Grasso, G., Olteanu, D., Schallhart, C. (eds.) BNCOD 2013. LNCS, vol. 7968, pp. 189–203. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39467-6_18
21. Milner, R.: Communication and Concurrency. Prentice-Hall Inc., Upper Saddle River (1989)
22. Nestmann, U., Pierce, B.C.: Decoding choice encodings. Inf. Comput. **163**(1), 1–59 (2000). https://doi.org/10.1006/inco.2000.2868
23. Parrow, J., Sjödin, P.: Multiway synchronization verified with coupled simulation. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 518–533. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0084813
24. Parrow, J., Sjödin, P.: The complete axiomatization of Cs-congruence. In: Enjalbert, P., Mayr, E.W., Wagner, K.W. (eds.) STACS 1994. LNCS, vol. 775, pp. 555–568. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-57785-8_171
25. Peters, K., van Glabbeek, R.J.: Analysing and comparing encodability criteria. In: Proceedings of the Combined 22th International Workshop on Expressiveness in Concurrency and 12th Workshop on Structural Operational Semantics, EXPRESS/SOS, pp. 46–60 (2015). https://doi.org/10.4204/EPTCS.190.4

26. Ranzato, F., Tapparo, F.: Generalizing the Paige-Tarjan algorithm by abstract interpretation. Inf. Comput. **206**(5), 620–651 (2008). https://doi.org/10.1016/j.ic.2008.01.001. Special Issue: The 17th International Conference on Concurrency Theory (CONCUR 2006)
27. Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Inf. Comput. **208**(1), 1–22 (2010). https://doi.org/10.1016/j.ic.2009.06.002
28. Rensink, A.: Action contraction. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 290–305. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_22
29. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press, New York (2012). https://doi.org/10.1017/CBO9780511777110
30. Stirling, C.: Modal and Temporal Properties of Processes. Springer, New York (2001). https://doi.org/10.1007/978-1-4757-3550-5
31. Voorhoeve, M., Mauw, S.: Impossible futures and determinism. Inf. Process. Lett. **80**(1), 51–58 (2001). https://doi.org/10.1016/S0020-0190(01)00217-4
32. Wenzel, M.: The Isabelle/Isar Reference Manual (2018). https://isabelle.in.tum.de/dist/Isabelle2018/doc/isar-ref.pdf
33. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref – a symbolic bisimulation tool box. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 477–492. Springer, Heidelberg (2006). https://doi.org/10.1007/11901914_35

Reachability Analysis for Termination and Confluence of Rewriting

Christian Sternagel[1(✉)] and Akihisa Yamada[2(✉)]

[1] University of Innsbruck, Innsbruck, Austria
christian.sternagel@uibk.ac.at
[2] National Institute of Informatics, Tokyo, Japan
akihisayamada@nii.ac.jp

Abstract. In term rewriting, reachability analysis is concerned with the problem of deciding whether or not one term is reachable from another by rewriting. Reachability analysis has several applications in termination and confluence analysis of rewrite systems. We give a unified view on reachability analysis for rewriting with and without conditions by means of what we call reachability constraints. Moreover, we provide several techniques that fit into this general framework and can be efficiently implemented. Our experiments show that these techniques increase the power of existing termination and confluence tools.

Keywords: Reachability analysis · Termination · Confluence · Conditional term rewriting · Infeasibility

1 Introduction

Reachability analysis for term rewriting [6] is concerned with the problem of, given a rewrite system $\mathcal{R}$, a source term s and a target term t, deciding whether the source reduces to the target by rewriting, which is usually written $s \to_{\mathcal{R}}^* t$. A useful generalization of this problem is the (un)satisfiability of the following reachability problem: given terms s and t containing variables, decide whether there is a substitution σ such that $s\sigma \to_{\mathcal{R}}^* t\sigma$ or not. This problem, also called (in)feasibility by Lucas and Guitiérrez [11], has various applications in termination and confluence analysis for plain and conditional rewriting.

This can be understood as a form of safety analysis, as illustrated below.

Example 1. Let $\mathcal{R}$ be a term rewrite system consisting of the following rules for division (where s stands for "successor"):

$$x - \mathsf{0} \to x \qquad \mathsf{s}(x) - \mathsf{s}(y) \to x - y \qquad \mathsf{0} \div \mathsf{s}(y) \to \mathsf{0}$$
$$\mathsf{s}(x) \div \mathsf{s}(y) \to \mathsf{s}((x - y) \div \mathsf{s}(y)) \qquad x \div \mathsf{0} \to \mathsf{err}(\texttt{"division by zero"})$$

The question "Can division yield an error?" is naturally formulated as the satisfiability of reachability from $x \div y$ to $\mathsf{err}(z)$. Unsurprisingly, the solution

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 262–278, 2019.
https://doi.org/10.1007/978-3-030-17462-0_15

$$\sigma = [y \mapsto 0, z \mapsto \texttt{"division by zero"}]$$

shows that it is actually possible to obtain an error.

In termination analysis we are typically interested in unsatisfiability of reachability and can thereby rule out certain recursive calls as potential source of nontermination. For confluence analysis of conditional term rewriting, infeasibility is crucial: some other techniques do not apply before critical pairs are shown infeasible, and removal of infeasible rules simplifies proofs.

In this work we provide a formal framework that allows us to uniformly speak about (un)satisfiability of reachability for plain and conditional rewriting, and give several techniques that are useful in practice.

More specifically, our contributions are as follows:

- We introduce the syntax and semantics of *reachability constraints* (Sect. 3) and formulate their satisfiability problem. We recast several concrete techniques for reachability analysis in the resulting framework.
- We present a new, simple, and efficient technique for reachability analysis based on what we call the *symbol transition graph* of a rewrite system (Sect. 4.1) and extend it to conditional rewriting (Sect. 5.2).
- Additionally, we generalize the prevalent existing technique for term rewriting to what we call *look-ahead reachability* (Sect. 4.2) and extend it to the conditional case (Sect. 5.3).
- Then, we present a new result for conditional rewriting that is useful for proving conditional rules infeasible (Sect. 5.1).
- Finally, we evaluate the impact of our work on existing automated tools NaTT [16] and ConCon [13] (Sect. 6).

2 Preliminaries

In the remainder, we assume some familiarity with term rewriting. Nevertheless, we recall required concepts and notations below. For further details on term rewriting, we refer to standard textbooks [3,14].

Throughout the paper $\mathcal{F}$ denotes a set of function symbols with associated arities, and $\mathcal{V}$ a countably infinite set of variables (so that fresh variables can always be picked) such that $\mathcal{F} \cap \mathcal{V} = \varnothing$. A *term* is either a variable $x \in \mathcal{V}$ or of the form $f(t_1, \ldots, t_n)$, where n is the arity of $f \in \mathcal{F}$ and the arguments $t_1, \ldots, t_n$ are terms. The set of all terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of variables occurring in a term t is denoted by $\mathsf{Var}(t)$. The *root symbol* of a term $t = f(t_1, \ldots, t_n)$ is f and denoted by $\mathsf{root}(t)$. When we want to indicate that a term is not a variable, we sometimes write $f(...)$, where "..." denotes an arbitrary list of terms.

A *substitution* is a mapping $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$. Given a term t, $t\sigma$ denotes the term obtained by replacing every occurrence of variable x in t by $\sigma(x)$. The *domain* of a substitution σ is $\mathsf{Dom}(\sigma) := \{x \in \mathcal{V} \mid x\sigma \neq x\}$, and σ is *idempotent* if $\mathsf{Var}(x\sigma) \cap \mathsf{Dom}(\sigma) = \varnothing$ for every $x \in \mathcal{V}$. A *renaming* is a bijection $\alpha : \mathcal{V} \to \mathcal{V}$.

Two terms s and t are *unifiable* if $s\sigma = t\sigma$ for some substitution σ, which is called a *unifier* of s and t.

A *context* is a term with exactly one occurrence of the special symbol $\Box$. We write $C[t]$ for the term resulting from replacing $\Box$ in context C by term t.

A *rewrite rule* is a pair of terms, written $l \to r$, such that the *variable conditions* $l \notin \mathcal{V}$ and $\mathsf{Var}(l) \supseteq \mathsf{Var}(r)$ hold. By a *variant* of a rewrite rule we mean a rule that is obtained by consistently renaming variables in the original rule to fresh ones. A *term rewrite system (TRS)* is a set $\mathcal{R}$ of rewrite rules. A function symbol $f \in \mathcal{F}$ is *defined* in $\mathcal{R}$ if $f(...) \to r \in \mathcal{R}$, and the set of defined symbols in $\mathcal{R}$ is $\mathcal{D}_\mathcal{R} := \{f \mid f(...) \to r \in \mathcal{R}\}$. We call $f \in \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$ a *constructor*.

There is an $\mathcal{R}$-*rewrite step* from s to t, written $s \to_\mathcal{R} t$, iff there exist a context C, a substitution σ, and a rule $l \to r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. We write $s \xrightarrow{\epsilon}_\mathcal{R} t$ if $C = \Box$ (called a *root step*), and $s \xrightarrow{>\epsilon}_\mathcal{R} t$ (called a *non-root step*), otherwise. We say a term s_0 is $\mathcal{R}$-*terminating* if it starts no infinite rewrite sequence $s_0 \to_\mathcal{R} s_1 \to_\mathcal{R} s_2 \to_\mathcal{R} \cdots$, and say $\mathcal{R}$ is *terminating* if every term is $\mathcal{R}$-terminating.

For a relation $\rightarrowtail \subseteq A \times A$, we denote its transitive closure by $\rightarrowtail^+$ and reflexive transitive closure by $\rightarrowtail^*$. We say that $a_1, \ldots, a_n \in A$ are *joinable* (*meetable*) at $b \in A$ with respect to $\rightarrowtail$ if $a_i \rightarrowtail^* b$ ($b \rightarrowtail^* a_i$) for every $i \in \{1, \ldots, n\}$.

3 Reachability Constraint Satisfaction

In this section we introduce the syntax and semantics of reachability constraints, a framework that allows us to unify several concrete techniques for reachability analysis on an abstract level. Reachability constraints are first-order formulas[1] with a single binary predicate symbol whose intended interpretation is reachability by rewriting with respect to a given rewrite system.

Definition 1 (Reachability Constraints). Reachability constraints *are given by the following grammar (where* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ *and* $x \in \mathcal{V}$*)*

$$\phi, \psi, \cdots ::= \top \mid \bot \mid s \twoheadrightarrow t \mid \phi \vee \psi \mid \phi \wedge \psi \mid \neg\phi \mid \forall x.\, \phi \mid \exists x.\, \phi$$

To save some space, we use conventional notation like $\bigwedge_{i \in I} \phi_i$ and $\exists x_1, \ldots, x_n.\, \phi$.

As mentioned above, the semantics of reachability constraints is defined with respect to a given rewrite system. In the following we define satisfiability of constraints with respect to a TRS. (This definition will be extended to conditional rewrite systems in Sect. 5).

Definition 2 (Satisfiability). *We define*[2] *inductively when a substitution* σ satisfies *a reachability constraint* ϕ *modulo a TRS* $\mathcal{R}$*, written* $\sigma \models_\mathcal{R} \phi$*, as follows:*

[1] While in general we allow an arbitrary first-order logical structure for formulas, for the purpose of this paper, negation and universal quantification are not required.

[2] It is also possible to give a model-theoretic account for these notions. However, the required preliminaries are outside the scope of this paper.

- $\sigma \models_{\mathcal{R}} \top$;
- $\sigma \models_{\mathcal{R}} s \twoheadrightarrow t$ *if* $s\sigma \to_{\mathcal{R}}^{*} t\sigma$;
- $\sigma \models_{\mathcal{R}} \phi \vee \psi$ *if* $\sigma \models_{\mathcal{R}} \phi$ *or* $\sigma \models_{\mathcal{R}} \psi$;
- $\sigma \models_{\mathcal{R}} \phi \wedge \psi$ *if* $\sigma \models_{\mathcal{R}} \phi$ *and* $\sigma \models_{\mathcal{R}} \psi$;
- $\sigma \models_{\mathcal{R}} \neg\phi$ *if* $\sigma \models_{\mathcal{R}} \phi$ *does not hold;*
- $\sigma \models_{\mathcal{R}} \forall x.\, \phi$ *if* $\sigma' \models_{\mathcal{R}} \phi$ *for every* σ' *that coincides with* σ *on* $\mathcal{V} \setminus \{x\}$.
- $\sigma \models_{\mathcal{R}} \exists x.\, \phi$ *if* $\sigma' \models_{\mathcal{R}} \phi$ *for some* σ' *that coincides with* σ *on* $\mathcal{V} \setminus \{x\}$.

We say ϕ *and* ψ *are* equivalent *modulo* $\mathcal{R}$*, written* $\phi \equiv_{\mathcal{R}} \psi$*, when* $\sigma \models_{\mathcal{R}} \phi$ *iff* $\sigma \models_{\mathcal{R}} \psi$ *for all* σ*. We say* ϕ *and* ψ *are* (logically) equivalent*, written* $\phi \equiv \psi$*, if they are equivalent modulo any* $\mathcal{R}$*. We say* ϕ *is* satisfiable *modulo* $\mathcal{R}$*, written* $\mathsf{SAT}_{\mathcal{R}}(\phi)$*, if there is a substitution* σ *that satisfies* ϕ *modulo* $\mathcal{R}$*, and call* σ *a* solution *of* ϕ *with respect to* $\mathcal{R}$.

Checking for satisfiability of reachability constraints is for example useful for proving termination of term rewrite systems via the *dependency pair method* [2], or more specifically in *dependency graph* analysis. For the dependency pair method, we assume a fresh *marked* symbol $f^{\sharp}$ for every $f \in \mathcal{D}_{\mathcal{R}}$, and write $s^{\sharp}$ to denote the term $f^{\sharp}(s_1, \ldots, s_n)$ for $s = f(s_1, \ldots, s_n)$. The set of *dependency pairs* of a TRS $\mathcal{R}$ is $\mathsf{DP}(\mathcal{R}) := \{ l^{\sharp} \to r^{\sharp} \mid l \to C[r] \in \mathcal{R},\ r \notin \mathcal{V},\ \mathsf{root}(r) \in \mathcal{D}_{\mathcal{R}} \}$. The standard definition of the dependency graph of a TRS [2] can be recast using reachability constraints as follows:

Definition 3 (Dependency Graph). *Given a TRS* $\mathcal{R}$*, its* dependency graph $\mathsf{DG}(\mathcal{R})$ *is the directed graph over* $\mathsf{DP}(\mathcal{R})$ *where there is an edge from* $l^{\sharp} \to s^{\sharp}$ *to* $t^{\sharp} \to r^{\sharp}$ *iff* $\mathsf{SAT}_{\mathcal{R}}(s^{\sharp} \twoheadrightarrow t^{\sharp}\alpha)$*, where* α *is a renaming of variables such that* $\mathsf{Var}(t^{\sharp}\alpha) \cap \mathsf{Var}(s^{\sharp}) = \varnothing$.

The nodes of the dependency graph correspond to the possible recursive calls in a program (represented by a TRS), while its edges encode the information which recursive calls can directly follow each other in arbitrary program executions. This is the reason why dependency graphs are useful for investigating the termination behavior of TRSs, as captured by the following result.

Theorem 1 ([10]). *A TRS* $\mathcal{R}$ *is terminating iff for every strongly connected component* $\mathcal{C}$ *of an over approximation of* $\mathsf{DG}(\mathcal{R})$*, there is no infinite chain* $s_0 \xrightarrow{\epsilon}_{\mathcal{C}} t_0 \to_{\mathcal{R}}^{*} s_1 \xrightarrow{\epsilon}_{\mathcal{C}} t_1 \to_{\mathcal{R}}^{*} \cdots$ *where every* t_i *is* $\mathcal{R}$*-terminating.*

Example 2. Consider the TRS $\mathcal{R}$ of Toyama [15] consisting of the single rule $\mathsf{f}(\mathsf{0}, \mathsf{1}, x) \to \mathsf{f}(x, x, x)$. Its dependency graph $\mathsf{DG}(\mathcal{R})$ consists of the single node:

$$\mathsf{f}^{\sharp}(\mathsf{0}, \mathsf{1}, x) \to \mathsf{f}^{\sharp}(x, x, x) \tag{1}$$

To show $\mathcal{R}$ terminates it suffices to show that $\mathsf{DG}(\mathcal{R})$ has no edge from (1) back to (1), that is, the unsatisfiability of the constraint (with a fresh variable x')

$$\mathsf{f}^{\sharp}(x, x, x) \twoheadrightarrow \mathsf{f}^{\sharp}(\mathsf{0}, \mathsf{1}, x') \tag{2}$$

The most popular method today for checking reachability during dependency graph analysis is unifiability between the target and an approximation of the topmost part of the source (its "cap") that does not change under rewriting, which is computed by the $\mathsf{tcap}_{\mathcal{R}}$ function [9].

Definition 4 (tcap). *Let $\mathcal{R}$ be a TRS. We recursively define $\mathsf{tcap}_{\mathcal{R}}(t)$ for a given term t as follows: $\mathsf{tcap}_{\mathcal{R}}(x)$ is a fresh variable if $x \in \mathcal{V}$; $\mathsf{tcap}_{\mathcal{R}}(f(t_1, \ldots, t_n))$ is a fresh variable if $u = f(\mathsf{tcap}_{\mathcal{R}}(t_1), \ldots, \mathsf{tcap}_{\mathcal{R}}(t_n))$ unifies with some left-hand side of the rules in $\mathcal{R}$; otherwise, it is u.*

The standard way of checking for nonreachability that is implemented in most tools is captured by of the following proposition.

Proposition 1. *If $\mathsf{tcap}_{\mathcal{R}}(s)$ and t are not unifiable, then $s \twoheadrightarrow t \equiv_{\mathcal{R}} \bot$.*

Example 3. Proposition 1 cannot prove the unsatisfiability of (2) of Example 2, since the term cap of the source $\mathsf{tcap}_{\mathcal{R}}(\mathsf{f}^\sharp(x, x, x)) = \mathsf{f}^\sharp(z, z', z'')$, where z, z', z'' are fresh variables, is unifiable with the target $\mathsf{f}^\sharp(0, 1, x')$.

4 Reachability in Term Rewriting

In this section we introduce some techniques for analyzing (un)satisfiability of reachability constraints. The first one described below formulates an obvious observation: no root rewrite step is applicable when starting from a term whose root is a constructor.

Definition 5 (Non-Root Reachability). *For terms $s = f(\ldots)$ and $t = g(\ldots)$, we define the* non-root *reachability constraint $s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t$ as follows:*

- $s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t = \bot$ *if $f \neq g$, and*
- $f(s_1, \ldots, s_n) \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow f(t_1, \ldots, t_n) = s_1 \twoheadrightarrow t_1 \wedge \ldots \wedge s_n \twoheadrightarrow t_n$.

The intention of non-root reachability constraints is to encode zero or more steps of non-root rewriting, in the following sense.

Lemma 1. *For $s, t \notin \mathcal{V}$, $s\sigma \xrightarrow{>\epsilon}{}^*_{\mathcal{R}} t\sigma$ iff $\sigma \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t$.*

Proof. The claim vacuously follows if $\mathsf{root}(s) \neq \mathsf{root}(t)$. So let $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$. We have $f(s_1, \ldots, s_n)\sigma \xrightarrow{>\epsilon}{}^*_{\mathcal{R}} f(t_1, \ldots, t_n)\sigma$ iff $s_1\sigma \to^*_{\mathcal{R}} t_1\sigma, \ldots, s_n\sigma \to^*_{\mathcal{R}} t_n\sigma$ iff $\sigma \models_{\mathcal{R}} s_1 \twoheadrightarrow t_1 \wedge \ldots \wedge s_n \twoheadrightarrow t_n$. □

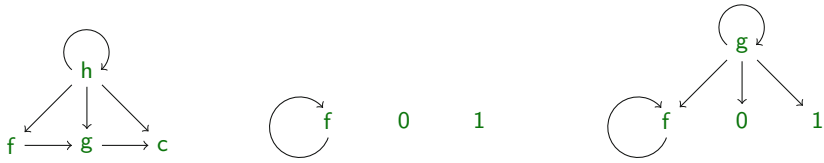
Combined with the observation that no root step is applicable to a term whose root symbol is a constructor, we obtain the following reformulation of a folklore result that reduces reachability to direct subterms.

Proposition 2. *If $s = f(\ldots)$ with $f \notin \mathcal{D}_{\mathcal{R}}$ and $t \notin \mathcal{V}$, then $s \twoheadrightarrow t \equiv_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t$.*

Proposition 2 is directly applicable in the analysis of dependency graphs.

Example 4. Consider again the constraint $\mathsf{f}^\sharp(x, x, x) \twoheadrightarrow \mathsf{f}^\sharp(0, 1, x')$ from Example 2. Since $\mathsf{f}^\sharp$ is not defined in $\mathcal{R}$, Proposition 2 reduces this constraint to $\mathsf{f}^\sharp(x, x, x) \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{f}^\sharp(0, 1, x')$, that is,

$$x \twoheadrightarrow 0 \wedge x \twoheadrightarrow 1 \wedge x \twoheadrightarrow x' \tag{3}$$



(a) Example 6 (b) Example 7 (c) Example 8

Fig. 1. Example symbol transition graphs.

4.1 Symbol Transition Graphs

Here we introduce a new, simple and efficient way of overapproximating reachability by tracking the relation of root symbols of terms according to a given set of rewrite rules. We first illustrate the intuition by an example.

Example 5. Consider a TRS $\mathcal{R}$ consisting of rules of the following form:

$$\mathsf{f}(...) \to \mathsf{g}(...) \qquad \mathsf{g}(...) \to \mathsf{c}(...) \qquad \mathsf{h}(...) \to x$$

Moreover, suppose $s \to_{\mathcal{R}}^* t$. Then we can make the following observations:

- If $\mathsf{root}(s) = \mathsf{c}$, then $\mathsf{root}(t) = \mathsf{c}$ since non-root steps preserve the root symbol and no root steps are applicable to terms of the form $\mathsf{c}(...)$.
- If $\mathsf{root}(s) = \mathsf{g}$, then $\mathsf{root}(t) \in \{\mathsf{g}, \mathsf{c}\}$ since non-root steps preserve the root symbol and the only possible root step is $\mathsf{g}(...) \to \mathsf{c}(...)$.
- If $\mathsf{root}(s) = \mathsf{f}$, then $\mathsf{root}(t) \in \{\mathsf{f}, \mathsf{g}, \mathsf{c}\}$ by the same reasoning.
- If $\mathsf{root}(s) = \mathsf{h}$, then t can be any term and $\mathsf{root}(t)$ can be arbitrary.

This informal argument is captured by the following definition.

Definition 6 (Symbol Transition Graphs). *The* symbol transition graph $\mathsf{SG}(\mathcal{R})$ *of a TRS* $\mathcal{R}$ *over signature* $\mathcal{F}$ *is the graph* $\langle \mathcal{F}, \rightarrowtail_{\mathcal{R}} \rangle$*, where* $f \rightarrowtail_{\mathcal{R}} g$ *iff* $\mathcal{R}$ *contains a rule of form* $f(...) \to g(...)$ *or* $f(...) \to x$ *with* $x \in \mathcal{V}$*.*

The following result tells us that for non-variable terms the symbol transition graph captures the relation between the root symbols of root rewrite steps.

Lemma 2. *If* $s \xrightarrow{\epsilon}_{\mathcal{R}} t$ *then* $t \in \mathcal{V}$ *or* $\mathsf{root}(s) \rightarrowtail_{\mathcal{R}} \mathsf{root}(t)$*.*

Proof. By assumption there exist $l \to r \in \mathcal{R}$ and σ such that $s = l\sigma$ and $r\sigma = t$. If $r \in \mathcal{V}$ then either $t \in \mathcal{V}$ or $\mathsf{root}(s) = \mathsf{root}(l) \rightarrowtail_{\mathcal{R}} \mathsf{root}(t)$. Otherwise, $\mathsf{root}(s) = \mathsf{root}(l) \rightarrowtail_{\mathcal{R}} \mathsf{root}(r) = \mathsf{root}(t)$. □

Since every rewrite sequence is composed of subsequences that take place entirely below the root (and hence do not change the root symbol) separated by root steps, we can extend the previous result to rewrite sequences.

Lemma 3. *If* $s = f(...) \to_{\mathcal{R}}^* g(...) = t$ *then* $f \rightarrowtail_{\mathcal{R}}^* g$*.*

Proof. We prove the claim for arbitrary s and f by induction on the derivation length of $s \to_{\mathcal{R}}^* t$. The base case is trivial, so consider $s \to_{\mathcal{R}} s' \to_{\mathcal{R}}^n t\sigma$. Since $t \notin \mathcal{V}$, we have $f' \in \mathcal{F}$ with $s' = f'(\ldots)$. Thus the induction hypothesis yields $f' \rightarrowtail_{\mathcal{R}}^* g$. If $s \xrightarrow{\epsilon}_{\mathcal{R}} s'$ then by Lemma 2 we conclude $f \rightarrowtail_{\mathcal{R}} f' \rightarrowtail_{\mathcal{R}}^* g$, and otherwise $f = f' \rightarrowtail_{\mathcal{R}}^* g$. □

It is now straightforward to derive the following from Lemma 3.

Corollary 1. *If $f \rightarrowtail_{\mathcal{R}}^* g$ does not hold, then $f(\ldots) \twoheadrightarrow g(\ldots) \equiv_{\mathcal{R}} \bot$.*

Example 6. The symbol transition graph for Example 5 is depicted in Fig. 1(a). By Corollary 1 we can conclude, for instance, $\mathsf{g}(\ldots) \twoheadrightarrow \mathsf{f}(\ldots)$ is unsatisfiable.

Corollary 1 is useful for checking (un)satisfiability of $s \twoheadrightarrow t$, only if neither s nor t is a variable. However, the symbol transition graph is also useful for unsatisfiability in the case when s and t may be variables.

Proposition 3. *If $\mathsf{SAT}_{\mathcal{R}}(x \twoheadrightarrow t_1 \wedge \ldots \wedge x \twoheadrightarrow t_n)$ for $t_1 = g_1(\ldots), \ldots, t_n = g_n(\ldots)$, then $g_1, \ldots, g_n$ are meetable with respect to $\rightarrowtail_{\mathcal{R}}$.*

Proof. By assumption there is a substitution σ such that $x\sigma \to_{\mathcal{R}}^* t_1\sigma, \ldots, x\sigma \to_{\mathcal{R}}^* t_n\sigma$. Clearly $x\sigma \in \mathcal{V}$ is not possible. Thus, suppose $x\sigma = f(\ldots)$ for some f. Finally, from Lemma 3, we have $f \rightarrowtail_{\mathcal{R}}^* g_1, \ldots, f \rightarrowtail_{\mathcal{R}}^* g_n$ and thereby conclude that $g_1, \ldots, g_n$ are meetable at f. □

The dual of Proposition 3 is proved in a similar way, but with some special care to ensure $x\sigma \in \mathcal{V}$.

Proposition 4. *If $\mathsf{SAT}_{\mathcal{R}}(s_1 \twoheadrightarrow x \wedge \ldots \wedge s_n \twoheadrightarrow x)$ for $s_1 = f_1(\ldots), \ldots, s_n = f_n(\ldots)$, then $f_1, \ldots, f_n$ are joinable with respect to $\rightarrowtail_{\mathcal{R}}$.*

Example 7 (Continuation of Example 4). Due to Proposition 3, proving (3) unsatisfiable reduces to proving that $\mathsf{0}$ and $\mathsf{1}$ are not meetable with respect to $\rightarrowtail_{\mathcal{R}}$. This is obvious from the symbol transition graph depicted in Fig. 1(b). Hence, we conclude the termination of $\mathcal{R}$.

Example 8. Consider the following extension of $\mathcal{R}$ from Example 2.

$$\mathsf{f}(\mathsf{0}, \mathsf{1}, x) \to \mathsf{f}(x, x, x) \qquad \mathsf{g}(x, y) \to x \qquad \mathsf{g}(x, y) \to y$$

The resulting system is not terminating [15]. The corresponding symbol transition graph is depicted in Fig. 1(c), where $\mathsf{0}$ and $\mathsf{1}$ are meetable, as expected.

4.2 Look-Ahead Reachability

Here we propose another method for overapproximating reachability, which eventually subsumes the tcap-unifiability method when target terms are linear. Note that this condition is satisfied in the dependency graph approximation of left-linear TRSs. Our method is based on the observation that any rewrite sequence either contains at least one root step, or takes place entirely below the root. This observation can be captured using our reachability constraints.

Definition 7 (Root Narrowing Constraints). *Let $l \to r$ be a rewrite rule with $\mathsf{Var}(l) = \{x_1, \ldots, x_n\}$. Then for terms s and t not containing $x_1, \ldots, x_n$, the* root narrowing constraint *from s to t via $l \to r$ is defined by*

$$s \rightsquigarrow_{l \to r} t := \exists x_1, \ldots, x_n.\, s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow l \wedge r \twoheadrightarrow t$$

We write $s \rightsquigarrow_{\mathcal{R}} t$ for $\bigvee_{l \to r \in \mathcal{R}'} s \rightsquigarrow_{l \to r} t$, where $\mathcal{R}'$ is a variant of $\mathcal{R}$ in which variables occurring in s or t are renamed to fresh ones.

In the definition above, the intuition is that if there are any root steps inside a rewrite sequence then we can pick the first one, which is only preceded by non-root steps. The following theorem justifies this intuition.

Theorem 2. *If $s, t \notin \mathcal{V}$, then $s \twoheadrightarrow t \equiv_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow t \vee s \rightsquigarrow_{\mathcal{R}} t$.*

Proof. Let $s = f(s_1, \ldots, s_n)$ and σ be a substitution. We show $\sigma \models_{\mathcal{R}} s \twoheadrightarrow t$ iff $\sigma \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow t \vee s \rightsquigarrow_{\mathcal{R}} t$. For the "if" direction suppose the latter. If $\sigma \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow t$, then t is of form $f(t_1, \ldots, t_n)$ and $s_i\sigma \to_{\mathcal{R}}^* t_i\sigma$ for every $i \in \{1, \ldots, n\}$, and thus $s\sigma \to_{\mathcal{R}}^* t\sigma$. If $\sigma \models_{\mathcal{R}} s \rightsquigarrow_{\mathcal{R}} t$, then we have a renamed variant $l \to r$ of a rule in $\mathcal{R}$ such that $\sigma \models_{\mathcal{R}} s \rightsquigarrow_{l \to r} t$. This indicates that there exists a substitution σ' that coincides with σ on $\mathcal{V} \setminus \mathsf{Var}(l)$, and satisfies

- $\sigma' \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow l$, that is, $l = f(l_1, \ldots, l_n)$ and $s_i\sigma' \to_{\mathcal{R}}^* l_i\sigma'$;
- $\sigma' \models_{\mathcal{R}} r \twoheadrightarrow t$, that is, $r\sigma' \to_{\mathcal{R}}^* t\sigma'$.

In combination, we have $s\sigma = s\sigma' \xrightarrow{>\epsilon}_{\mathcal{R}}^* l\sigma' \xrightarrow{\epsilon}_{\mathcal{R}} r\sigma' \to_{\mathcal{R}}^* t\sigma' = t\sigma$.

Now consider the "only if" direction. Suppose that σ is an idempotent substitution such that $s\sigma \to_{\mathcal{R}}^* t\sigma$. We may assume idempotence, since from any solution σ' of $s \twoheadrightarrow t$, we obtain idempotent solution σ by renaming variables in $\mathsf{Var}(s) \cup \mathsf{Var}(t)$ to fresh ones. We proceed by the following case analysis:

- *No root step is involved:* $s\sigma \xrightarrow{>\epsilon}_{\mathcal{R}}^* t\sigma$. Then Lemma 1 implies $\sigma \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow t$.
- *At least one root step is involved:* there is a rule $l \to r \in \mathcal{R}$ and a substitution θ such that $s\sigma \xrightarrow{>\epsilon}_{\mathcal{R}}^* l\theta$ and $r\theta \to_{\mathcal{R}}^* t\sigma$. Since variables in $l\theta$ must occur in $s\sigma$ (due to our assumptions on rewrite rules), we have $l\theta = l\theta\sigma$ since σ is idempotent. Thus from Lemma 1 we have $\sigma \models_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\!\rightarrow l\theta$. Further, variables in $r\theta$ must occur in $l\theta$ and thus in $s\theta$, we also have $r\theta\sigma = r\theta \to_{\mathcal{R}}^* t\sigma$, and hence $\sigma \models_{\mathcal{R}} r\theta \twoheadrightarrow t$. This concludes $\sigma \models_{\mathcal{R}} s \rightsquigarrow_{l \to r} t$. □

Proposition 2 is a corollary of Theorem 2 together with the following easy lemma, stating that if the root symbol of the source term is not a defined symbol, then no root step can occur.

Lemma 4. *If $c \notin \mathcal{D}_{\mathcal{R}}$ then $c(\ldots) \rightsquigarrow_{\mathcal{R}} t \equiv \bot$.*

Example 9. Consider the TRS $\mathcal{R}$ consisting of the following rules:

$$0 > x \to \mathsf{false} \qquad \mathsf{s}(x) > 0 \to \mathsf{true} \qquad \mathsf{s}(x) > \mathsf{s}(y) \to x > y$$

Applying Theorem 2 once reduces the reachability constraint $0 > z \twoheadrightarrow \mathsf{true}$ to the disjunction of

1. $0 > z \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{true}$,
2. $\exists x.\ 0 > z \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow 0 > x \ \wedge\ \mathsf{false} \twoheadrightarrow \mathsf{true}$
3. $\exists x.\ 0 > z \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{s}(x) > 0 \ \wedge\ \mathsf{true} \twoheadrightarrow \mathsf{true}$
4. $\exists x, y.\ 0 > z \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{s}(x) > \mathsf{s}(y) \ \wedge\ x > y \twoheadrightarrow \mathsf{true}$

Disjuncts 1, 3, and 4 expand to $\bot$ by definition of $\xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow$. For disjunct 2, applying Theorem 2 or Proposition 2 to $\mathsf{false} \twoheadrightarrow \mathsf{true}$ yields $\bot$.

Note that Theorem 2 can be applied arbitrarily often. Thus, to avoid nontermination in an implementation, we need to control how often it is applied. For this purpose we introduce the following definition.

Definition 8 (k-**Fold Look-Ahead**). *We define the* k-fold look-ahead transformation *with respect to a TRS* $\mathcal{R}$ *as follows:*

$$\mathsf{L}^k_{\mathcal{R}}(s \twoheadrightarrow t) := \begin{cases} \mathsf{L}^k_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t) \vee s \rightsquigarrow^k_{\mathcal{R}} t & \text{if } k \geq 1 \text{ and } s, t \notin \mathcal{V} \\ s \twoheadrightarrow t & \text{otherwise} \end{cases}$$

which is homomorphically extended to reachability constraints. Here, $\rightsquigarrow^k_{\mathcal{R}}$ *is defined as in Definition 7, but* k *controls the number of root steps to be expanded:*

$$s \rightsquigarrow^k_{l \to r} t := \exists x_1, \ldots, x_n.\ \mathsf{L}^k_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow l) \wedge \mathsf{L}^{k-1}_{\mathcal{R}}(r \twoheadrightarrow t)$$

It easily follows from Theorem 2 and induction on k that the k-fold look-ahead preserves the semantics of reachability constraints.

Corollary 2. $\mathsf{L}^k_{\mathcal{R}}(\phi) \equiv_{\mathcal{R}} \phi$.

The following results indicate that, whenever $\mathsf{tcap}_{\mathcal{R}}$-unifiability (Proposition 1) proves $s \twoheadrightarrow t$ unsatisfiable for linear t, $\mathsf{L}^1_{\mathcal{R}}$ can also conclude it.

Lemma 5. *Let* $s = f(s_1, \ldots, s_n)$ *and* $t \notin \mathcal{V}$ *be a linear term, and suppose that* $f(\mathsf{tcap}_{\mathcal{R}}(s_1), \ldots, \mathsf{tcap}_{\mathcal{R}}(s_n))$ *does not unify with* t *or any left-hand side in* $\mathcal{R}$. *Then* $\mathsf{L}^1_{\mathcal{R}}(s \twoheadrightarrow t) \equiv \bot$.

Proof. By structural induction on s. First, we show $\mathsf{L}^1_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t) \equiv \bot$. This is trivial if $\mathsf{root}(t) \neq f$. So let $t = f(t_1, \ldots, t_n)$. By assumption there is an $i \in \{1, \ldots, n\}$ such that $\mathsf{tcap}_{\mathcal{R}}(s_i)$ does not unify with t_i. Hence $\mathsf{tcap}_{\mathcal{R}}(s_i)$ cannot be a fresh variable, and thus s_i is of the form $g(u_1, \ldots, u_m)$ and $\mathsf{tcap}_{\mathcal{R}}(s_i) = g(\mathsf{tcap}_{\mathcal{R}}(u_1), \ldots, \mathsf{tcap}_{\mathcal{R}}(u_m))$ is not unifiable with any left-hand side in $\mathcal{R}$. Therefore, the induction hypothesis applies to s_i, yielding $\mathsf{L}^1_{\mathcal{R}}(s_i \twoheadrightarrow t_i) \equiv \bot$. This concludes $\mathsf{L}^1_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t) = \mathsf{L}^1_{\mathcal{R}}(s_1 \twoheadrightarrow t_1) \wedge \ldots \wedge \mathsf{L}^1_{\mathcal{R}}(s_n \twoheadrightarrow t_n) \equiv \bot$.

Second, we show $\mathsf{L}^1_{\mathcal{R}}(s \rightsquigarrow^1_{\mathcal{R}} t) \equiv \bot$. To this end, we show for an arbitrary variant $l \to r$ of a rule in $\mathcal{R}$ that $\mathsf{L}^1_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow l) \equiv \bot$. This is clear if $\mathsf{root}(l) \neq f$. So let $l = f(l_1, \ldots, l_n)$. By assumption there is an $i \in \{1, \ldots, n\}$ such that $\mathsf{tcap}_{\mathcal{R}}(s_i)$ and l_i are not unifiable. By a similar reasoning as above the induction hypothesis applies to s_i and yields $\mathsf{L}^1_{\mathcal{R}}(s_i \twoheadrightarrow l_i) \equiv \bot$. This concludes $\mathsf{L}^1_{\mathcal{R}}(s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow l) \equiv \bot$. □

Corollary 3. *If* $\mathsf{tcap}_{\mathcal{R}}(s)$ *and* t *are not unifiable, then* $\mathsf{L}^1_{\mathcal{R}}(s \twoheadrightarrow t) \equiv \bot$.

5 Conditional Rewriting

Conditional rewriting is a flavor of rewriting where rules are guarded by conditions. On the one hand, this gives us a boost in expressiveness in the sense that it is often possible to directly express equations with preconditions and that it is easier to directly express programming constructs like the where-clauses of Haskell. On the other hand, the analysis of conditional rewrite systems is typically more involved than for plain rewriting.

In this section we first recall the basics of conditional term rewriting. Then, we motivate the importance of reachability analysis for the conditional case. Finally, we extend the techniques of Sect. 4 to conditional rewrite systems.

Preliminaries. A *conditional rewrite rule* $l \to r \Leftarrow \phi$ consists of two terms $l \notin \mathcal{V}$ and r (the left-hand side and right-hand side, respectively) and a list ϕ of pairs of terms (its *conditions*). A *conditional term rewrite system* (CTRS for short) is a set of conditional rewrite rules. Depending on the interpretation of conditions, conditional rewriting can be separated into several classes. For the purposes of this paper we are interested in *oriented* CTRSs, where conditions are interpreted as reachability constraints with respect to conditional rewriting. Hence, from now on we identify conditions $\langle s_1, t_1\rangle, \ldots, \langle s_n, t_n\rangle$ with the reachability constraint $s_1 \twoheadrightarrow t_1 \wedge \ldots \wedge s_n \twoheadrightarrow t_n$, and the empty list with $\top$ (omitting "$\Leftarrow \top$" from rules).

The rewrite relation of a CTRS is layered into *levels*: given a CTRS $\mathcal{R}$ and level $i \in \mathbb{N}$, the corresponding (unconditional) TRS $\mathcal{R}_i$ is defined recursively:

$$\begin{aligned} \mathcal{R}_0 &:= \varnothing \\ \mathcal{R}_{i+1} &:= \{l\sigma \to r\sigma \mid l \to r \Leftarrow \phi \in \mathcal{R},\ \sigma \models_{\mathcal{R}_i} \phi\} \end{aligned}$$

Then the *(conditional) rewrite relation at level* i, written $\to_{\mathcal{R},i}$ (or $\to_i$ whenever $\mathcal{R}$ is clear from the context), is the plain rewrite relation $\to_{\mathcal{R}_i}$ induced by the TRS $\mathcal{R}_i$. Finally, the *induced (conditional) rewrite relation* of a CTRS $\mathcal{R}$ is defined by $\to_{\mathcal{R}} := \bigcup\{\to_i \mid i \geq 0\}$. At this point Definition 2 is extended to the conditional case in a straightforward manner.

Definition 9 (Level Satisfiability). *Let $\mathcal{R}$ be a CTRS and ϕ a reachability constraint. We say that a substitution σ* satisfies *ϕ modulo $\mathcal{R}$* at level *i, whenever $\sigma \models_{\mathcal{R},i} \phi$. If we are not interested in a specific satisfying substitution we say that ϕ* is satisfiable *modulo $\mathcal{R}$* at level *i and write $\mathsf{SAT}_{\mathcal{R},i}(\phi)$ (or just $\mathsf{SAT}_i(\phi)$ whenever $\mathcal{R}$ is clear from the context).*

5.1 Infeasibility

The main area of interest for reachability analysis in the conditional case is checking for *infeasibility*. While a formal definition of this concept follows below, for the moment, think of it as unsatisfiability of conditions. The two predominant applications of infeasibility are: (1) if the conditions of a rule are unsatisfiable,

the rule can never be applied and thus safely be removed without changing the induced rewrite relation; (2) if the conditions of a conditional critical pair (which arises from confluence analysis of CTRSs) are unsatisfiable, then it poses no problem to confluence and can safely be ignored.

Definition 10 (Infeasibility). *We say that a conditional rewrite rule* $l \to r \Leftarrow \phi$ *is* applicable at level i with respect to a CTRS $\mathcal{R}$ *iff* $\mathsf{SAT}_{\mathcal{R},i-1}(\phi)$. *A set* $\mathcal{S}$ *of rules is* infeasible with respect to $\mathcal{R}$ *when no rule in* $\mathcal{S}$ *is applicable at any level.*

The next theorem allows us to remove some rules from a CTRS while checking for infeasibility of rules.

Theorem 3. *A set* $\mathcal{S}$ *of rules is infeasible with respect to a CTRS* $\mathcal{R}$ *iff it is infeasible with respect to* $\mathcal{R} \setminus \mathcal{S}$.

Proof. The 'only if' direction is trivial. Thus we concentrate on the 'if' direction. To this end, assume that $\mathcal{S}$ is infeasible with respect to $\mathcal{R} \setminus \mathcal{S}$, but not infeasible with respect to $\mathcal{R}$. That is, at least one rule in $\mathcal{S}$ is applicable at some level with respect to $\mathcal{R}$. Let m be the minimum level such that there is a rule $l \to r \Leftarrow \phi \in \mathcal{S}$ that is applicable at level m with respect to $\mathcal{R}$. Now if $m = 0$ then $l \to r \Leftarrow \phi$ is applicable at level 0 and thus $\mathsf{SAT}_{\mathcal{R},0}(\phi)$, which trivially implies $\mathsf{SAT}_{\mathcal{R}\setminus\mathcal{S},0}(\phi)$, contradicting the assumption that all rules in $\mathcal{S}$ are infeasible with respect to $\mathcal{R} \setminus \mathcal{S}$. Otherwise, $m = k+1$ for some $k \geq 0$ and since $l \to r \Leftarrow \phi$ is applicable at level m we have $\mathsf{SAT}_{\mathcal{R},k}(\phi)$. Moreover, the rewrite relations $\to_{\mathcal{R},k}$ and $\to_{\mathcal{R}\setminus\mathcal{S},k}$ coincide (since all rules in $\mathcal{S}$ are infeasible at levels smaller than m by our choice of m). Thus we also have $\mathsf{SAT}_{\mathcal{R}\setminus\mathcal{S},k}(\phi)$, again contradicting the assumption that all rules in $\mathcal{S}$ are infeasible with respect to $\mathcal{R} \setminus \mathcal{S}$. □

The following example from *the confluence problems data base* (Cops)[3] shows that Theorem 3 is beneficial for showing infeasibility of conditional rewrite rules.

Example 10 (Cops 794). Consider the CTRS $\mathcal{R}$ consisting of the two rules:

$$\mathsf{a} \to \mathsf{c} \Leftarrow \mathsf{f}(\mathsf{a}) \twoheadrightarrow \mathsf{f}(\mathsf{b}) \qquad \mathsf{f}(\mathsf{b}) \to \mathsf{b}$$

The tcap-method does not manage to conclude infeasibility of the first rule, since $\mathsf{tcap}_{\mathcal{R}}(\mathsf{f}(\mathsf{a})) = x$ for some fresh variable x and thus unifies with $\mathsf{f}(\mathsf{b})$. The reason for this result was that for computing $\mathsf{tcap}_{\mathcal{R}}$ we had to recursively (in a bottom-up fashion) try to unify arguments of functions with left-hand sides of rules, which succeeded for the left-hand side of the first rule and the argument a of $\mathsf{f}(\mathsf{a})$, thereby obtaining $\mathsf{f}(x)$ which, in turn, unifies with the left-hand side of the second rule. But by Theorem 3 we do not need to consider the first rule for computing the term cap and thus obtain $\mathsf{tcap}_{\{\mathsf{f}(\mathsf{b})\to\mathsf{b}\}}(\mathsf{f}(\mathsf{a})) = \mathsf{f}(\mathsf{a})$ which does not unify with $\mathsf{f}(\mathsf{b})$ and thereby shows that the first rule is infeasible.

[3] http://cops.uibk.ac.at/?q=ctrs+oriented.

Fig. 2. Inductive and plain symbol transition graph of Example 11.

5.2 Symbol Transition Graphs in the Presence of Conditions

In the presence of conditions in rules we replace Definition 6 by the following inductive definition:

Definition 11 (Inductive Symbol Transition Graphs). *The* symbol transition graph $\mathsf{SG}(\mathcal{R})$ *of a CTRS* $\mathcal{R}$ *over a signature* $\mathcal{F}$ *is the graph* $\langle \mathcal{F}, \rightarrowtail_{\mathcal{R}} \rangle$ *where* $\rightarrowtail_{\mathcal{R}}$ *is defined inductively by the following two inference rules:*

$$\frac{f(\ldots) \to x \Leftarrow \phi \in \mathcal{R} \quad \forall \langle s,t \rangle \in \phi.\, s \in \mathcal{V} \vee t \in \mathcal{V} \vee \mathsf{root}(s) \rightarrowtail_{\mathcal{R}}^* \mathsf{root}(t)}{f \rightarrowtail_{\mathcal{R}} g}\ g \in \mathcal{F}$$

$$\frac{f(\ldots) \to g(\ldots) \Leftarrow \phi \in \mathcal{R} \quad \forall \langle s,t \rangle \in \phi.\, s \in \mathcal{V} \vee t \in \mathcal{V} \vee \mathsf{root}(s) \rightarrowtail_{\mathcal{R}}^* \mathsf{root}(t)}{f \rightarrowtail_{\mathcal{R}} g}$$

The example below shows the difference between the symbol transition graph for TRSs (which can be applied as a crude overapproximation also to CTRSs by dropping all conditions) and the inductive symbol transition graph for CTRSs.

Example 11 (Cops 293). Consider the CTRS consisting of the three rules:

$$\mathsf{a} \to \mathsf{b} \qquad \mathsf{a} \to \mathsf{c} \qquad \mathsf{b} \to \mathsf{c} \Leftarrow \mathsf{b} \twoheadrightarrow \mathsf{c}$$

The corresponding inductive symbol transition graph is depicted in Fig. 2(a) and implies unsatisfiability of $\mathsf{b} \twoheadrightarrow \mathsf{c}$. Note that this conclusion cannot be drawn from the plain symbol transition graph of the TRS obtained by dropping the condition of the third rule, shown in Fig. 2(b).

The inductive symbol transition graph gives us a sufficient criterion for concluding nonreachability with respect to a given CTRS, as shown in the following.

Lemma 6. *If* $f(\ldots) \to_{\mathcal{R}}^* g(\ldots)$ *then* $f \rightarrowtail_{\mathcal{R}}^* g$.

Proof. Let $s = f(\ldots)$ and $u = g(\ldots)$ and assume that s rewrites to u at level i, that is, $s \to_i^* u$. We prove the statement by induction on the level i. If $i = 0$ then we are done, since $\to_0$ is empty and therefore $f(\ldots) = s = u = g(\ldots)$, which trivially implies $f \rightarrowtail_{\mathcal{R}}^* g$. Otherwise, $i = j + 1$ and we obtain the induction hypothesis (IH) that $s \to_j^* t$ implies $\mathsf{root}(s) \rightarrowtail_{\mathcal{R}}^* \mathsf{root}(t)$ for arbitrary non-variable terms s and t. We proceed to show that $s \to_i^* u$ implies $f \rightarrowtail_{\mathcal{R}}^* g$ by an inner induction on the length of this derivation. If the derivation is empty, then $f(\ldots) = s =$

$u = g(...)$ and therefore trivially $f \rightarrowtail^*_{\mathcal{R}} g$. Otherwise, the derivation is of the shape $s \to_i^* t \to_i u$ for some non-variable term $t = h(...)$ and we obtain the inner induction hypothesis that $f \rightarrowtail^*_{\mathcal{R}} h$. It remains to show $h \rightarrowtail^*_{\mathcal{R}} g$ in order to conclude the proof. To this end, consider the step $t = C[l\sigma] \to_i C[r\sigma] = u$ for some context C, substitution σ, and rule $l \to r \Leftarrow \phi \in \mathcal{R}$ such that $\sigma \models_j \phi$. Now, by IH, we obtain that $s' \in \mathcal{V}$ or $t' \in \mathcal{V}$ or $\mathsf{root}(s') \rightarrowtail^*_{\mathcal{R}} \mathsf{root}(t')$ for all $\langle s', t' \rangle \in \phi$. Thus, by Definition 11, we obtain that $\mathsf{root}(l\sigma) \rightarrowtail_{\mathcal{R}} \mathsf{root}(r\sigma)$. We conclude by a case analysis on the structure of the context C. If C is empty, that is $C = \Box$, then $h = \mathsf{root}(l\sigma) \rightarrowtail^*_{\mathcal{R}} \mathsf{root}(r\sigma) = g$ and we are done. Otherwise, $h = \mathsf{root}(t) = \mathsf{root}(u) = g$ and therefore trivially $h \rightarrowtail^*_{\mathcal{R}} g$. □

Corollary 4. *If $f \rightarrowtail^*_{\mathcal{R}} g$ does not hold, then $f(...) \twoheadrightarrow g(...) \equiv_{\mathcal{R}} \bot$.*

5.3 Look-Ahead Reachability in the Presence of Conditions

In the following definition we extend our look-ahead technique from plain rewriting to conditional rewriting.

Definition 12 (Conditional Root Narrowing Constraints). *Let $l \to r \Leftarrow \phi$ be a conditional rewrite rule with $\mathsf{Var}(l) = \{x_1, \ldots, x_n\}$. Then for terms s and t not containing $x_1, \ldots, x_n$, the* conditional root narrowing constraint *from s to t via $l \to r \Leftarrow \phi$ is defined by*

$$s \rightsquigarrow_{l \to r \Leftarrow \phi} t := \exists x_1, \ldots, x_n.\, s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow l \wedge r \twoheadrightarrow t \wedge \phi$$

We write $s \rightsquigarrow_{\mathcal{R}} t$ for $\bigvee_{l \to r \Leftarrow \phi \in \mathcal{R}'} s \rightsquigarrow_{l \to r \Leftarrow \phi} t$, where $\mathcal{R}'$ is a variant of $\mathcal{R}$ in which variables occurring in s or t are renamed to fresh ones.

And we obtain a result similar to Theorem 2.

Lemma 7. *If $s, t \notin \mathcal{V}$, then $s \twoheadrightarrow t \equiv_{\mathcal{R}} s \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow t \vee s \rightsquigarrow_{\mathcal{R}} t$.*

Example 12 (Cops 793). Consider the CTRS $\mathcal{R}$ consisting of the two rules:

$$\mathsf{a} \to \mathsf{a} \Leftarrow \mathsf{f}(\mathsf{a}) \twoheadrightarrow \mathsf{a} \qquad\qquad \mathsf{f}(x) \to \mathsf{a} \Leftarrow x \twoheadrightarrow \mathsf{b}$$

To show infeasibility of the first rule we can safely remove it from $\mathcal{R}$ by Theorem 3, resulting in the modified CTRS $\mathcal{R}'$. Then we have to check $\mathsf{SAT}_{\mathcal{R}'}(\mathsf{f}(\mathsf{a}) \twoheadrightarrow \mathsf{a})$ which is made easier by the following chain of equivalences:

$$\begin{aligned}
\mathsf{f}(\mathsf{a}) \twoheadrightarrow \mathsf{a} &\equiv_{\mathcal{R}'} \mathsf{f}(\mathsf{a}) \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{a} \vee \mathsf{f}(\mathsf{a}) \rightsquigarrow_{\mathsf{f}(x) \to \mathsf{a} \Leftarrow x \twoheadrightarrow \mathsf{b}} \mathsf{a} && \text{(by Lemma 7)} \\
&\equiv_{\mathcal{R}'} \mathsf{f}(\mathsf{a}) \rightsquigarrow_{\mathsf{f}(x) \to \mathsf{a} \Leftarrow x \twoheadrightarrow \mathsf{b}} \mathsf{a} && \text{(by Definition 5)} \\
&\equiv_{\mathcal{R}'} \exists x.\, \mathsf{f}(\mathsf{a}) \xrightarrow{>\epsilon}\!\!\!\!\twoheadrightarrow \mathsf{f}(x) \wedge \mathsf{a} \twoheadrightarrow \mathsf{a} \wedge x \twoheadrightarrow \mathsf{b} && \text{(by Definition 12)} \\
&\equiv_{\mathcal{R}'} \exists x.\, \mathsf{a} \twoheadrightarrow x \wedge \mathsf{a} \twoheadrightarrow \mathsf{a} \wedge x \twoheadrightarrow \mathsf{b} && \text{(by Definition 5)}
\end{aligned}$$

Since satisfiability of the final constraint above implies $\mathsf{SAT}_{\mathcal{R}'}(\mathsf{a} \twoheadrightarrow \mathsf{b})$ and we also have $\mathsf{a} \not\rightarrowtail^*_{\mathcal{R}} \mathsf{b}$, we can conclude unsatisfiability of the original constraint by Corollary 4 and hence that the first rule of $\mathcal{R}$ is infeasible.

Table 1. Experimental results for dependency graph analysis (TRSs).

		Look-ahead				
		$\mathsf{L}^0_{\mathcal{R}}$	$\mathsf{L}^1_{\mathcal{R}}$	$\mathsf{L}^2_{\mathcal{R}}$	$\mathsf{L}^3_{\mathcal{R}}$	$\mathsf{L}^8_{\mathcal{R}}$
None	UNSAT	0	104 050	105 574	105 875	105 993
	time (s)	33.96	38.98	38.13	39.15	116.52
Corollary 1	UNSAT	307 207	328 216	328 430	328 499	328 636
	time (s)	38.50	42.71	42.72	43.00	66.82

6 Assessment

We implemented our techniques in the TRS termination prover NaTT [16][4] version 1.8 for dependency graph analysis, and the CTRS confluence prover ConCon [13][5] version 1.7 for infeasibility analysis. In both cases we only need a complete satisfiability checker, or equivalently, a sound unsatisfiability checker. Hence, to conclude unsatisfiability of given reachability constraints, we apply Corollary 2 with appropriate k together with a complete approximation of constraints. One such approximation is the symbol transition graph (Corollary 1). In the following we describe the experimental results on TRS termination and CTRS confluence. Further details of our experiments can be found at http://cl-informatik.uibk.ac.at/experiments/reachability/.

TRS Termination. For plain rewriting, we take all the 1498 TRSs from the TRS standard category of the *termination problem data base* version 10.6,[6] the benchmark used in the annual *Termination Competition* [8], and over-approximate their dependency graphs. This results in 1 133 963 reachability constraints, which we call "edges" here. Many of these edges are actually satisfiable, but we do not know the exact number (the problem is undecidable in general).

For checking unsatisfiability of edges, we combine Corollary 2 for various values of k (0, 1, 2, 3, and 8), and either Corollary 1 or 'None'. Here 'None' concludes unsatisfiability only for constraints that are logically equivalent to $\bot$. In Table 1 we give the number of edges that could be shown unsatisfiable. Here, the 'UNSAT' row indicates the number of detected unsatisfiable edges and the 'time' row indicates the total runtime in seconds. (We ran our experiments on an Amazon EC2 instance model `c5.xlarge`: 4 virtual 3.0 GHz Intel Xeon Platinum CPUs on 8 GB of memory).

The starting point is $\mathsf{L}^1_{\mathcal{R}}$ + None, which corresponds to the `tcap` technique, the method that was already implemented in NaTT before. The benefit of symbol transition graphs turns out to be quite significant, while the overhead in runtime seems acceptable. Moreover, increasing k of the look-ahead reasonably improves the power of unsatisfiability checks, both with and without the symbol transition

[4] https://www.trs.css.i.nagoya-u.ac.jp/NaTT/.
[5] http://cl-informatik.uibk.ac.at/software/concon/.
[6] http://www.termination-portal.org/wiki/TPDB.

graph technique. In terms of the overall termination proving power, NaTT using only tcap solves 1039 out of the 1498 termination problems, while using $\mathsf{L}^{8}_{\mathcal{R}}$ and Corollary 1, it proves termination of 18 additional problems.

CTRS Confluence. For conditional rewriting, we take the 148 oriented CTRSs of Cops,[7] a benchmark of confluence problems used in the annual *Confluence Competition* [1]. Compared to version 1.5 of ConCon (the winner of the CTRS category in the last competition in 2018) our new version (1.7) can solve five more systems (that is a gain of roughly 3%) by incorporating a combination of Theorem 3, inductive symbol transition graphs (Corollary 4), and k-fold look-ahead (Lemma 7), where for the latter we fixed $k = 1$ since we additionally have to control the level of conditional rewriting.

7 Related Work

Reachability is a classical topic in term rewriting; cf. Genet [7] for a survey. Some modern techniques include the tree-automata-completion approach [5,6] and a Knuth-Bendix completion-like approach [4]. Compared to these lines of work, first of all our interest is not directly in reachability problems but their (un)satisfiability. Middeldorp [12] proposed tree-automata techniques to approximate dependency graphs and made a theoretical comparison to an early term-cap-unifiability method [2], a predecessor of the tcap-based method. It is indeed possible (after some approximations of input TRSs) to encode our satisfiability problems into reachability problems between regular tree languages. However, our main motivation is to efficiently test reachability when analyzing other properties like termination and confluence. In that setting, constructing tree automata often leads to excessive overhead.

Our work is inspired by the work of Lucas and Gutiérrez [11]. Their *feasibility sequences* serve the same purpose as our reachability constraints, but are limited to atoms and conjunctions. Our formulation, allowing other constructions of logic formulas, is essential for introducing look-ahead reachability.

8 Conclusion

We introduced reachability constraints and their satisfiability problem. Such problems appear in termination and confluence analysis of plain and conditional rewriting. Moreover, we proposed two efficient techniques to prove (un)satisfiability of reachability constraints, first for plain and then for conditional rewriting. Finally, we implemented these techniques in the termination prover NaTT and the confluence prover ConCon, and experimentally verified their significance.

[7] http://cops.uibk.ac.at/?q=oriented+ctrs.

Acknowledgments. We thank Aart Middeldorp and the anonymous reviewers for their insightful comments. This work is supported by the Austrian Science Fund (FWF) project P27502 and ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST.

References

1. Aoto, T., Hirokawa, N., Nagele, J., Nishida, N., Zankl, H.: Confluence competition 2015. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 101–104. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_5
2. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theor. Compt. Sci. **236**(1–2), 133–178 (2000). https://doi.org/10.1016/S0304-3975(99)00207-8
3. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
4. Burel, G., Dowek, G., Jiang, Y.: A completion method to decide reachability in rewrite systems. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS (LNAI), vol. 9322, pp. 205–219. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_13
5. Felgenhauer, B., Thiemann, R.: Reachability, confluence, and termination analysis with state-compatible automata. Inf. Comput. **253**, 467–483 (2017). https://doi.org/10.1016/j.ic.2016.06.011
6. Feuillade, G., Genet, T., Viet Triem Tong, V.: Reachability analysis over term rewriting systems. J. Autom. Reason. **33**(341), 341–383 (2004). https://doi.org/10.1007/s10817-004-6246-0
7. Genet, T.: Reachability analysis of rewriting for software verification. Habilitation à diriger des recherches, Université de Rennes 1 (2009)
8. Giesl, J., Mesnard, F., Rubio, A., Thiemann, R., Waldmann, J.: Termination competition (termCOMP 2015). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 105–108. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_6
9. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_12
10. Hirokawa, N., Middeldorp, A.: Dependency pairs revisited. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 249–268. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-25979-4_18
11. Lucas, S., Gutiérrez, R.: Use of logical models for proving infeasibility in term rewriting. Inf. Process. Lett. **136**, 90–95 (2018). https://doi.org/10.1016/j.ipl.2018.04.002
12. Middeldorp, A.: Approximating dependency graphs using tree automata techniques. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 593–610. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_49
13. Sternagel, T., Middeldorp, A.: Conditional confluence (system description). In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 456–465. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_31

14. TeReSe: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
15. Toyama, Y.: Counterexamples to termination for the direct sum of term rewriting systems. Inf. Process. Lett. **25**(3), 141–143 (1987). https://doi.org/10.1016/0020-0190(87)90122-0
16. Yamada, A., Kusakari, K., Sakabe, T.: Nagoya termination tool. In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 466–475. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_32

Model Checking

VoxLogicA: A Spatial Model Checker for Declarative Image Analysis

Gina Belmonte[1], Vincenzo Ciancia[2(✉)], Diego Latella[2], and Mieke Massink[2]

[1] Azienda Ospedaliera Universitaria Senese, Siena, Italy
[2] Consiglio Nazionale delle Ricerche - Istituto di Scienza e Tecnologie dell'Informazione 'A. Faedo', CNR, Pisa, Italy
vincenzo.ciancia@isti.cnr.it

Abstract. Spatial and spatio-temporal model checking techniques have a wide range of application domains, among which large scale distributed systems and signal and image analysis. We explore a new domain, namely (semi-)automatic contouring in Medical Imaging, introducing the tool `VoxLogicA` which merges the state-of-the-art library of computational imaging algorithms `ITK` with the unique combination of declarative specification and optimised execution provided by spatial logic model checking. The result is a *rapid*, logic based analysis development methodology. The analysis of an existing benchmark of medical images for segmentation of brain tumours shows that simple `VoxLogicA` analysis can reach state-of-the-art accuracy, competing with best-in-class algorithms, with the advantage of *explainability* and easy *replicability*. Furthermore, due to a two-orders-of-magnitude speedup compared to the existing *general-purpose* spatio-temporal model checker `topochecker`, `VoxLogicA` enables *interactive* development of analysis of 3D medical images, which can greatly facilitate the work of professionals in this domain.

Keywords: Spatial logics · Closure spaces · Model checking · Medical Imaging

1 Introduction and Related Work

Spatial and Spatio-temporal model checking have gained an increasing interest in recent years in various domains of application ranging from Collective Adaptive Systems [11,15,18] and networked systems [27], to *signals* [32] and *digital images* [14,26]. Research in this field has its origin in the *topological* approach to spatial logics, dating back to the work of Alfred Tarski (see [9] for a thorough introduction). More recently these early theoretical foundations have been extended to encompass reasoning about *discrete* spatial structures, such as graphs and images, extending the theoretical framework of topology to *(quasi discrete) closure spaces* (see for instance [1,23,24]). That framework has subsequently been taken further in recent work by Ciancia et al. [13,14,17] resulting in the definition of the *Spatial Logic for Closure Spaces* (SLCS), temporal extensions (see [12,32,36]), and related model checking algorithms and tools.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 281–298, 2019.
https://doi.org/10.1007/978-3-030-17462-0_16

The main idea of spatial (and spatio-temporal) model checking is to use specifications written in a suitable logical language to describe spatial properties and to automatically identify patterns and structures of interest in a variety of domains (see e.g., [5,16,18]). In this paper we focus on one such domain, namely medical imaging for radiotherapy, and brain tumour segmentation in particular, which is an important and currently very active research domain of its own. One of the technical challenges of the development of automated (brain) tumour segmentation is that lesion areas are only defined through differences in the intensity (luminosity) in the (black & white) images that are *relative* to the intensity of the surrounding normal tissue. A further complication is that even (laborious and time consuming) manual segmentation by experts shows significant variations when intensity gradients between adjacent tissue structures are smooth or partially obscured [31]. Moreover, there is a considerable variation across images from different patients and images obtained with different Magnetic Resonance Images (MRI) scanners. Several automatic and semi-automatic methods have been proposed in this very active research area (see e.g., [20–22,29,34,37]).

This paper continues the research line of [3,7,8], introducing the free and open source tool `VoxLogicA` (*Voxel-based Logical Analyser*)[1], catering for a novel approach to image segmentation, namely a *rapid-development*, declarative, logic-based method, supported by *spatial model checking*. This approach is particularly suitable to reason at the "macro-level", by exploiting the *relative* spatial relations between tissues or organs at risk. `VoxLogicA` is similar, in the accepted logical language, and functionality, to the spatio-temporal model checker `topochecker`[2], but specifically designed for the analysis of (possibly multi-dimensional, e.g. 3D) *digital images* as a specialised image analysis tool. It is tailored to usability and efficiency by employing state-of-the-art algorithms and open source libraries, borrowed from computational image processing, in combination with efficient spatial model checking algorithms.

We show the application of `VoxLogicA` on BraTS 2017[3] [2,31,35], a publicly available set of benchmark MRI images for brain tumour segmentation, linked to a yearly challenge. For each image, a manual segmentation of the tumour by domain experts is available, enabling rigorous and objective qualitative comparisons via established similarity indexes. We propose a simple, yet effective, high-level specification for glioblastoma segmentation. The procedure, partly derived from the one presented in [3], directly competes in accuracy with the state-of-the-art techniques submitted to the BraTS 2017 challenge, most of which based on machine learning. Our approach to segmentation has the unique advantage of *explainability*, and is easy to replicate; in fact, the structure of a logically specified procedure can be explained to domain experts, and improved to encompass new observations. A mathematically formalised, unambiguous semantics permits results to be replicated not only by executing them in the multi-platform, open source tool that has been provided, but also by computing them via different implementations.

[1] `VoxLogicA`: https://github.com/vincenzoml/VoxLogicA.

[2] Topochecker: *a topological model checker*, see http://topochecker.isti.cnr.it, https://github.com/vincenzoml/topochecker.

[3] See https://www.med.upenn.edu/sbia/brats2017/data.html.

2 The Spatial Logic Framework

In this section, we briefly recall the logical language ImgQL (*Image Query Language*) proposed in [3], which is based on the *Spatial Logic for Closure Spaces* SLCS [13,14] and which forms the *kernel* of the framework we propose in the present paper. In Sect. 4 we will see how the resulting logic can be used for actual analysis via spatial model checking.

2.1 Foundations: Spatial Logics for Closure Spaces

The logic for closure spaces we use in the present paper is closely related to SLCS [13,14] and, in particular, to the SLCS extension with distance-based operators presented in [3]. As in [3], the resulting logic constitutes the *kernel* of a solid logical framework for reasoning about texture features of digital images, when interpreted as closure spaces. In the context of our work, a *digital image* is not only a 2-dimensional grid of *pixels*, but, more generally, a multi-dimensional (very often, 3-dimensional) grid of hyper-rectangular elements that are called *voxels* ("volumetric picture elements"). When voxels are not *hypercubes*, images are said to be *anisotropic*; this is usually the case in medical imaging. Furthermore, a digital image may contain information about its "real world" spatial dimensions, position (origin) and rotation, permitting one to compute the real-world coordinates of the centre and edges of each voxel. In medical imaging, such information is typically encapsulated into data by machines such as MRI scanners. In the remainder of the paper, we make no dimensionality assumptions. From now on, we refer to picture elements either as voxels or simply as points.

Definition 1. *A* closure space *is a pair* $(X, \mathcal{C})$ *where* X *is a non-empty set (of* points*) and* $\mathcal{C} : 2^X \to 2^X$ *is a function satisfying the following axioms:* $\mathcal{C}(\emptyset) = \emptyset$*;* $Y \subseteq \mathcal{C}(Y)$ *for all* $Y \subseteq X$*;* $\mathcal{C}(Y_1 \cup Y_2) = \mathcal{C}(Y_1) \cup \mathcal{C}(Y_2)$ *for all* $Y_1, Y_2 \subseteq X$*.* •

Given any relation $R \subseteq X \times X$, function $\mathcal{C}_R : 2^X \to 2^X$ with $\mathcal{C}_R(Y) \triangleq Y \cup \{x \mid \exists y \in Y. y\, R\, x\}$ satisfies the axioms of Definition 1 thus making $(X, \mathcal{C}_R)$ a closure space. Whenever a closure space is generated by a relation as above, it is called a *quasi-discrete* closure space. A quasi-discrete closure space $(X, \mathcal{C}_R)$, can be used as the basis for a mathematical model of a digital image. X represents the finite set of *voxels* and R is the reflexive and symmetric *adjacency* relation between voxels [25]. A closure space $(X, \mathcal{C})$ can be enriched with a notion of *distance*, i.e. a function $d : X \times X \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ such that $d(x, y) = 0$ iff $x = y$, leading to the *distance closure space* $((X, \mathcal{C}), d)$.[4]

[4] We recall that for $\emptyset \neq Y \subseteq X$, $d(x, Y) \triangleq \inf\{d(x, y) \mid y \in Y\}$, with $d(x, \emptyset) = \infty$. In addition, as the definition of d might require the elements of R to be weighted, quasi-discrete distance closure spaces may be enriched with a R-weighting function $\mathcal{W} : R \to \mathbb{R}$ assigning the weight $\mathcal{W}(x, y)$ to each $(x, y) \in R$. In the sequel we will keep $\mathcal{W}$ implicit, whenever possible and for the sake of simplicity.

It is sometimes convenient to equip the points of a closure space with *attributes*; for instance, in the case of images, such attributes could be the color or intensity of voxels. We assume sets A and V of attribute *names* and *values*, and an *attribute valuation* function $\mathcal{A}$ such that $\mathcal{A}(x, a) \in V$ is the value of attribute a of point x. Attributes can be used in *assertions* α, i.e. boolean expressions, with standard syntax and semantics. Consequently, we abstract from related details here and assume function $\mathcal{A}$ extended in the obvious way; for instance, $\mathcal{A}(x, a \leq c) = \mathcal{A}(x, a) \leq c$, for appropriate constant c.

A (quasi-discrete) *path* π in $(X, \mathcal{C}_R)$ is a function $\pi : \mathbb{N} \to X$, such that for all $Y \subseteq \mathbb{N}$, $\pi(\mathcal{C}_{Succ}(Y)) \subseteq \mathcal{C}_R(\pi(Y))$, where $(\mathbb{N}, \mathcal{C}_{Succ})$ is the closure space of natural numbers with the *successor* relation: $(n, m) \in Succ \Leftrightarrow m = n + 1$. Intuitively: the ordering in the path imposed by $\mathbb{N}$ is compatible with relation R, i.e. $\pi(i)\, R\, \pi(i + 1)$. For given set P of *atomic predicates* p, and interval of $\mathbb{R}$ I, the syntax of the logic we use in this paper is given below:

$$\Phi ::= p \mid \neg \Phi \mid \Phi_1 \wedge \Phi_2 \mid \mathcal{N}\Phi \mid \rho\, \Phi_1[\Phi_2] \mid \mathcal{D}^I \Phi \tag{1}$$

We assume that space is modelled by the set of points of a distance closure space; each atomic predicate $p \in P$ models a specific *feature* of *points* and is thus associated with the points that have this feature[5]. A point x satisfies $\mathcal{N}\Phi$ if a point satisfying Φ can be reached from x in at most one (closure) step, i.e. if x is *near* (or *close*) to a point satisfying Φ; x satisfies $\rho\, \Phi_1[\Phi_2]$ if x *may reach* a point satisfying Φ_1 via a path passing only by points satisfying Φ_2; it satisfies $\mathcal{D}^I\, \Phi$ if its distance from the set of points satisfying Φ falls in interval I. The logic includes logical negation ($\neg$) and conjunction ($\wedge$). In the following we formalise the semantics of the logic. A *distance closure model* $\mathcal{M}$ is a tuple $\mathcal{M} = (((X, \mathcal{C}), d), \mathcal{A}, \mathcal{V})$, where $((X, \mathcal{C}), d)$ is a distance closure space, $\mathcal{A} : X \times A \to V$ an attribute valuation, and $\mathcal{V} : P \to 2^X$ is a valuation of atomic propositions.

Definition 2. Satisfaction $\mathcal{M}, x \models \Phi$ *of a formula* Φ *at point* $x \in X$ *in model* $\mathcal{M} = (((X, \mathcal{C}), d), \mathcal{A}, \mathcal{V})$ *is defined by induction on the structure of formulas:*

$$\begin{array}{ll}
\mathcal{M}, x \models p \in P & \Leftrightarrow x \in \mathcal{V}(p) \\
\mathcal{M}, x \models \neg \Phi & \Leftrightarrow \mathcal{M}, x \models \Phi \text{ does not hold} \\
\mathcal{M}, x \models \Phi_1 \wedge \Phi_2 & \Leftrightarrow \mathcal{M}, x \models \Phi_1 \text{ and } \mathcal{M}, x \models \Phi_2 \\
\mathcal{M}, x \models \mathcal{N}\Phi & \Leftrightarrow x \in \mathcal{C}(\{y \mid \mathcal{M}, y \models \Phi\}) \\
\mathcal{M}, x \models \rho\, \Phi_1[\Phi_2] & \Leftrightarrow \text{there is path } \pi \text{ and index } \ell \text{ s.t. } \pi(0) = x \text{ and } \mathcal{M}, \pi(\ell) \models \Phi_1 \\
 & \quad \text{and for all indexes } j : 0 < j < \ell \text{ implies } \mathcal{M}, \pi(j) \models \Phi_2 \\
\mathcal{M}, x \models \mathcal{D}^I\, \Phi & \Leftrightarrow d(x, \{y \mid \mathcal{M}, y \models \Phi\}) \in I
\end{array}$$

where, when $p := \alpha$ *is a definition for* p*, we let* $x \in \mathcal{V}(p)$ *iff* $\mathcal{A}(x, \alpha)$ *is true.* •

[5] In particular, a predicate p can be a *defined* one, by means of a definition as $p := \alpha$, meaning that the feature of interest is characterized by the (boolean) value of α.

In the logic proposed in [13,14], the "may reach" operator is not present, and the *surrounded* operator $\mathcal{S}$ has been defined as basic operator as follows: x satisfies $\Phi_1 \mathcal{S} \Phi_2$ if and only if x belongs to an area of points satisfying Φ_1 and one cannot "escape" from such an area without hitting a point satisfying Φ_2. Several types of *reachability* predicates can be derived from $\mathcal{S}$. However, reachability is in turn a widespread, more basic primitive, implemented in various forms (e.g., *flooding*, *connected components*) in programming libraries. Thus, in this work, we prefer to use reachability as a basic predicate of the logic, as in [4], which is dedicated to extending the *Spatial Signal Temporal Logic* of [32]. In the sequel we show that $\mathcal{S}$ can be derived from the operators defined above, employing a definition patterned after the model-checking algorithm of [13]. This change simplifies the definition of several derived connectives, including that of *touch* (see below), and resulted in notably faster execution times for analyses using such derived connectives. We recall the definition of $\mathcal{S}$ from [14]: $\mathcal{M}, x \models \Phi_1 \mathcal{S} \Phi_2$ if and only if $\mathcal{M}, x \models \Phi_1$ and for all paths π and indexes ℓ we have: if $\pi(0) = x$ and $\mathcal{M}, \pi(\ell) \models \neg\Phi_1$, then there is j such that $0 < j \leq \ell$ and $\mathcal{M}, \pi(j) \models \Phi_2$.

Proposition 1. *For all closure models* $\mathcal{M} = ((X, \mathcal{C}), \mathcal{A}, \mathcal{V})$ *and all formulas* Φ_1, Φ_2 *the following holds:* $\Phi_1 \mathcal{S} \Phi_2 \equiv \Phi_1 \wedge \neg(\rho\ \neg(\Phi_1 \vee \Phi_2)[\neg\Phi_2])$ ◇

Definition 3. *We define some derived operators that are of particular use in medical image analysis:* $touch(\Phi_1, \Phi_2) \triangleq \Phi_1 \wedge \rho\ \Phi_2[\Phi_1]$; $grow(\Phi_1, \Phi_2) \triangleq \Phi_1 \vee touch(\Phi_2, \Phi_1)$; $flt(r, \Phi_1) \triangleq \mathcal{D}^{<r}(\mathcal{D}^{\geq r}\neg\Phi_1)$ •

The formula $touch(\Phi_1, \Phi_2)$ is satisfied by points that satisfy Φ_1 and that are on a path of points satisfying Φ_1 that reaches a point satisfying Φ_2. The formula $grow(\Phi_1, \Phi_2)$ is satisfied by points that satisfy Φ_1 and by points that satisfy Φ_2 which are on a path of points satisfying Φ_2 that reaches a point satisfying Φ_1. The formula $flt(r, \Phi_1)$ is satisfied by points that are at a distance of less than r from a point that is at least at distance r from points that do not satisfy Φ_1. This operator works as a filter; only contiguous areas satisfying Φ_1 that have a minimal diameter of at least $2r$ are preserved; these are also smoothened if they have an irregular shape (e.g. protrusions of less than the indicated distance).

Example 1. In Fig. 1, the top row shows four pictures using colours *blue* and *red*, interpreted as atomic propositions. Each picture in the bottom row shows in white the points that satisfy a given formula. In particular: Fig. 1e is $blue\,\mathcal{S}\,red$ of (a); Fig. 1f is touch$(red, blue)$ of (b); Fig. 1g is grow$(red, blue)$ of (c); Fig. 1h is $red\,\mathcal{S}\,(\mathcal{D}^{\leq 11} blue)$ of (d). For more details the reader is referred to [6].

2.2 Region Similarity via Statistical Cross-correlation

In the sequel, we provide some details on a logical operator, first defined in [3], that we use in the context of Texture Analysis (see for example [10,19,28,30]) for defining a notion of *statistical similarity* between image regions. The statistical distribution of an area Y of a black and white image is approximated

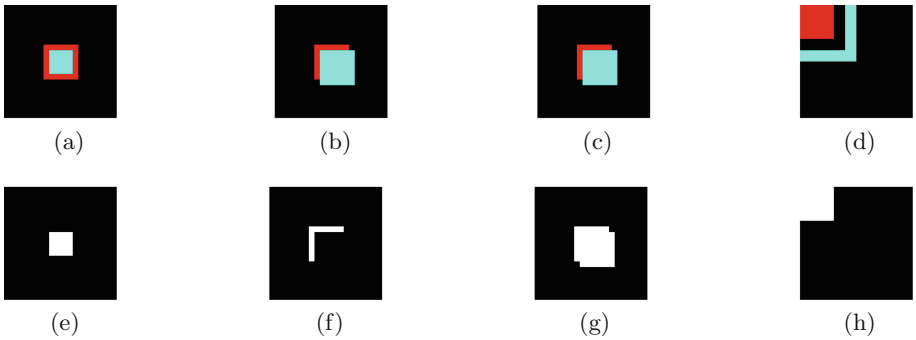


Fig. 1. Some examples of `ImgQL` operators (see Example 1). (Color figure online)

by the *histogram* of the grey levels of points (voxels) belonging to Y, limiting the representation to those levels laying in a certain interval $[m, M]$, the latter being split into k *bins*. In the case of images modelled as closure models, where each point may have several attributes, the histogram can be defined for different attributes. Given a closure model $\mathcal{M} = ((X, \mathcal{C}), \mathcal{A}, \mathcal{V})$, define function $\mathcal{H} : A \times 2^X \times \mathbb{R} \times \mathbb{R} \times \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$ such that for all $m < M$, $k > 0$ and $i \in \{1, \ldots, k\}$, $\mathcal{H}(a, Y, m, M, k)(i) = \left|\{y \in Y \mid (i-1) \cdot \Delta \leq \mathcal{A}(y, a) - m < i \cdot \Delta\}\right|$ where $\Delta = \frac{M-m}{k}$. We call $\mathcal{H}(a, Y, m, M, k)$ the *histogram* of Y (for attribute a), with k bins and m, M min and max values respectively. The *mean* $\overline{h}$ of a histogram h with k *bins* is the quantity $\frac{1}{k}\sum_{i=1}^{k} h(i)$. The *cross correlation* between two histograms h_1, h_2 with the same number k of *bins* is defined as follows: $\mathbf{r}(h_1, h_2) = \frac{\sum_{i=1}^{k}\left(h_1(i)-\overline{h_1}\right)\left(h_2(i)-\overline{h_2}\right)}{\sqrt{\sum_{i=1}^{k}\left(h_1(i)-\overline{h_1}\right)^2}\sqrt{\sum_{i=1}^{k}\left(h_2(i)-\overline{h_2}\right)^2}}$. The value of $\mathbf{r}$ is *normalised* so that $-1 \leq \mathbf{r} \leq 1$; $\mathbf{r}(h_1, h_2) = 1$ indicates that h_1 and h_2 are *perfectly correlated* (that is, $h_1 = ah_2 + b$, with $a > 0$); $\mathbf{r}(h_1, h_2) = -1$ indicates *perfect anti-correlation* (that is, $h_1 = ah_2 + b$, with $a < 0$). On the other hand, $\mathbf{r}(h_1, h_2) = 0$ indicates no correlation.

We embed *statistical similarity* $\triangle\!\!\!\triangle_{\bowtie c}\left[\begin{smallmatrix} m & M & k \\ r & a & b \end{smallmatrix}\right]$ in the logic by adding it to the grammar defined by (1) and extending the definition of the satisfaction relation (Definition 2) with the following equation, for m, M, k as above:

$$\mathcal{M}, x \models \triangle\!\!\!\triangle_{\bowtie c}\left[\begin{smallmatrix} m & M & k \\ r & a & b \end{smallmatrix}\right]\Phi \Leftrightarrow \mathbf{r}(h_a, h_b) \bowtie c$$

where $h_a = \mathcal{H}(a, S(x, r), m, M, k)$, $h_b = \mathcal{H}(b, \{y \mid \mathcal{M}, y \models \Phi\}, m, M, k)$, c is a constant in $[-1, 1]$, $\bowtie \in \{<, \leq, =, \geq, >\}$ and $S(x, r) = \{y \in X \mid d(x, y) \leq r\}$ is the *sphere* of radius r centred in x. Note that, differently from `topochecker` that was used in [3], in `VoxLogicA`, for efficiency reasons, $S(x, r)$ is actually the *hypercube* with edge size $2r$, which, for anisotropic images, becomes a hyperrectangle. So $\triangle\!\!\!\triangle_{\bowtie c}\left[\begin{smallmatrix} m & M & k \\ r & a & b \end{smallmatrix}\right]\Phi$ compares the region of the image constituted by the sphere (hypercube) of radius r centred in x against the region characterised by Φ. The comparison is based on the cross correlation of the histograms of the

chosen attributes of (the points of) the two regions, namely a and b and both histograms share the same range ($[m, M]$) and the same bins ($[1, k]$). In summary, the operator allows to check *to which extent* the *sphere (hypercube) around the point of interest* is *statistically similar* to a given region (specified by) Φ.

3 The Tool VoxLogicA

VoxLogicA is a framework for image analysis, that embeds the logic ImgQL into a user-oriented expression language to manipulate images. More precisely, the VoxLogicA type system distinguishes between *boolean-valued* images, that can be arguments or results of the application of ImgQL operators, and *number-valued* images, resulting from imaging primitives. Underlying such expression language is a *global model checker*, that is, the set of points satisfying a logic formula is computed at once; this is done implicitly when an expression corresponding to a logic formula is saved to an image. Functionality-wise, VoxLogicA specialises `topochecker` to the case of spatial analysis of *multi-dimensional images*. It interprets a specification written in the ImgQL language, using a set of multi-dimensional images[6] as models of the spatial logic, and produces as output a set of multi-dimensional images representing the valuation of user-specified expressions. For logical operators, such images are Boolean-valued, that is, *regions of interest* in medical imaging terminology, which may be loaded as *overlays* in medical image viewers. Non-logical operators may generate number-valued images. VoxLogicA augments ImgQL with file loading and saving primitives, and a set of additional commodity operators, specifically aimed at image analysis, that is destined to grow along with future developments of the tool. The main execution modality of VoxLogicA is *batch execution*. A (currently experimental) *graphical user interface* is under development.

Implementation-wise, the tool achieves a two-orders-of-magnitude speedup with respect to `topochecker`. Such speedup has permitted the rapid development of a novel procedure for automatic segmentation of *glioblastoma* that, besides being competitive with respect to the state-of-the-art in the field (see Sect. 4), is also easily *replicable* and *explainable* to humans, and therefore amenable of improvement by the community of medical imaging practitioners.

3.1 Functionality

We provide an overview of the tool functionality, starting from its syntax. For space reasons, we omit details on parsing rules (delegated to the tool documentation). In the following, `f, x1,..., xN, x` are identifiers, `"s"` is a string, and `e1, ..., eN, e` are expressions (to be detailed later). A VoxLogicA specification consists of a text file containing a sequence of **commands** (see Specification 1 in Sect. 4 as an example). Five commands are currently implemented:

[6] Besides common bitmap formats, the model loader of VoxLogicA currently supports the NIfTI (Neuro-imaging Informatics Technology Initiative) format (https://nifti.nimh.nih.gov/, version 1 and 2). 3D MR-FLAIR images in this format very often have a slice size of 256 by 256 pixels, multiplied by 20 to 30 slices.

- `let f(x1,...,xN) = e` is used for *function declaration*, also in the form `let f = e` *(constant declaration)*, and with special syntactic provisions to define *infix* operators. After execution of the command, name `f` is bound to a function or constant that evaluates to `e` with the appropriate substitutions of parameters;
- `load x = "s"` loads an image from file `"s"` and binds it to `x` for subsequent usage;
- `save "s" e` stores the image resulting from evaluation of expression `e` to file `"s"`;
- `print "s" e` prints to the log the string `s` followed by the numeric, or boolean, result of computing `e`;
- `import "s"` imports a library of declarations from file `"s"`; subsequent import declarations for the *same* file are not processed; furthermore, such imported files can only contain `let` or `import` commands.

`VoxLogicA` comes equipped with a set of built-in functions, such as arithmetic operators, logic primitives as described in Sect. 2, and imaging operators, for instance for computing the gray-scale intensity of a colour image, or its colour components, or the percentiles of its values (see Sect. 4.1). An exhaustive list of the available built-ins is provided in the user manual[7]. Furthermore, a "standard library" is provided containing short-hands for commonly used functions, and for derived operators. An **expression** may be a numeric literal (no distinction is made between floating point and integer constants), an identifier (e.g. `x`), a function application (e.g. `f(x1,x2)`), an infix operator application (e.g. `x1 + x2`), or a parenthesized (sub-)expression (e.g. `(x1 + x2)`).

The language features **strong dynamic typing**, that is, types of expressions are unambiguously checked and errors are precisely reported, but such checks are only performed at "run time", that is, when evaluating closed-form expressions with no free variables. The type system has currently been kept lightweight (the only typing rules regard constants and function application), in order to leave the design space open to future improvements. For instance, a planned development is function and operator *overloading*, as well as some form of static typing not interfering with the usability of the tool.

However, it is *not* the case that a type error may waste a long-running analysis. Type checking occurs after loading and parsing, but before analysis is run. Actual program execution after parsing is divided into two phases. First (usually, in a negligible amount of time), all the "save" and "print" instructions are examined to determine what expressions actually *need* to be computed; in this phase, name binding is resolved, all constant and function applications are substituted with closed expressions, types are checked and the environment binding expressions to names is discarded. Finally, the set of closed expressions to be evaluated is transformed into a set of tasks to be executed, possibly in parallel, and dependencies among them. After this phase, no further syntax processing or name resolution are needed, and it is guaranteed that the program is free from type errors. The second phase simply runs each task – in an order compliant with dependencies – parallelising execution on multiple CPU cores.

Each built-in logical operator has an associated type of its input parameters and output result. The available types are inductively defined as `Number`, `Bool`,

[7] See https://github.com/vincenzoml/VoxLogicA.

`String`, `Model`, and `Valuation(t)`, where `t` is in turn a type. The type `Model` is the type assigned to `x` in `load x = "f"`; operations such as the extraction of RGB components take this type as input, and return as output the only parametric type: `Valuation(t)`, which is the type of a multi-dimensional image in which each voxel contains a value of type `t`. For instance, the red component of a loaded model has type `Valuation(Number)`, whereas the result of evaluating a logic formula has type `Valuation(Bool)`[8].

An important aspect of the execution semantics of `VoxLogicA` specifications is *memoization*, constituting the core of its execution engine, and used to achieve maximal sharing of subformulas. In `VoxLogicA`, no expression is ever computed twice, freeing the user from worrying about how many times a given function is called, and making execution of complex macros and logical operators feasible.

3.2 Implementation Details

`VoxLogicA` is implemented in the functional, object-oriented programming language `FSharp`, using the `.NET Core` implementation of the `.NET` specification[9]. This permits a single code base with minimal environment-dependent setup to be cross-compiled and deployed as a standalone executable, for the major desktop operating systems, namely *Linux*, *macOS*, and *Windows*. Despite `.NET` code is compiled for an intermediate machine, this does not mean that efficiency of `VoxLogicA` is somehow "non-native". There are quite a number of measures in place to maximise efficiency. First and foremost, the execution time is heavily dominated by the time spent in native libraries (more details below), and `VoxLogicA` acts as a higher-level, declarative front-end for such libraries, adding a logical language, memoization, parallel execution, and abstraction from a plethora of technical details that a state-of-the-art imaging library necessarily exposes. In our experiments, parsing, memoization, and preparation of the tasks to be run may take a fraction of a second; the rest of the execution time (usually, several seconds, unless the analysis is extremely simple) is spent in *foreign function calls*. The major performance boosters in `VoxLogicA` are: a state-of-the-art computational imaging library (`ITK`); the optimised implementation of the *may reach* operator; a new algorithm for statistical cross-correlation; an efficient memoizing execution engine; parallel evaluation of independent tasks, exploiting modern multi-core CPUs. Moreover, special care has been put in making all performance-critical loops *allocationless*. All used memory along the loops is pre-allocated, avoiding the risk to trigger garbage collection during computation. We will address each of them briefly in the following.

ITK Library. `VoxLogicA` uses the state-of-the-art imaging library `ITK`, via the `SimpleITK` glue library[10]. Most of the operators of `VoxLogicA` are implemented

[8] Although such type system would permit "odd" types such as `Valuation(Model)`, there is no way to construct them; in the future this may change when appropriate.
[9] See https://fsharp.org and https://dotnet.github.io.
[10] See https://itk.org and http://www.simpleitk.org.

directly by a library call. Notably, this includes the *Maurer distance transform*, used to efficently implement the distance operators of ImgQL.

Novel Algorithms. The two most relevant operators that do not have a direct implementation in `ITK` are `mayReach` and `crossCorrelation`, implementing, respectively, the logical operator ρ, and statistical comparison described in Sect. 2.2. The computation of the voxels satisfying $\rho\ \phi_1[\phi_2]$ can be implemented either using the (classical, in computer graphics) *flood-fill* primitive, or by exploiting the *connected components* of ϕ_2 as a reachability primitive; both solutions are available in `SimpleITK`. In our experiments, connected components perform better using this library from `FSharp`, for large input seeds. Several critical logical connectives (e.g. *surrounded* and *touch*), are defined in terms of `mayReach`. Therefore, an optimised algorithm for `mayReach` is a key performance improvement. The `crossCorrelation` operation is resource-intensive, as it uses the histogram of a multi-dimensional hyperrectangle at each voxel. Pre-computation methods such as the *integral histogram* [33], would not yield the expected benefits, because cross-correlation is called only few times on the same image. In this work, we designed a parallel algorithm exploiting *additivity* of histograms. Given two sets of values P_1, P_2, let h_1, h_2 be their respective histograms, and let h'_1, h'_2 be the histograms of $P_1 \backslash P_2$ and $P_2 \backslash P_1$. For i a bin, we have $h_2(i) = h_1(i) - h'_1(i) + h'_2(i)$. This property leads to a particularly efficient algorithm when P_1 and P_2 are two hyperrectangles centred over adjacent voxels, as $P_1 \backslash P_2$ and $P_2 \backslash P_1$ are *hyperfaces*, having one dimension less than hyperrectangles. Our algorithm divides the image into as many partitions as the number of available processors, and then computes a *Hamiltonian path* for each partition, passing by each of its voxels exactly once. All partitions are visited in parallel, in the order imposed by such Hamiltonian paths; the histogram is computed incrementally as described above; finally cross-correlation is also computed and stored in the resulting image. The *asymptotic algorithmic complexity* of the implementation of ImgQL primitives in `VoxLogicA` is linear in the number of voxels, with the exception of `crossCorrelation`, which, by the above explanation, has complexity $O(k \cdot n)$, where n is the number of voxels, and k is the size of the largest hyperface of the considered hypercube.

Memoizing Execution Semantics. Sub-expressions in `VoxLogicA` are *by construction* identified up-to syntactic equality and assigned a number, representing a unique identifier (UID). UIDs start from 0 and are contiguous, therefore admitting an array of all existing sub-formulas to be used to pre-computed valuations of expressions without further hashing.

3.3 Design and Data Structures

The design of `VoxLogicA` defines three implementation layers. The *core* execution engine implements the concurrent, memoizing semantics of the tool. The *interpreter* is responsible for translating source code into core library invocations. These two layers only include some basic arithmetic and boolean primitives.

Operators can be added by inheriting from the abstract base class `Model`. The third implementation layer is the instantiation of the core layer to define operators from ImgQL, and loading and saving of graphical models, using the `ITK` library. We provide some more detail on the design of the core layer, which is the most critical part of `VoxLogicA`. At the time of writing, the core consists of just 350 lines of `FSharp` code, that has been carefully engineered not only for performance, but also for ease of maintenance and future extensions.

The essential **classes** are `ModelChecker`, `FormulaFactory`, `Formula`, and `Operator`, of which `Constant` is a subclass. Class `Operator` describes the available operators and their evaluation method. Class `Formula` is a symbolic representation of a syntactic sub-expression. Each instance of `Formula` has a unique numeric id (UID), an instance of `Operator`, and (inductively) a list of `Formula` instances, denoting its arguments. The UID of a formula is determined by the operator name (which is unique across the application), and the list of parameter UIDs. Therefore, by construction, it is not possible to build two different instances of `Formula` that are syntactically equal. UIDs are contiguous and start from 0. By this, all created formulas can be inserted into an array. Furthermore, UIDs are allocated in such a way that the natural number order is a topological sort of the dependency graph between subformulas (that is, if f_1 is a parameter of f_2, the UID of f_1 is greater than the UID of f_2). This is exploited in class `ModelChecker`; internally, the class uses an array to store the results of evaluating each `Formula` instance, implementing memoization. The class `ModelChecker` turns each formula into a task to be executed. Whenever a formula with UID i is a parameter of the formula with UID j, a dependency is noted between the associated tasks. The high-level, lightweight concurrent programming library `Hopac`[11] and its abstractions are used to evaluate the resulting task graph, in order to maximise CPU usage on multi-core machines.

4 Experimental Evaluation

The performance of `VoxLogicA` has been evaluated on the Brain Tumor Image Segmentation Benchmark (BraTS) of 2017 [2,31] containing 210 multi contrast MRI scans of high grade glioma patients that have been obtained from multiple institutions and were acquired with different clinical protocols and various scanners. All the imaging data sets provided by BraTS 2017 have been segmented manually and approved by experienced neuro-radiologists. In our evaluation we used the T2 Fluid Attenuated Inversion Recovery (FLAIR) type of scans, which is one of the four provided modalities in the benchmark. Use of other modalities is planned for future work. For training, the numeric parameters of the `VoxLogicA` specification presented in Sect. 4.1 were manually calibrated against a subset of 20 cases. Validation of the method was conducted as follows. A priori, 17 of the 210 cases can be excluded because the current procedure is not suitable for these images. This is because of the presence of multi-focal tumours (different tumours in different areas of the brain), or due to clearly distinguishable artifacts in the

[11] See https://github.com/Hopac/Hopac.

FLAIR acquisition, or because the hyperintense area is too large and clearly not significant (possibly by incorrect acquisition). Such cases require further investigation. For instance, the current procedure may be improved to identify specific types of artefacts, whereas multi-modal analysis can be used to complement the information provided by the FLAIR image in cases where FLAIR hyperintensity is not informative enough. In Sect. 4.2, we present the results both for the full dataset (210 cases), and for the subset without these problematic cases (193 cases). We considered both the *gross tumour volume* (GTV), corresponding to what can actually be seen on an image, and the *clinical target volume* (CTV) which is an extension of the GTV. For glioblastomas this margin is a 2–2.5 cm isotropic expansion of the GTV volume within the brain.

4.1 ImgQL Segmentation Procedure

Specification 1 shows the tumour segmentation procedure that we used for the evaluation[12]. The syntax is that of `VoxLogicA`, namely: `|,&,!` are boolean *or*, *and*, *not*; `distlt(c,phi)` is the set $\{y \mid \mathcal{M}, y \models \mathcal{D}^{<c}\texttt{phi}\}$ (similarly, `distgeq`; distances are in millimiters); `crossCorrelation(r,a,b,phi,m,M,k)` yields a cross-correlation coefficient for each voxel, to which a predicate c may be applied to obtain the *statistical similarity* function of Sect. 2.2; the `>` operator performs thresholding of an image; `border` is true on voxels that lay at the border of the image. Operator `percentiles(img,mask)`, where `img` is a number-valued image, and `mask` is boolean-valued, considers the points identified by `mask`, and assigns to each such point x the fraction of points that have an intensity below that of x in `img`. Other operators are explained in Definition 3 (see also Fig. 1). Figure 2 shows the intermediate phases of the procedure, for axial view of one specific 2D slice of an example 3D MRI scan of the BraTS 2017 data set.

We briefly discuss the specification (see [6] for more details). Lines 1–8 merely define utility functions and load the image, calling it `flair`. Lines 9–10 define the `background` as all voxels in the area of intensity less than 0.1 that touches the border of the image, and the `brain` as the complement of the background. The application of `percentiles` in line 11 assigns to each point of the brain the percentile rank of its intensity among those that are part of `brain`. Based on these percentiles, hyper-intense and very-intense points are identified that satisfy hI and vI, respectively (lines 12–13). Hyper-intense points have a very high likelihood to belong to tumour tissue; very-high intensity points are likely to belong to the tumour as well, or to the oedema that is usually surrounding the tumour. However, not all hyper-intense and very-intense points are part of a tumour. The idea is to identify the actual tumour using further spatial information. In lines 14–15 the hyper-intense and very-intense points are filtered, thus removing noise, and considering only areas of a certain relevant size.

[12] Note that, although the procedure is loosely inspired by the one in [3], there are major differences, partly due to a different method for identification of hyperintensities (using percentiles), and partly since the task in this work is simpler, as we only identify the CTV and GTV (avoiding, for instance, to label the oedema).

The points that satisfy `hyperIntense` and `veryIntense` are shown in red in Fig. 2a and in Fig. 2b, respectively. In line 16 the areas of hyper-intense points are extended via the `grow` operator, with those areas that are very intense (possibly belonging to the oedema), and in turn touch the hyper-intense areas. The points that satisfy `growTum` are shown in red in Fig. 2c. In line 17 the previously-defined (line 8) similarity operator is used to assign to all voxels a texture-similarity score with respect to `growTum`. In line 18 this operator is used to find those voxels that have a high cross correlation coefficient and thus are likely part of the tumour. The result is shown in Fig. 2d. Finally (line 19), the voxels that are identified as part of the whole tumour are those that satisfy `growTum` extended with those that are statistically similar to it via the `grow` operator. Points that satisfy `tumFinal` are shown in red in Fig. 2e and points identified by manual segmentation are shown for comparison in blue in the same figure (overlapping areas are purple).

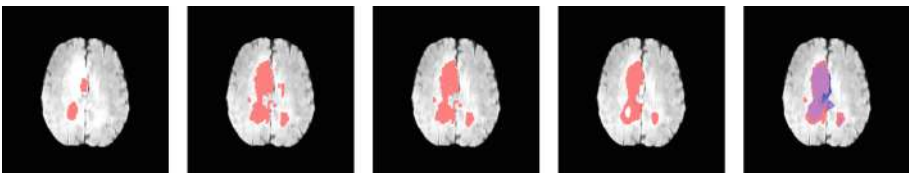
ImgQL **Specification 1:** Full specification of tumour segmentation

```
1  import "stdlib.imgql"
2  let grow(a,b) = (a | touch(b,a))
3  let flt(r,a) = distlt(r,distgeq(r,!a))
4  load imgFLAIR = "Brats17_2013_2_1_flair.nii.gz"
5  load imgManualSeg = "Brats17_2013_2_1_seg.nii.gz"
6  let manualContouring = intensity(imgManualSeg) > 0
7  let flair = intensity(imgFLAIR)
8  let similarFLAIRTo(a) =
   crossCorrelation(5,flair,flair,a,min(flair),max(flair),100)
9  let background = touch(flair < 0.1,border)
10 let brain = !background
11 let pflair = percentiles(flair,brain)
12 let hI = pflair > 0.95
13 let vI = pflair > 0.86
14 let hyperIntense = flt(5.0,hI)
15 let veryIntense = flt(2.0,vI)
16 let growTum = grow(hyperIntense,veryIntense)
17 let tumSim = similarFLAIRTo(growTum)
18 let tumStatCC = flt(2.0,(tumSim > 0.6))
19 let tumFinal= grow(growTum,tumStatCC)
20 save "output_Brats17_2013_2_1/complete-FLAIR_FL-seg.nii" tumFinal
```

Interesting aspects of the ImgQL specification are its relative simplicity and abstraction level, fitting that of neuro-radiologists, its explainability, its time-efficient verification, admitting a rapid development cycle, and its independence of normalisation procedures through the use of percentiles rather than absolute values for the intensity of voxels.

4.2 Validation Results

Results of tumour segmentation are evaluated based on a number of indexes commonly used to compare the quality of different techniques (see [31]). These indexes are based on the true positive (TP) voxels (voxels that are identified as part of a tumour in both manual and `VoxLogicA` segmentation), true negatives (TN) voxels (those that are *not* identified as part of a tumour in both manual and `VoxLogicA` segmentation), false positives (FP) voxels (those identified as part of a tumour by `VoxLogicA` but not by manual segmentation) and false negatives (FN) voxels (those identified as part of a tumour by manual segmentation but not by `VoxLogicA`). Based on these four types the following indexes are defined: *sensitivity*: $\mathrm{TP}/(\mathrm{TP}+\mathrm{FN})$; *specificity*: $\mathrm{TN}/(\mathrm{TN}+\mathrm{FP})$; *Dice*: $2*\mathrm{TP}/(2*\mathrm{TP}+\mathrm{FN}+\mathrm{FP})$. Sensitivity measures the fraction of voxels that are correctly identified as part of a tumour. Specificity measures the fraction of voxels that are correctly identified as *not* being part of a tumour. The Dice similarity coefficient is used to provide a measure of the similarity of two segmentations. Table 1 shows the mean values of the above indexes both for GTV and CTV volumes for Specification 1 applied to the BraTS 2017 training phase collection. The top-scoring methods of the BraTS 2017 Challenge [35] can be considered a good sample of the state-of-the-art in this domain. Among those, in order to collect significant statistics, we selected the 18 techniques that have been applied to at least 100 cases of the dataset. The *median* and *range of values* of the sensitivity, specificity and Dice indexes for the GTV segmentation of the whole tumour are, respectively, 0.88 (ranging from 0.55 to 0.97), 0.99 (0.98 to 0.999) and 0.88 (0.64 to 0.96). The 3D images used in this experiment have size $240 \times 240 \times 155$ (about 9 million voxels). The evaluation of each case study takes about 10 s on a desktop computer equipped with an Intel Core I7 7700 processor (with 8 cores) and 16 GB of RAM.



(a) (b) (c) (d) (e)

Fig. 2. Tumour segmentation of image `Brats17_2013_2_1`, FLAIR, axial 2D slice at X = 155, Y = 117 and Z = 97. (a) hyperIntense (b) veryIntense (c) growTum (d) tumStatCC (e) tumFinal (red) and manual (blue, overlapping area is purple). (Color figure online)

Table 1. VoxLogicA evaluation on the BraTS 2017 benchmark.

	Sensitivity (193 cases)	Specificity (193 cases)	Dice (193 cases)	Sensitivity (210 cases)	Specificity (210 cases)	Dice (210 cases)
GTV	0.89(0.10)	1.0(0.00)	0.85(0.10)	0.86(0.16)	1.0(0.0)	0.81(0.18)
CTV	0.95(0.07)	0.99(0.01)	0.90(0.09)	0.93(0.14)	0.99(0.2)	0.87(0.15)

4.3 Comparison with topochecker

The evaluation of VoxLogicA that we presented in this section uses features that are present in VoxLogicA, but not in topochecker. On the other hand, the example specification in [3], and its variant aimed at 3D images, are quite similar to the one we presented, and can be readily used to compare the performance of VoxLogicA and topochecker. The specifications consist of two human-authored text files of about 30 lines each. The specifications were run on a desktop computer equipped with an Intel Core I7 7700 processor (with 8 cores) and 16 GB of RAM. In the 2D case (image size: 512 × 512), topochecker took 52 s to complete the analysis, whereas VoxLogicA took 750 ms. In the 3D case (image size: 512 × 512 × 24), topochecker took about 30 min, whereas VoxLogicA took 15 s. As we mentioned before, this huge improvement is due to the combination of a specialised imaging library, new algorithms (e.g., for statistical similarity of regions), parallel execution and other optimisations. More details could be obtained by designing a specialised set of benchmarks, where some of which can also be run using topochecker; however, for the purposes of the current paper, the performance difference is so large that we do not deem such detailed comparison necessary.

5 Conclusions and Future Work

We presented VoxLogicA, a spatial model checker designed and optimised for the analysis of multi-dimensional digital images. The tool has been successfully evaluated on 193 cases of an international brain tumour 3D MRI segmentation benchmark. The obtained results are well-positioned w.r.t. the performance of state-of-the-art segmentation techniques, both efficiency-wise and accuracy-wise. Future research work based on the tool will focus on further benchmarking (e.g. various other types of tumours and tumour tissue such as necrotic and non-enhancing parts), and clinical application. On the development side, planned future work includes a graphical (web) interface for interactive parameter calibration (for that, execution times will need to be further improved, possibly employing *GPU computing*); improvements in the type-system (e.g. *operator overloading*); turning the core design layer into a reusable library available for other projects. Finally, the (currently small, albeit useful) library of logical and imaging-related primitives available will be enhanced, based on input from case studies. Calibration of the numerical parameters of our Glioblastoma segmentation was done manually. Future work aims at exploring different possibilities for

human-computer interaction in designing such procedures (e.g. via ad-hoc graphical interfaces), to improve user friendliness for domain experts. Experimentation in combining *machine-learning* methods with the logic-based approach of `VoxLogicA` are also worth being explored in this respect.

References

1. Aiello, M., Pratt-Hartmann, I., van Benthem, J.: Handbook of Spatial Logics. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5587-4
2. Bakas, S., et al.: Advancing the cancer genome atlas glioma MRI collections with expert segmentation labels and radiomic features. Sci. Data **4** (2017). https://doi.org/10.1038/sdata.2017.117. Accessed 05 Sept 2017
3. Banci Buonamici, F., Belmonte, G., Ciancia, V., et al.: Int. J. Softw. Tools Technol. Transfer (2019). https://doi.org/10.1007/s10009-019-00511-9
4. Bartocci, E., Bortolussi, L., Loreti, M., Nenzi, L.: Monitoring mobile and spatially distributed cyber-physical systems. In: Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE 2017, pp. 146–155. ACM, New York (2017). http://doi.acm.org/10.1145/3127041.3127050
5. Bartocci, E., Gol, E.A., Haghighi, I., Belta, C.: A formal methods approach to pattern recognition and synthesis in reaction diffusion networks. IEEE Trans. Control Netw. Syst. 1 (2016). https://doi.org/10.1109/tcns.2016.2609138
6. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: VoxLogicA: a spatial model checker for declarative image analysis (Extended Version). ArXiv e-prints, November 2018. https://arxiv.org/abs/1811.05677
7. Belmonte, G., et al.: A topological method for automatic segmentation of glioblastoma in MRI flair for radiotherapy. Magn. Reson. Mater. Phys. Biol. Med. **30**(S1), 437 (2017). https://doi.org/10.1007/s10334-017-0634-z. In ESMRMB 2017, 34th annual scientific meeting
8. Belmonte, G., Ciancia, V., Latella, D., Massink, M.: From collective adaptive systems to human centric computation and back: spatial model checking for medical imaging. In: ter Beek, M.H., Loreti, M. (eds.) Proceedings of the Workshop on FORmal Methods for the Quantitative Evaluation of Collective Adaptive SysTems, FORECAST@STAF 2016, Vienna, Austria, 8 July 2016. EPTCS, vol. 217, pp. 81–92 (2016). https://doi.org/10.4204/EPTCS.217.10
9. van Benthem, J., Bezhanishvili, G.: Modal logics of space. In: Aiello, M., Pratt-Hartmann, I., Van Benthem, J. (eds.) Handbook of Spatial Logics, pp. 217–298. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5587-4_5
10. Castellano, G., Bonilha, L., Li, L., Cendes, F.: Texture analysis of medical images. Clin. Radiol. **59**(12), 1061–1069 (2004)
11. Ciancia, V., Gilmore, S., Latella, D., Loreti, M., Massink, M.: Data verification for collective adaptive systems: spatial model-checking of vehicle location data. In: Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW, pp. 32–37. IEEE Computer Society (2014)
12. Ciancia, V., Grilletti, G., Latella, D., Loreti, M., Massink, M.: An experimental spatio-temporal model checker. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 297–311. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_24

13. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Specifying and verifying properties of space. In: Diaz, J., Lanese, I., Sangiorgi, D. (eds.) TCS 2014. LNCS, vol. 8705, pp. 222–235. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44602-7_18
14. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Model checking spatial logics for closure spaces. Log. Methods Comput. Sci. **12**(4), October 2016. http://lmcs.episciences.org/2067
15. Ciancia, V., Latella, D., Massink, M., Pakauskas, R.: Exploring spatio-temporal properties of bike-sharing systems. In: 2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops, pp. 74–79. IEEE Computer Society (2015)
16. Ciancia, V., Gilmore, S., Grilletti, G., Latella, D., Loreti, M., Massink, M.: Spatio-temporal model checking of vehicular movement in public transport systems. Int. J. Softw. Tools Technol. Transfer (2018). https://doi.org/10.1007/s10009-018-0483-8
17. Ciancia, V., Latella, D., Loreti, M., Massink, M.: Spatial logic and spatial model checking for closure spaces. In: Bernardo, M., De Nicola, R., Hillston, J. (eds.) SFM 2016. LNCS, vol. 9700, pp. 156–201. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34096-8_6
18. Ciancia, V., Latella, D., Massink, M., Paškauskas, R., Vandin, A.: A tool-chain for statistical spatio-temporal model checking of bike sharing systems. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016, Part I. LNCS, vol. 9952, pp. 657–673. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_46
19. Davnall, F., et al.: Assessment of tumor heterogeneity: an emerging imaging tool for clinical practice? Insights Imaging **3**(6), 573–589 (2012)
20. Despotović, I., Goossens, B., Philips, W.: MRI segmentation of the human brain: challenges, methods, and applications. Comput. Math. Methods Med. **2015**, 1–23 (2015). https://doi.org/10.1155/2015/450341
21. Dupont, C., Betrouni, N., Reyns, N., Vermandel, M.: On image segmentation methods applied to glioblastoma: state of art and new trends. IRBM **37**(3), 131–143 (2016). https://doi.org/10.1016/j.irbm.2015.12.004
22. Fyllingen, E.H., Stensjøen, A.L., Berntsen, E.M., Solheim, O., Reinertsen, I.: Glioblastoma segmentation: comparison of three different software packages. PLOS ONE **11**(10), e0164891 (2016). https://doi.org/10.1371/journal.pone.0164891
23. Galton, A.: The mereotopology of discrete space. In: Freksa, C., Mark, D.M. (eds.) COSIT 1999. LNCS, vol. 1661, pp. 251–266. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48384-5_17
24. Galton, A.: A generalized topological view of motion in discrete space. Theor. Comput. Sci. **305**(1–3), 111–134 (2003). https://doi.org/10.1016/S0304-3975(02)00701-6
25. Galton, A.: Discrete mereotopology. In: Calosi, C., Graziani, P. (eds.) Mereology and the Sciences: Parts and Wholes in the Contemporary Scientific Context, pp. 293–321. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05356-1_11
26. Grosu, R., Smolka, S., Corradini, F., Wasilewska, A., Entcheva, E., Bartocci, E.: Learning and detecting emergent behavior in networks of cardiac myocytes. Commun. ACM **52**(3), 97–105 (2009)
27. Haghighi, I., Jones, A., Kong, Z., Bartocci, E., Grosu, R., Belta, C.: Spatel: a novel spatial-temporal logic and its applications to networked systems. In: Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC 2015, pp. 189–198. ACM, New York (2015)

28. Kassner, A., Thornhill, R.E.: Texture analysis: a review of neurologic MR imaging applications. Am. J. Neuroradiol. **31**(5), 809–816 (2010)
29. Lemieux, L., Hagemann, G., Krakow, K., Woermann, F.: Fast, accurate, and reproducible automatic segmentation of the brain in t1-weighted volume mri data. Magn. Reson. Med. **42**(1), 127–135 (1999)
30. Lopes, R., et al.: Prostate cancer characterization on MR images using fractal features. Med. Phys. **38**(1), 83 (2011)
31. Menze, B.H., et al.: The multimodal brain tumor image segmentation benchmark (brats). IEEE Trans. Med. Imaging **34**(10), 1993–2024 (2015)
32. Nenzi, L., Bortolussi, L., Ciancia, V., Loreti, M., Massink, M.: Qualitative and quantitative monitoring of spatio-temporal properties. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 21–37. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_2
33. Porikli, F.M.: Integral histogram: a fast way to extract histograms in Cartesian spaces. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), vol. 1, pp. 829–836 (2005)
34. Simi, V., Joseph, J.: Segmentation of glioblastoma multiforme from MR images-a comprehensive review. Egypt. J. Radiol. Nucl. Med. **46**(4), 1105–1110 (2015). https://doi.org/10.1016/j.ejrnm.2015.08.001
35. Spyridon (Spyros) Bakas, et al. (Ed.): 2017 international MICCAI BraTS Challenge: Pre-conference Proceedings, September 2017. https://www.cbica.upenn.edu/sbia/Spyridon.Bakas/MICCAI_BraTS/MICCAI_BraTS_2017_proceedings_shortPapers.pdf
36. Tsigkanos, C., Kehrer, T., Ghezzi, C.: Modeling and verification of evolving cyber-physical spaces. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, pp. 38–48. ACM, New York (2017). http://doi.acm.org/10.1145/3106237.3106299
37. Zhu, Y., et al.: Semi-automatic segmentation software for quantitative clinical brain glioblastoma evaluation. Acad. Radiol. **19**(8), 977–985 (2012). https://doi.org/10.1016/j.acra.2012.03.026

On Reachability in Parameterized Phaser Programs

Zeinab Ganjei[1], Ahmed Rezine[1(✉)], Ludovic Henrio[2], Petru Eles[1], and Zebo Peng[1]

[1] Linköping University, Linköping, Sweden
{zeinab.ganjei,ahmed.rezine,petru.eles,zebo.peng}@liu.se
[2] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, 69342 Lyon Cedex 07, France
ludovic.henrio@ens-lyon.fr

Abstract. We address the problem of statically checking safety properties (such as assertions or deadlocks) for parameterized *phaser programs*. Phasers embody a non-trivial and modern synchronization construct used to orchestrate executions of parallel tasks. This generic construct supports dynamic parallelism with runtime registrations and deregistrations of spawned tasks. It generalizes many synchronization patterns such as collective and point-to-point schemes. For instance, phasers can enforce barriers or producer-consumer synchronization patterns among all or subsets of the running tasks. We consider in this work programs that may generate arbitrarily many tasks and phasers. We propose an exact procedure that is guaranteed to terminate even in the presence of unbounded phases and arbitrarily many spawned tasks. In addition, we prove undecidability results for several problems on which our procedure cannot be guaranteed to terminate.

1 Introduction

We focus on the parameterized verification problem of parallel programs that adopt the phasers construct for synchronization [15]. This coordination construct unifies collective and point-to-point synchronization. Parameterized verification is particularly relevant for mainstream parallel programs as the number of interdependent tasks in many applications, from scientific computing to web services or e-banking, may not be known apriori. Parameterized verification of phaser programs is a challenging problem due to the arbitrary numbers of involved tasks and phasers. In this work, we address this problem and provide an exact symbolic verification procedure. We identify parameterized problems for which our procedure is guaranteed to terminate and prove the undecidability of several variants on which our procedure cannot be guaranteed to terminate in general.

Phasers build on the clock construct from the X10 programming language [5] and are implemented in Habanero Java [4]. They can be added to any parallel programming language with a shared address space. Conceptually, phasers are synchronization entities to which tasks can be registered or unregistered.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 299–315, 2019.
https://doi.org/10.1007/978-3-030-17462-0_17

Registered tasks may act as producers, consumers, or both. Tasks can individually issue `signal`, `wait`, and `next` commands to a phaser they are registered to. Intuitively, a `signal` command is used to inform other tasks registered to the same phaser that the issuing task is done with its current phase. It increments the *signal* value associated to the issuing task on the given phaser. The `wait` command on the other hand checks whether all *signal* values in the phaser are strictly larger than the number of `wait`s issued by this task, i.e. all registered tasks have passed the issuing task's *wait* phase. It then increments the *wait* value associated to the task on the phaser. As a result, the `wait` command might block the issuing task until other tasks issue enough `signal`s. The `next` command consists in a `signal` followed by a `wait`. The `next` command may be associated to a sequence of statements that are to be executed in isolation by one of the registered tasks participating in the command. A program that does not use this feature of the next statement is said to be *non-atomic*. A task deregisters from a phaser by issuing a `drop` command on it.

The dynamic synchronization allowed by the construct suits applications that need dynamic load balancing (e.g., for solving non-uniform problems with unpredictable load estimates [17]). Dynamic behavior is enabled by the possible runtime creation of tasks and phasers and their registration/de-registration. Moreover, the spawned tasks can work in different phases, adding flexibility to the synchronization pattern. The generality of the construct makes it also interesting from a theoretical perspective, as many language constructs can be expressed using phasers. For example, synchronization barriers of Single Program Multiple Data programs, the Bulk Synchronous Parallel computation model [16], or promises and futures constructs [3] can be expressed using phasers.

We believe this paper provides general (un)decidability results that will guide verification of other synchronization constructs. We identify combinations of features (e.g., unbounded differences between signal and wait phases, atomic statements) and properties to be checked (e.g., assertions, deadlocks) for which the parameterized verification problem becomes undecidable. These help identify synchronization constructs with enough expressivity to result in undecidable parameterized verification problems. We also provide a symbolic verification procedure that terminates even on fragments with arbitrary phases and numbers of spawned tasks. We get back to possible implications in the conclusion:

- We show an operational model for phaser programs based on [4,6,9,15].
- We propose an exact symbolic verification procedure for checking reachability of sets of configurations for non-atomic phaser programs even when arbitrarily many tasks and phasers may be generated.
- We prove undecidability results for several reachability problems.
- We show termination of our procedure when checking assertions for non-atomic programs even when arbitrary many tasks may be spawned.
- We show termination of our procedure when checking deadlock-freedom and assertions for non-atomic programs with bounded gaps between *signal* and *wait* values, even when arbitrary many tasks may be spawned.

Related work. The closest work to ours is [9]. It is the only work on automatic and static formal verification of phaser programs. It does not consider the parameterized case. The current work studies decidability of different parameterized reachability problems and proposes a symbolic procedure that, for example, decides program assertions even in the presence of arbitrary many tasks. This is well beyond [9]. The work of [6] considers dynamic deadlock verification of phaser programs and can therefore only detect deadlocks at runtime. The work in [2] uses Java Path Finder [11] to explore all concrete execution paths. A more general description of the phasers mechanism has also been formalized in Coq [7].

Outline. We describe phasers in Sect. 2. The construct is formally introduced in Sect. 3 where we show a general reachability problem to be undecidable. We describe in Sect. 4 our symbolic representation and state some of its non-trivial properties. We use the representation in Sect. 5 to instantiate a verification procedure and establish decidability results. We refine our undecidability results in Sect. 6 and summarize our findings in Sect. 7. Proofs can be found in [10].

```
1  bool a, done;
2  main(){
3   done = false;
4   p= newPhaser(Sig_Wait);
5   c= newPhaser(Sig_Wait);
6   while(ndet()){
7    asynch(Prod,p:Sig,c:Wait);
8    asynch(Cons,p:Wait,c:Sig);
9   }
10  p.drop();
11  c.drop();
12 }
```

```
14 Prod(p:Sig,c:Wait)
15 {
16   while(¬done)
17   {
18     p.signal();
19     c.wait();
20     assert(a);
21     a = false;
22   };
23   p.drop();
24   c.drop();
25 }
```

```
27 Cons(p:Wait,c:Sig)
28 {
29  while(¬done){
30    p.wait();
31    if(ndet())
32     done = true;
33    a = true;
34    c.signal();
35  };
36  p.drop();
37  c.drop();
38 }
```

Fig. 1. An unbounded number of producers and consumers are synchronized using two phasers. In this construction, each consumer requires all producers to be ahead of it (wrt. the `p` phaser) in order for it to consume their respective products. At the same time, each consumer needs to be ahead of all producers (wrt. the `c` phaser) in order for the producers to be able to move to the next phase and produce new items.

2 Motivating Example

The program listed in Fig. 1 uses Boolean shared variables $\mathtt{B} = \{\mathtt{a}, \mathtt{done}\}$. The `main` task creates two phasers (line 4–5). When creating a phaser, the task gets automatically registered to it. The main task also creates an unbounded number of other task instances (lines 7–8). When a task t is registered to a phaser p, a pair (w_t^p, s_t^p) in $\mathbb{N}^2$ can be associated to the couple (t, p). The pair represents the individual *wait* and *signal* phases of task t on phaser p.

Registration of a task to a phaser can occur in one of three modes: Sig_Wait, Wait and Sig. In Sig_Wait mode, a task may issue both `signal` and `wait` commands. In Wait (resp. Sig) mode, a task may only issue `wait` (resp. `signal`) commands on the phaser. Issuing a `signal` command by a task on a phaser

results in the task incrementing its signal phase associated to the phaser. This is non-blocking. On the other-hand, issuing a `wait` command by a task on a phaser p will block until **all** tasks registered to p get signal values on p that are strictly larger than the wait value of the issuing task. The wait phase of the issuing task is then incremented. Intuitively, signals allow issuing tasks to state other tasks need not wait for them. In retrospect, waits allow tasks to make sure all registered tasks have moved past their wait phases.

Upon creation of a phaser, wait and signal phases are initialized to 0 (except in WAIT mode where no signal phase is associated to the task in order to not block other waiters). The only other way a task may get registered to a phaser is if an already registered task spawns and registers it in the same mode (or in WAIT or SIG if the registrar is registered in SIG_WAIT). In this case, wait and signal phases of the newly registered task are initialized to those of the registrar. Tasks are therefore dynamically registered (e.g., lines 7–8). They can also dynamically deregister themselves (e.g., line 10–11).

Here, an unbounded number of producers and consumers synchronize using two phasers. Consumers require producers to be ahead wrt. the phaser they point to with `p`. At the same time, consumers need to be ahead of all producers wrt. the phaser pointed to with `c`. It should be clear that phasers can be used as barriers for synchronizing dynamic subsets of concurrent tasks. Observe that tasks need not, in general, proceed in a lock step fashion. The difference between the largest signal value and the smallest wait value can be arbitrarily large (several signals before waits catch up). This allows for more flexibility.

We are interested in checking: (a) control reachability as in assertions (e.g., line 20), race conditions (e.g., mutual exclusion of lines 20 and 33) or registration errors (e.g., signaling a dropped phaser), and (b) plain reachability as in deadlocks (e.g., a producer at line 19 and a consumer at line 30 with equal phases waiting for each other). Both problems deal with reachability of sets of configurations. The difference is that control state reachability defines the targets with the states of the tasks (their control locations and whether they are registered to some phasers). Plain reachability can, in addition, constrain values of the phases in the target configurations (e.g., requiring equality between wait and signal values for deadlocks).

3 Phaser Programs and Reachability

We describe syntax and semantics of a core language. We make sure the language is representative of general purpose languages with phasers so that our results have a practical impact. A phaser program $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ involves a set `T` of tasks including a unique "main" task `main(){stmt}`. Arbitrary many instances of each task might be spawned during a program execution. All task instances share a set `B` of Boolean variables and make use of a set `V` of phaser variables that are local to individual task instances. Arbitrary many phasers might also be generated during program execution. Syntax of programs is as follows.

$$
\begin{aligned}
\texttt{prg} \;::=\; & \texttt{bool}\ \ \texttt{b}_1, \ldots, \texttt{b}_p; \\
& \texttt{task}_1(\texttt{v}_{1_1}, \ldots, \texttt{v}_{k_1})\,\{\texttt{stmt}_1\}; \\
& \ldots \\
& \texttt{task}_n(\texttt{v}_{1_n}, \ldots, \texttt{v}_{k_n})\,\{\texttt{stmt}_n\};
\end{aligned}
$$

$$
\begin{aligned}
\texttt{stmt} ::= \;& \texttt{v} = \texttt{newPhaser}(); \mid \texttt{asynch}(\texttt{task}, \texttt{v}_1, \ldots, \texttt{v}_k); \mid \texttt{v.drop}(); \mid \texttt{v.signal}(); \\
& \mid \texttt{v.wait}(); \mid \texttt{v.next}(); \mid \texttt{v.next}()\{\texttt{stmt}\}; \mid \texttt{b} := \texttt{cond}; \mid \texttt{assert}(\texttt{cond}); \\
& \mid \texttt{while}(\texttt{cond})\,\{\texttt{stmt}\}; \mid \texttt{stmt}\ \texttt{stmt} \mid \texttt{exit};
\end{aligned}
$$

$$
\texttt{cond} ::= \texttt{ndet}() \mid \texttt{true} \mid \texttt{false} \mid \texttt{b} \mid \texttt{cond} \vee \texttt{cond} \mid \texttt{cond} \wedge \texttt{cond} \mid \neg\texttt{cond}
$$

Initially, a unique task instance starts executing the $\texttt{main}()\{\texttt{stmt}\}$ task. A phaser can recall a pair of values (i.e., wait and signal) for each task instance registered to it. A task instance can create a new phaser with $\texttt{v} = \texttt{newPhaser}()$, get registered to it (i.e., gets zero as wait and signal values associated to the new phaser) and refer to the phaser with its local variable v. We simplify the presentation by assuming all registrations to be in SIG_WAIT mode. Including the other modes is a matter of depriving WAIT-registered tasks of a signal value (to ensure they do not block other registered tasks) and of ensuring issued commands respect registration modes. We use V for the union of all local phaser variables. A task $\texttt{task}(\texttt{v}_1, \ldots, \texttt{v}_k)\,\{\texttt{stmt}\}$ in T takes the phaser variables $\texttt{v}_1, \ldots \texttt{v}_k$ as parameters (write $\texttt{paramOf}(\texttt{task})$ to mean these parameters). A task instance can spawn another task instance with $\texttt{asynch}(\texttt{task}, \texttt{v}_1, \ldots, \texttt{v}_n)$. The issuing task instance registers the spawned task to the phasers pointed to by $\texttt{v}_1, \ldots, \texttt{v}_n$, with its own wait and signal values. Spawner and Spawnee execute concurrently. A task instance can deregister itself from a phaser with $\texttt{v.drop}()$.

A task instance can issue signal or wait commands on a phaser referenced by v and on which it is registered. A wait command on a phaser blocks until the wait value of the task instance executing the wait on the phaser is strictly smaller than the signal value of all task instances registered to the phaser. In other words, $\texttt{v.wait}()$ blocks if v points to a phaser such that at least one of the signal values stored by the phaser is equal to the wait value of the task that tries to perform the wait. A signal command does not block. It only increments the signal value of the task instance executing the signal command on the phaser. $\texttt{v.next}()$ is syntactic sugar for a signal followed by a wait. Moreover, $\texttt{v.next}()\{\texttt{stmt}\}$ is similar to $\texttt{v.next}()$ but the block of code stmt is executed atomically by exactly one of the tasks participating in the synchronization before all tasks continue the execution that follows the barrier. $\texttt{v.next}()\{\texttt{stmt}\}$ thus requires all tasks to be synchronized on exactly the same statement and is less flexible. Absence of a $\texttt{v.next}()\{\texttt{stmt}\}$ makes a program *non-atomic*.

Note that assignment of phaser variables is excluded from the syntax; additionally, we restrict task creation $\texttt{asynch}(\texttt{task}, \texttt{v}_1, \ldots, \texttt{v}_n)$ and require that parameter variables $\texttt{v}_i$ are all different. This prevents two variables from pointing to the same phaser and avoids the need to deal with aliasing: we can reason

on the single variable in a process that points to a phaser. Extending our work to deal with aliasing is easy but would require heavier notations.

We will need the notions of configurations, partial configurations and inclusion in order to define the reachability problems we consider in this work. We introduce them in the following and assume a phaser program $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$.

Configurations. Configurations of a phaser program describe valuations of its variables, control sequences of its tasks and registration details to the phasers.

Control sequences. We define the set $\texttt{Suff}$ of control sequences of $\texttt{prg}$ to be the set of suffixes of all sequences $\texttt{stmt}$ appearing in some statement $\texttt{task}(\ldots)\{\texttt{stmt}\}$. In addition, we define $\texttt{UnrSuff}$ to be the smallest set containing $\texttt{Suff}$ in addition to the suffixes of all (i) $\texttt{s}_1;\texttt{while}(\texttt{cond})\{\texttt{s}_1\};\texttt{s}_2$ if $\texttt{while}(\texttt{cond})\{\texttt{s}_1\};\texttt{s}_2$ is in $\texttt{UnrSuff}$, and of all (ii) $\texttt{s}_1;\texttt{s}_2$ if $\texttt{if}(\texttt{cond})\{\texttt{s}_1\};\texttt{s}_2$ is in $\texttt{UnrSuff}$, and of all (iii) $\texttt{s}_1;\texttt{v.next}()\{\};\texttt{s}_2$ if $\texttt{v.next}()\{\texttt{s}_1\};\texttt{s}_2$ in $\texttt{UnrSuff}$, and finally of all (iv) $\texttt{v.signal}();\texttt{v.wait}();\texttt{s}_2$ if $\texttt{v.next}()\{\};\texttt{s}_2$ is in $\texttt{UnrSuff}$. We write $\texttt{hd}(\texttt{s})$ and $\texttt{tl}(\texttt{s})$ to respectively mean the head and the tail of a sequence $\texttt{s}$.

Partial configurations. Partial configurations allow the characterization of sets of configurations by partially stating some of their common characteristics. A *partial configuration* c of $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ is a tuple $\big(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}\big)$ where:

- $\mathcal{T}$ is a finite set of task identifiers. We let t, u range over the values in $\mathcal{T}$.
- $\mathcal{P}$ is a finite set of phaser identifiers. We let p, q range over the values in $\mathcal{P}$.
- $\mathit{bv} : \texttt{B} \to \mathbb{B}^{\{*\}}$ fixes the values of some of the shared variables.[1]
- $\mathit{seq} : \mathcal{T} \to \texttt{UnrSuff}^{\{*\}}$ fixes the control sequences of some of the tasks.
- $\mathit{phase} : \mathcal{T} \to \texttt{partialFunctions}\left(\mathcal{P}, \texttt{V}^{\{-,*\}} \times \left(\mathbb{N}^2 \cup \{(*,*), \texttt{nreg}\}\right)\right)$ is a mapping that associates to each task t in $\mathcal{T}$ a partial mapping stating which phasers are known by the task and with which registration values.

Intuitively, partial configurations are used to state some facts about the valuations of variables and the control sequences of tasks and their registrations. Partial configurations leave some details unconstrained using partial mappings or the symbol $*$. For instance, if $\mathit{bv}(\texttt{b}) = *$ in a partial configuration $\big(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}\big)$, then the partial configuration does not constrain the value of the shared variable $\texttt{b}$. Moreover, a partial configuration does not constrain the relation between a task t and a phaser p when $\mathit{phase}(t)(p)$ is undefined. Instead, when the partial mapping $\mathit{phase}(t)$ is defined on phaser p, it associates a pair $\mathit{phase}(t)(p) = (\texttt{var}, \texttt{val})$ to p. If $\texttt{var} \in \texttt{V}^{\{-,*\}}$ is a variable $\texttt{v} \in \texttt{V}$ then the task t in $\mathcal{T}$ uses its variable $\texttt{v}$ to refer to the phaser p in $\mathcal{P}$[2]. If $\texttt{var}$ is the symbol $-$ then the task t does not refer to $\texttt{v}$ with any of its variables in $\texttt{V}$. If $\texttt{var}$ is the symbol $*$, then the task might or might not refer to p. The value $\texttt{val}$ in $\mathit{phase}(t)(p) = (\texttt{var}, \texttt{val})$ is either the value $\texttt{nreg}$ or a pair (w, s). The value $\texttt{nreg}$ means the task t is not registered to phaser p. The pair (w, s) belongs to $(\mathbb{N} \times \mathbb{N}) \cup \{(*,*)\}$. In this case, task t is registered to phaser p with a wait phase

[1] For any set S, $S^{\{a,b,\ldots\}}$ denotes $S \cup \{a, b, \ldots\}$.

[2] The uniqueness of this variable is due to the absence of aliasing discussed above.

$$\frac{\texttt{hd}(\mathit{seq}(t)) = \texttt{assert(cond)} \quad [\![\texttt{cond}]\!]_{\mathit{bv}} = \texttt{false}}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \in \texttt{AssertErrors}} \ (\texttt{assertion})$$

$$\frac{\begin{array}{c}\texttt{hd}(\mathit{seq}(t)) = \texttt{b}' := \texttt{cond}' \quad (b' \text{ coincides with } b \text{ or appears in } \texttt{cond}) \\ \texttt{hd}(\mathit{seq}(u)) \in \{\texttt{b} := \texttt{cond}, \texttt{if(cond)}\,\{\texttt{stmt}\}, \texttt{while(cond)}\,\{\texttt{stmt}\}, \texttt{assert(cond)}\}\end{array}}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \in \texttt{RaceErrors}} \ (\texttt{race})$$

$$\frac{\begin{array}{c}t \in \mathcal{T} \quad \texttt{v} \in \mathbb{V} \quad p \in \mathcal{P} \quad \mathit{phase}(t)(p) = (\texttt{v}, \texttt{nreg}) \\ \texttt{hd}(\mathit{seq}(t)) \in \{\texttt{asynch(task,}\ldots\texttt{v}\ldots\texttt{)}\{\texttt{stmt}\}, \texttt{v.signal()}, \texttt{v.wait()}, \texttt{v.drop()}\}\end{array}}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \in \texttt{RegisterErrors}} \ (\texttt{registration})$$

$$\frac{\begin{array}{c}t_0, \ldots t_n \in \mathcal{T} \quad \texttt{v}_0, \ldots \texttt{v}_n \in \mathbb{V} \quad var_0, \ldots var_n \in \mathbb{V}^{\{-\}} \quad p_0, \ldots p_n \in \mathcal{P} \\ s_0 \ldots s_n, s'_0 \ldots s'_n \in \mathbb{N} \quad w_0 \ldots w_n, w'_0 \ldots w'_n \in \mathbb{N} \quad \forall i : 0 \leq i \leq n. \quad \texttt{hd}(\mathit{seq}(t_i)) = \texttt{v}_i\texttt{.wait()} \\ \mathit{phase}(t_i)(p_i) = (\texttt{v}_i, (w_i, s_i)) \quad \mathit{phase}(t_i)(p_{(i+1)\%(n+1)}) = (var_i, (w'_i, s'_i)) \quad s'_i = w_{(i+1)\%(n+1)}\end{array}}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \in \texttt{DeadlockErrors}} \ (\texttt{deadlock})$$

Fig. 2. Definition of error configurations. For instance, a deadlock is obtained in $(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ if tasks $\{t_0, \ldots, t_n\} \subseteq \mathcal{T}$ form a cycle where each t_i blocks the wait being executed by $t_{(i+1)\%(n+1)}$ on phaser $p_{(i+1)\%(n+1)}$.

w and a signal phase s. The value $*$ means that the wait phase w (resp. signal phase s) can be any value in $\mathbb{N}$. For instance, $\mathit{phase}(t)(p) = (\texttt{v}, \texttt{nreg})$ means variable $\texttt{v}$ of the task t refers to phaser p but the task is not registered to p. On the other hand, $\mathit{phase}(t)(p) = (-, (*, *))$ means the task t does not refer to p but is registered to it with arbitrary wait and signal phases.

Concrete configurations. A concrete configuration (or configuration for short) is a partial configuration $(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ where $\mathit{phase}(t)$ is total for each $t \in \mathcal{T}$ and where the symbol $*$ does not appear in any range. It is a tuple $(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ where $\mathit{bv} : \texttt{B} \to \mathbb{B}$, $\mathit{seq} : \mathcal{T} \to \texttt{UnrSuff}$, and $\mathit{phase} : \mathcal{T} \to \texttt{totalFunctions}\,(\mathcal{P}, \mathbb{V}^{\{-\}} \times ((\mathbb{N} \times \mathbb{N}) \cup \{\texttt{nreg}\}))$. For a concrete configuration $(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$, we write $\texttt{isReg}(\mathit{phase}, t, p)$ to mean the predicate $\mathit{phase}(t)(p) \notin (\mathbb{V}^{\{-\}} \times \{\texttt{nreg}\})$. The predicate $\texttt{isReg}(\mathit{phase}, t, p)$ captures whether the task t is registered to phaser p according to the mapping phase.

Inclusion of configurations. A configuration $c' = (\mathcal{T}', \mathcal{P}', \mathit{bv}', \mathit{seq}', \mathit{phase}')$ *includes* a partial configuration $c = (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ if renaming and deleting tasks and phasers from c' can give a configuration that "matches" c. More formally, c' includes c if $((\mathit{bv}(\texttt{b}) \neq \mathit{bv}'(\texttt{b})) \implies (\mathit{bv}(\texttt{b}) = *))$ for each $\texttt{b} \in \texttt{B}$ and there are injections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ s.t. for each $t \in \mathcal{T}$ and $p \in \mathcal{P}$: (1) $((\mathit{seq}(t) \neq \mathit{seq}'(\tau(t))) \implies (\mathit{seq}(t) = *))$, and either (2.a) $\mathit{phase}(t)(p)$ is undefined, or (2.b) $\mathit{phase}(t)(p) = (var, val)$ and $\mathit{phase}'(\tau(t))(\pi(p)) = (var', val')$ with $((var \neq var') \implies (var = *))$ and either $(val = val' = \texttt{nreg})$ or $val = (w, s)$ and $val' = (w', s')$ with $((w \neq w') \implies (w = *))$ and $((s \neq s') \implies (s = *))$.

Semantics and reachability. Given a program $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$, the main task $\texttt{main()}\{\texttt{stmt}\}$ starts executing $\texttt{stmt}$ from an initial configuration $c_{init} = (\mathcal{T}_{init}, \mathcal{P}_{init}, \mathit{bv}_{init}, \mathit{seq}_{init}, \mathit{phase}_{init})$ where $\mathcal{T}_{init}$ is a singleton, $\mathcal{P}_{init}$ is empty, bv_{init} sends all shared variables to $\texttt{false}$ and seq_{init} associates $\texttt{stmt}$ to the unique task in $\mathcal{T}_{init}$. We write $c \xrightarrow[\texttt{stmt}]{t} c'$ to mean a task t in c can fire statement $\texttt{stmt}$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{v} := \mathtt{newPhaser}() \quad p \notin \mathcal{P} \quad \mathcal{P}' = \mathcal{P} \cup \{p\} \quad \mathit{phase}' = \mathit{phase}[t \leftarrow \{p \leftarrow (\mathtt{v}, (0,0))\}] \quad \mathit{phase}'' = \mathit{phase}'\big[\{u \leftarrow \mathit{phase}'(u)[p \leftarrow (-, \mathtt{nreg})] \mid u \in \mathcal{T} \setminus \{t\}\}\big]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v}:=\mathtt{newPhaser}()]{t} (\mathcal{T}, \mathcal{P}', \mathit{bv}, \mathit{seq}\big[t \leftarrow \mathtt{tl}(\mathit{seq}(t))\big], \mathit{phase}'')} \ (\mathtt{newPhaser})$$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{asynch}(\mathtt{task}, \mathtt{v}_1, \ldots \mathtt{v}_k)\{\mathtt{stmt}\} \quad \mathtt{paramOf}(\mathtt{task}) = (\mathtt{w}_1, \ldots \mathtt{w}_k) \quad u \notin \mathcal{T} \quad \mathcal{T}' = \mathcal{T} \cup \{u\} \quad \text{for each } i : 1 \le i \le \mathtt{k}.\ \mathit{phase}(t)(p_i) = (\mathtt{v}_i, (w_i, s_i)) \quad \text{for each } i, j : 1 \le i, j \le \mathtt{k}.\ i \ne j \Rightarrow \mathtt{v}_i \ne \mathtt{v}_j \quad \mathit{phase}' = \mathit{phase}[u \leftarrow \{p_i \leftarrow (\mathtt{w}_i, (w_i, s_i)) \mid 1 \le i \le \mathtt{k}\} \cup \{p \leftarrow (-, \mathtt{nreg}) \mid p \notin \{p_i \mid 1 \le i \le \mathtt{k}\}\}]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{asynch}(\mathtt{task}, \mathtt{v}_1, \ldots \mathtt{v}_k)\{\mathtt{stmt}\}]{t} (\mathcal{T}', \mathcal{P}, \mathit{bv}, \mathit{seq}'\big[\{u \leftarrow \mathtt{stmt}\} \cup \{t \leftarrow \mathtt{tl}(\mathit{seq}(t))\}\big], \mathit{phase}')} \ (\mathtt{asynch})$$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{v.signal}() \quad \mathit{phase}(t)(p) = (\mathtt{v}, (w, s)) \quad \mathit{phase}' = \mathit{phase}[t \leftarrow \{p \leftarrow (\mathtt{v}, (w, 1+s))\}]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v.signal}()]{t} (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}\big[t \leftarrow \mathtt{tl}(\mathit{seq}(t))\big], \mathit{phase}')} \ (\mathtt{signal})$$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{v.wait}() \quad \mathit{phase}(t)(p) = (\mathtt{v}, (w_t, s_t)) \quad \mathit{phase}' = \mathit{phase}[t \leftarrow \{p \leftarrow (\mathtt{v}, (1+w_t, s_t))\}] \quad \forall u \in \mathcal{T}, var \in \mathtt{V}^{\{-\}}.\ (\mathit{phase}(u)(p) = (var, (w_u, s_u)) \Rightarrow w_t < s_u)}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v.wait}()]{t} (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}\big[t \leftarrow \mathtt{tl}(\mathit{seq}(t))\big], \mathit{phase}')} \ (\mathtt{wait})$$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{v.next}() \quad \mathit{seq}' = \mathit{seq}\big[t \leftarrow \mathtt{v.signal}(); \mathtt{v.wait}(); \mathtt{tl}(\mathit{seq}(t))\big]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v.next}()]{t} (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}', \mathit{phase})} \ (\mathtt{next})$$

$$\frac{\mathcal{U} = \big\{u \mid \mathtt{isReg}(\mathit{phase}, u, p)\big\} \quad \forall u \in \mathcal{U}.\, \mathtt{hd}(\mathit{seq}(u)) = \mathtt{v.next}()\{\mathtt{stmt}\} \quad t \in \mathcal{U} \quad \mathit{seq}' = \mathit{seq}\big[t \leftarrow \mathtt{stmt}; \mathtt{v.next}(); \mathtt{tl}(\mathit{seq}(t))\big]\big[\{u \leftarrow \mathtt{v.next}(); \mathtt{tl}(\mathit{seq}(u)) \mid u \in \mathcal{U} \setminus \{t\}\}\big]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v.next}()\{\mathtt{stmt}\}]{t} (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}', \mathit{phase})} \ (\mathtt{next}\{\mathtt{stmt}\})$$

$$\frac{\mathtt{hd}(\mathit{seq}(t)) = \mathtt{v.drop}() \quad \mathit{phase}(t)(p) = (\mathtt{v}, (w, s)) \quad \mathit{phase}' = \mathit{phase}\big[t \leftarrow \mathit{phase}(t)[p \leftarrow (\mathtt{v}, \mathtt{nreg})]\big]}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase}) \xrightarrow[\mathtt{v.drop}()]{t} (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}\big[t \leftarrow \mathtt{tl}(\mathit{seq}(t))\big], \mathit{phase}')} \ (\mathtt{drop})$$

Fig. 3. Operational semantics of phaser statements. Each transition corresponds to a task $t \in \mathcal{T}$ executing a statement from a configuration $(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$. For instance, the `drop` transition corresponds to a task t executing `v.drop()` when registered to phaser $p \in \mathcal{P}$ (with phases (w, s)) and refering to it with variable `v`. The result is the same configuration where task t moves to its next statement without being registered to p.

resulting in configuration c'. See Fig. 3 for a description of phaser semantics. We write $c \xrightarrow[\mathtt{stmt}]{} c'$ if $c \xrightarrow[\mathtt{stmt}]{t} c'$ for some task t and $c \to c'$ if $c \xrightarrow[\mathtt{stmt}]{} c'$ for some `stmt`. We write $\xrightarrow[\mathtt{stmt}]{}^{+}$ for the transitive closure of $\xrightarrow[\mathtt{stmt}]{}$ and let $\to^*$ be the reflexive transitive closure of $\to$. Figure 2 identifies erroneous configurations.

We are interested in the reachability of sets of configurations (i.e., checking safety properties). We differentiate between two reachability problems depending on whether the target sets of configurations constrain the registration phases or not. The *plain reachability* problem may constrains the registration phases of the target configurations. The *control reachability* problem may not. We will see that decidability of the two problems can be different. The two problems are defined in the following.

Plain reachability. First, we define equivalent configurations. A configuration $c = (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ is equivalent to configuration $c' = (\mathcal{T}', \mathcal{P}', \mathit{bv}', \mathit{seq}', \mathit{phase}')$ if $\mathit{bv} = \mathit{bv}'$ and there are bijections $\tau : \mathcal{T} \to \mathcal{T}'$ and $\pi : \mathcal{P} \to \mathcal{P}'$ such that, for

all $t \in \mathcal{T}$, $p \in \mathcal{P}$ and $var \in \mathtt{V}^{\{-\}}$, $\mathit{seq}(t) = \mathit{seq}'(\tau(t))$ and there are some integers $(k_p)_{p\in\mathcal{P}}$ such that $\mathit{phase}(t)(p) = (var, (w, s))$ iff $\mathit{phase}'(\tau(t))(\pi(p)) = (var, (w + k_p, s + k_p))$. We write $c \sim c'$ to mean that c and c' are equivalent. Intuitively, equivalent configurations simulate each other. We can establish the following:

Lemma 1 (Equivalence). *Assume two configurations c_1 and c_2. If $c_1 \to c_2$ and $c_1' \sim c_1$ then there is a configuration c_2' s.t. $c_2' \sim c_2$ and $c_1' \to c_2'$.*

Observe that if the wait value of a task t on a phaser p is equal to the signal of a task t' on the same phaser p in some configuration c, then this is also the case, up to a renaming of the phasers and tasks, in all equivalent configurations. This is particularly relevant for defining deadlock configurations where a number of tasks are waiting for each other. The plain reachability problem is given a program and a target partial configuration and asks whether a configuration (equivalent to a configuration) that includes the target partial configuration is reachable.
More formally, given a program `prg` and a partial configuration c, let c_{init} be the initial configuration of `prg`, then $\mathtt{reach}(\mathtt{prg}, c)$ if and only if $c_{init} \to^* c_1$ for $c_1 \sim c_2$ and c_2 includes c.

Definition 1 (Plain reachability). *For a program* `prg` *and a partial configuration c, decide whether* $\mathtt{reach}(\mathtt{prg}, c)$ *holds.*

Control reachability. A partial configuration $c = (\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{phase})$ is said to be a *control partial configuration* if for all $t \in \mathcal{T}$ and $p \in \mathcal{P}$, either $\mathit{phase}(t)(p)$ is undefined or $\mathit{phase}(t)(p) \in (\mathtt{V}^{\{-,*\}} \times \{(*, *), \mathtt{nreg}\})$. Intuitively, control partial configurations do not constrain phase values. They are enough to characterize, for example, configurations where an assertion is violated (see Fig. 2).

Definition 2 (Control reachability). *For a program* `prg` *and a control partial configuration c, decide whether* $\mathtt{reach}(\mathtt{prg}, c)$ *holds.*

Observe that plain reachability is at least as hard to answer as control reachability since any control partial configuration is also a partial configuration. It turns out the control reachability problem is undecidable for programs resulting in arbitrarily many tasks and phasers as stated by the theorem below. This is proven by reduction of the state reachability problem for 2-counter Minsky machines. A 2-counter Minsky machine $(S, \{x_1, x_2\}, \Delta, s_0, s_F)$ has a finite set S of states, two counters $\{x_1, x_2\}$ with values in $\mathbb{N}$, an initial state s_0 and a final state s_F. Transitions may increment, decrement or test a counter. For example $(s_0, \mathrm{test}(x_1), s_F)$ takes the machine from s_0 to s_F if the counter x_1 is zero.

Theorem 1 (Minsky machines [14]). *Checking whether s_F is reachable from configuration $(s_0, 0, 0)$ for 2-counter machines is undecidable in general.*

Theorem 2. *Control reachability is undecidable in general.*

Proof sketch. State reachability of an arbitrary 2-counters Minsky machine is encoded as the control reachability problem of a phaser program. The phaser

program (see [10]) has three tasks `main`, `xUnit` and `yUnit`. It uses Boolean shared variables to encode the state $s \in S$ and to pass information between different task instances. The phaser program builds two chains, one with `xUnit` instances for the x-counter, and one with `yUnit` instances for the y-counter. Each chain alternates a phaser and a task and encodes the values of its counter with its length. The idea is to have the phaser program simulate all transitions of the counter machine, i.e., increments, decrements and tests for zero. Answering state reachability of the counter machine amounts to checking whether there are reachable configurations where the boolean variables encoding the counter machine can evaluate to the target machine state s_F.

4 A Gap-Based Symbolic Representation

The symbolic representation we propose builds on the following intuitions. First, observe the language semantics impose, for each phaser, the invariant that signal values are always larger or equal to wait values. We can therefore assume this fact in our symbolic representation. In addition, our reachability problems from Sect. 3 are defined in terms of reachability of equivalence classes, not of individual configurations. This is because configurations violating considered properties (see Fig. 2) are not defined in terms of concrete phase values but rather in terms of relations among them (in addition to the registration status, control sequences and variable values). Finally, we observe that if a wait is enabled with smaller gaps on a given phaser, then it will be enabled with larger ones. We therefore propose to track the gaps of the differences between signal and wait values wrt. to an existentially quantified level (per phaser) that lies between wait and signal values of all tasks registered to the considered phaser.

We formally define our symbolic representation and describe a corresponding entailment relation. We also establish a desirable property (namely being a $\mathcal{WQO}$, i.e., well-quasi-ordering [1,8]) on some classes of representations. This is crucial for the decidability of certain reachability problems (see Sect. 5).

Named gaps. A *named gap* is associated to a task-phaser pair. It consists in a tuple (var, val) in $\mathbb{G} = \left(\mathtt{V}^{\{-,*\}} \times \left(\left(\mathbb{N}^4 \cup \left(\mathbb{N}^2 \times \{\infty\}^2\right)\right) \cup \{\mathtt{nreg}\}\right)\right)$. Like for partial configurations in Sect. 3, $var \in \mathtt{V}^{\{-,*\}}$ constrains variable values. The val value describes task registration to the phaser. If registered, then val is a 4-tuple $(\mathtt{lw}, \mathtt{ls}, \mathtt{uw}, \mathtt{us})$. This intuitively captures, together with some level l common to all tasks registered to the considered phaser, all concrete wait and signal values (w, s) satisfying $\mathtt{lw} \leq (l - w) \leq \mathtt{uw}$ and $\mathtt{ls} \leq (s - l) \leq \mathtt{us}$. A named gap $(var, (\mathtt{lw}, \mathtt{ls}, \mathtt{uw}, \mathtt{us}))$ is said to be *free* if $\mathtt{uw} = \mathtt{us} = \infty$. It is said to be *B-gap-bounded*, for $B \in \mathbb{N}$, if both $\mathtt{uw} \leq B$ and $\mathtt{us} \leq B$ hold. A set $\mathcal{G} \subseteq \mathbb{G}$ is said to be free (resp. B-gap-bounded) if all its named gaps are free (resp. B-gap-bounded). The set $\mathcal{G}$ is said to be *B-good* if each one of its named gaps is either free or B-gap-bounded. Finally, $\mathcal{G}$ is said to be *good* if it is B-good for some $B \in \mathbb{N}$. Given a set $\mathcal{G}$ of named gaps, we define the partial order $\trianglelefteq$ on $\mathcal{G}$, and write $(var, val) \trianglelefteq (var', val')$, to mean (i) $(var \neq var' \Rightarrow var = *)$, and (ii)

$(val = \texttt{nreg}) \iff (val' = \texttt{nreg})$, and (iii) if $val = (\texttt{lw}, \texttt{ls}, \texttt{uw}, \texttt{us})$ and $val' = (\texttt{lw}', \texttt{ls}', \texttt{uw}', \texttt{us}')$ then $\texttt{lw} \leq \texttt{lw}'$, $\texttt{ls} \leq \texttt{ls}'$, $\texttt{uw}' \leq \texttt{uw}$ and $\texttt{us}' \leq \texttt{us}$. Intuitively, named gaps are used in the definition of constraints to capture relations (i.e., reference, registration and possible phases) of tasks and phasers. The partial order $(var, val) \trianglelefteq (var', val')$ ensures relations allowed by (var', val') are also allowed by (var, val).

Constraints. A constraint ϕ of $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ is a tuple $\big(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{gap}, \mathit{egap}\big)$ that denotes a possibly infinite set of configurations. Intuitively, $\mathcal{T}$ and $\mathcal{P}$ respectively represent a minimal set of tasks and phasers that are required in any configuration denoted by the constraint. In addition:

- $\mathit{bv} : \texttt{B} \to \mathbb{B}^{\{*\}}$ and $\mathit{seq} : \mathcal{T} \to \texttt{UnrSuff}^{\{*\}}$ respectively represent, like for partial configurations, a valuation of the shared Boolean variables and a mapping of tasks to their control sequences.
- $\mathit{gap} : \mathcal{T} \to \texttt{totalFunctions}\,(\mathcal{P}, \mathbb{G})$ constrains relations between $\mathcal{T}$-tasks and $\mathcal{P}$-phasers by associating to each task t a mapping $\mathit{gap}(t)$ that defines for each phaser p a named gap $(var, val) \in \mathbb{G}$ capturing the relation of t and p.
- $\mathit{egap} : \mathcal{P} \to \mathbb{N}^2$ associates *lower bounds* $(\texttt{ew}, \texttt{es})$ on gaps of tasks that are registered to $\mathcal{P}$-phasers but which are not explicitly captured by $\mathcal{T}$. This is described further in the constraints denotations below.

We write $\texttt{isReg}(\mathit{gap}, t, p)$ to mean the task t is registered to the phaser p, i.e., $\mathit{gap}(t)(p) \notin (\texttt{V}^{\{-,*\}} \times \{\texttt{nreg}\})$. A constraint ϕ is said to be free (resp. B-gap-bounded or B-good) if the set $\mathcal{G} = \big\{\mathit{gap}(t)(p) \mid t \in \mathcal{T}, p \in \mathcal{P}\big\}$ is free (resp. B-gap-bounded or B-good). The dimension of a constraint is the number of phasers it requires (i.e., $|\mathcal{P}|$). A set of constraints Φ is said to be free, B-gap-bounded, B-good or K-dimension-bounded if each of its constraints are.

Denotations. We write $c \models \phi$ to mean constraint $\phi = \Big(\mathcal{T}_\phi, \mathcal{P}_\phi, \mathit{bv}_\phi, \mathit{seq}_\phi, \mathit{gap}_\phi, \mathit{egap}_\phi\Big)$ denotes configuration $c = \big(\mathcal{T}_c, \mathcal{P}_c, \mathit{bv}_c, \mathit{seq}_c, \mathit{phase}_c\big)$. Intuitively, the configuration c should have at least as many tasks (captured by a surjection τ from a subset $\mathcal{T}_c^1$ of $\mathcal{T}_c$ to $\mathcal{T}_\phi$) and phasers (captured by a bijection π from a subset $\mathcal{P}_c^1$ of $\mathcal{P}_c$ to $\mathcal{P}_\phi$). Constraints on the tasks and phasers in $\mathcal{T}_c^1$ and $\mathcal{P}_c^1$ ensure target configurations are reachable. Additional constraints on the tasks in $\mathcal{T}_c^2 = \mathcal{T}_c \setminus \mathcal{T}_c^1$ ensure this reachability is not blocked by tasks not captured by $\mathcal{T}_\phi$. More formally:

1. for each $\texttt{b} \in \texttt{B}$, $(\mathit{bv}_\phi(\texttt{b}) \neq \mathit{bv}_c(\texttt{b})) \implies (\mathit{bv}_\phi(\texttt{b}) = *)$, and
2. $\mathcal{T}_c$ and $\mathcal{P}_c$ can be written as $\mathcal{T}_c = \mathcal{T}_c^1 \uplus \mathcal{T}_c^2$ and $\mathcal{P}_c = \mathcal{P}_c^1 \uplus \mathcal{P}_c^2$, with
3. $\tau : \mathcal{T}_c^1 \to \mathcal{T}_\phi$ is a surjection and $\pi : \mathcal{P}_c^1 \to \mathcal{P}_\phi$ is a bijection, and
4. for $t_c \in \mathcal{T}_c^1$ with $t_\phi = \tau(t_c)$, $(\mathit{seq}_\phi(t_\phi) \neq \mathit{seq}_c(t_c)) \implies (\mathit{seq}_\phi(t_\phi) = *)$, and
5. for each $p_\phi = \pi(p_c)$, there is a natural level $\ell : 0 \leq \ell$ such that:
   (a) if $t_c \in \mathcal{T}_c^1$ with $t_\phi = \tau(t_c)$, $\mathit{phase}_c(t_c)(p_c) = (var_c, val_c)$ and $\mathit{gap}_\phi(t_\phi)(p_\phi) = (var_\phi, val_\phi)$, then it is the case that:
      i. $(var_c \neq var_\phi) \implies (var_\phi = *)$, and

ii. $(val_c = \texttt{nreg}) \iff (val_\phi = \texttt{nreg})$, and
iii. if $(val_c = (w, s))$ and $(val_\phi = (\texttt{lw}, \texttt{ls}, \texttt{uw}, \texttt{us}))$ then $\texttt{lw} \leq \ell - w \leq \texttt{uw}$ and $\texttt{ls} \leq s - \ell \leq \texttt{us}$.

(b) if $t_c \in \mathcal{T}_c^2$, then for each $p_\phi = \pi(p_c)$ with $\mathit{phase}_c(t_c)(p_c) = (var_c, (w, s))$ and $\mathit{egap}(p_\phi) = (\texttt{ew}, \texttt{es})$, we have: $(\texttt{es} \leq s - \ell)$ and $(\texttt{ew} \leq \ell - w)$

Intuitively, for each phaser, the bounds given by gap constrain the values of the phases belonging to tasks captured by $\mathcal{T}_\phi$ (i.e., those in $\mathcal{T}_c^1$) and registered to the given phaser. This is done with respect to some non-negative level, one per phaser. The same level is used to constrain phases of tasks registered to the phaser but not captured by $\mathcal{T}_\phi$ (i.e., those in $\mathcal{T}_c^2$). For these tasks, lower bounds are enough as we only want to ensure they do not block executions to target sets of configurations. We write $[\![\phi]\!]$ for $\{c \mid c \models \phi\}$.

Entailment. We write $\phi_a \sqsubseteq \phi_b$ to mean $\phi_a = \left(\mathcal{T}_a, \mathcal{P}_a, \mathit{bv}_a, \mathit{seq}_a, \mathit{gap}_a, \mathit{egap}_a\right)$ is entailed by $\phi_b = \left(\mathcal{T}_b, \mathcal{P}_b, \mathit{bv}_b, \mathit{seq}_b, \mathit{gap}_b, \mathit{egap}_b\right)$. This will ensure that configurations denoted by ϕ_b are also denoted by ϕ_a. Intuitively, ϕ_b should have at least as many tasks (captured by a surjection τ from a subset $\mathcal{T}_b^1$ of $\mathcal{T}_b$ to $\mathcal{T}_a$) and phasers (captured by a bijection π from a subset $\mathcal{P}_b^1$ of $\mathcal{P}_b$ to $\mathcal{P}_a$). Conditions on tasks and phasers in $\mathcal{T}_b^1$ and $\mathcal{P}_b^1$ ensure the conditions in ϕ_a are met. Additional conditions on the tasks in $\mathcal{T}_b^2 = \mathcal{T}_b \setminus \mathcal{T}_b^1$ ensure at least the egap_a conditions in ϕ_a are met. More formally:

1. $(\mathit{bv}_a(\texttt{b}) \neq \mathit{bv}_b(\texttt{b})) \implies (\mathit{bv}_a(\texttt{b}) = *)$, for each $\texttt{b} \in \texttt{B}$ and
2. $\mathcal{T}_b$ and $\mathcal{P}_b$ can be written as $\mathcal{T}_b = \mathcal{T}_b^1 \uplus \mathcal{T}_b^2$ and $\mathcal{P}_b = \mathcal{P}_b^1 \uplus \mathcal{P}_b^2$ with
3. $\tau : \mathcal{T}_b^1 \to \mathcal{T}_a$ is a surjection and $\pi : \mathcal{P}_b^1 \to \mathcal{P}_a$ is a bijection, and
4. $(\mathit{seq}_b(t_b) \neq \mathit{seq}_a(t_a)) \implies (\mathit{seq}_b(t_b) = *)$ for each $t_b \in \mathcal{T}_b^1$ with $t_a = \tau(t_b)$, and
5. for each phaser $p_a = \pi(p_b)$ in $\mathcal{P}_a$:
   (a) if $\mathit{egap}_a(p_a) = (\texttt{ew}_a, \texttt{es}_a)$ and $\mathit{egap}_b(p_b) = (\texttt{ew}_b, \texttt{es}_b)$ then $\texttt{ew}_a \leq \texttt{ew}_b$ and $\texttt{es}_a \leq \texttt{es}_b$
   (b) for each $t_b \in \mathcal{T}_b^1$ with $t_a = \tau(t_b)$ and $\mathit{gap}_a(t_a)(p_a) = (var_a, val_a)$, and $\mathit{gap}_b(t_b)(p_b) = (var_b, val_b)$, it is the case that:
      i. $(var_b \neq var_a) \implies (var_a = *)$, and
      ii. $(val_b = \texttt{nreg}) \iff (val_a = \texttt{nreg})$, and
      iii. if $val_a = (\texttt{lw}_a, \texttt{ls}_a, \texttt{uw}_a, \texttt{us}_a)$ and $val_b = (\texttt{lw}_b, \texttt{ls}_b, \texttt{uw}_b, \texttt{us}_b)$, then $(\texttt{lw}_a \leq \texttt{lw}_b)$, $(\texttt{ls}_a \leq \texttt{ls}_b)$, $(\texttt{uw}_b \leq \texttt{uw}_a)$ and $(\texttt{us}_b \leq \texttt{us}_a)$.
   (c) for each $t_b \in \mathcal{T}_b^2$ with $\mathit{gap}_b(t_b)(p_b) = (var, (\texttt{lw}_a, \texttt{ls}_a, \texttt{uw}_a, \texttt{us}_a))$, with $\mathit{egap}_a(p_a) = (\texttt{ew}_a, \texttt{es}_a)$, both $(\texttt{ew}_a \leq \texttt{lw}_b)$ and $(\texttt{es}_a \leq \texttt{ls}_b)$ hold.

The following lemma shows that it is safe to eliminate entailing constraints in the working list procedure of Sect. 5.

Lemma 2 (Constraint entailment). *$\phi_a \sqsubseteq \phi_b$ implies $[\![\phi_b]\!] \subseteq [\![\phi_a]\!]$*

A central contribution that allows establishing the positive results of Sect. 5 is to show $\sqsubseteq$ is actually $\mathcal{WQO}$ on any K-dimension-bounded and B-good set of constraints. For this, we prove $\left(\mathcal{M}\left(\texttt{UnrSuff} \times \mathcal{G}^K\right), \exists\preceq\forall\right)$ is $\mathcal{WQO}$ if $\mathcal{G}$ is B-good, where $\mathcal{M}\left(\texttt{UnrSuff} \times \mathcal{G}^K\right)$ is the set of multisets over $\texttt{UnrSuff} \times \mathcal{G}^K$

and $M \ {}_{\exists}\preceq_{\forall} \ M'$ requires each $(\mathtt{s}', g'_1, \ldots, g'_K) \in M'$ may be mapped to some $(\mathtt{s}, g_1, \ldots, g_K) \in M$ for which $\mathtt{s} = \mathtt{s}'$ and $g_i \trianglelefteq g'_i$ for each $i : 1 \leq i \leq K$ (written $(\mathtt{s}, g_1, \ldots, g_K) \preceq (\mathtt{s}', g'_1, \ldots, g'_K)$). Intuitively, we need to use ${}_{\exists}\preceq_{\forall}$, and not simply ${}_{\forall}\preceq_{\exists}$, in order to "cover all registered tasks" in the larger constraint as otherwise some tasks may block the path to the target configurations. Rado's structure [12,13] shows that, in general, $(\mathcal{M}(S), \ {}_{\exists}\preceq_{\forall})$ need not be $\mathcal{WQO}$ just because $\preceq$ is $\mathcal{WQO}$ over S. The proof details can be found in [10].

Theorem 3. *$(\Phi, \sqsubseteq)$ is $\mathcal{WQO}$ if Φ is K-dimension-bounded and B-good for some pre-defined $K, B \in \mathbb{N}$.*

5 A Symbolic Verification Procedure

We use the constraints defined in Sect. 4 as a symbolic representation in an instantiation of the classical framework of Well-Structured-Transition-Systems [1,8]. The instantiation (described in [10]) is a working-list procedure that takes as arguments a program `prg` and a $\sqsubseteq$-minimal set Φ of constraints denoting the targeted set of configurations. Such constraints can be easily built from the partial configurations described in Fig. 2.

The procedure computes a fixpoint using the entailment relation of Sect. 4 and a predecessor computation that results, for a constraint ϕ and a statement `stmt`, in a finite set $\mathtt{pre}_{\mathtt{stmt}} = \left\{\phi' \mid \phi \xrightarrow[\mathtt{stmt}]{}\!\!\!\!\rightarrow \phi'\right\}$. Figure 4 describes part of the computation for the `v.signal()` instruction (see [10] for other instructions). For all but atomic statements, the set $\mathtt{pre}_{\mathtt{stmt}} = \left\{\phi' \mid \phi \xrightarrow[\mathtt{stmt}]{}\!\!\!\!\rightarrow \phi'\right\}$ is exact in the sense that $\left\{c' \mid c \in [\![\phi]\!] \text{ and } c' \xrightarrow[\mathtt{stmt}]{} c\right\} \subseteq \bigcup_{\phi' \in \mathtt{pre}_{\mathtt{stmt}}} [\![\phi']\!] \subseteq \left\{c' \mid c \in [\![\phi]\!] \text{ and } c' \xrightarrow[\mathtt{stmt}]{}{}^{+} c\right\}$. Intuitively, the predecessors calculation for the atomic `v.next(){stmt}` is only an over-approximation because such an instruction can encode a test-and-set operation. Our representation allows for more tasks, but the additional tasks may not be able to carry the atomic operation. We would therefore obtain a non-exact over-approximation and avoid this issue by only applying the procedure to non-atomic programs. We can show the following theorems.

$$\frac{\begin{array}{c}\mathtt{s}', \mathtt{s} \in \mathtt{UnrSuff} \quad \mathtt{s}' = \mathtt{v.signal}(); \mathtt{s} \quad \mathit{seq}' = \mathit{seq}[t \leftarrow \mathtt{s}'] \quad \mathit{gap}(t)(p) = (\mathtt{v}, (\mathtt{lw}_t, \mathtt{ls}_t, \mathtt{uw}_t, \mathtt{us}_t) \\ \mathcal{U} = \{u \mid u \in \mathcal{T} \text{ and } \mathit{gap}(u)(p) = (\mathit{var}_u, (\mathtt{lw}_u, \mathtt{ls}_u, \mathtt{uw}_u, \mathtt{us}_u))\} \quad \text{for each } u \in \mathcal{U}.\ \mathtt{uw}_u \geq 1 \\ \mathit{gap}_1 = \mathit{gap}[\{u \leftarrow \mathit{gap}(u)[p \leftarrow (\mathit{var}_u, ((\mathtt{lw}_u - 1)^+, \mathtt{ls}_u + 1, \mathtt{uw}_u - 1, \mathtt{us}_u + 1))] \mid u \in \mathcal{U} \setminus \{t\}\}] \\ \mathit{gap}' = \mathit{gap}_1[t \leftarrow \mathit{gap}_1[p \leftarrow (\mathit{var}_t, ((\mathtt{lw}_t - 1)^+, \mathtt{ls}_t, \mathtt{uw}_t - 1, \mathtt{us}_t))]] \quad \mathit{egap}' = \mathit{egap}[p \leftarrow (\mathtt{ew} - 1)^+, \mathtt{es} + 1]\end{array}}{(\mathcal{T}, \mathcal{P}, \mathit{bv}, \mathit{seq}, \mathit{gap}, \mathit{egap}) \xrightarrow[\mathtt{v.signal()}]{}\!\!\!\!\rightarrow (\mathcal{T}', \mathcal{P}', \mathit{bv}, \mathit{seq}', \mathit{gap}', \mathit{egap}')}$$

Fig. 4. Part of the predecessors computation for the `v.signal()` phaser statement where x^+ stands for $max(0, x)$. Task $t \in \mathcal{T}$ is registered to $p \in \mathcal{P}$ and refers to it with `v` with all registered tasks having a non-zero upper bound on their waiting phases.

Theorem 4. *Control reachability is decidable for non-atomic phaser programs generating a finite number of phasers.*

The idea is to systematically drop, in the instantiated backward procedure, constraints violating K-dimension-boundedness (as none of the denoted configurations is reachable). Also, the set of target constraints is free (since we are checking control reachability) and this is preserved by the predecessors computation (see [10]). Finally, we use the exactness of the $\texttt{pre}_{\texttt{stmt}}$ computation, the soundness of the entailment relation and Theorem 3. We can use a similar reasoning for plain reachability of programs generating a finite number of phasers and bounded gap-values for each phaser.

Theorem 5. *Plain reachability is decidable for non-atomic phaser programs generating a finite number of phasers with, for each phaser, bounded phase gaps.*

6 Limitations of Deciding Reachability

Assume a program $\texttt{prg} = (\texttt{B}, \texttt{V}, \texttt{T})$ and its initial configuration c_{init}. We show a number of parameterized reachability problems to be undecidable. First, we address checking control reachability when restricting to configurations with at most K task-referenced phasers. We call this K-control-reachability.

Definition 3 (K-control-reachability). *Given a partial control configuration c, we write $\texttt{reach}_K(\texttt{prg}, c)$, and say c is K-control-reachable, to mean there are $n+1$ configurations $(c_i)_{i:0\leq i\leq n}$, each with at most K reachable phasers (i.e., phasers referenced by at least a task variable) s.t. $c_{init} = c_0$ and $c_i \longrightarrow c_{i+1}$ for $i : 0 \leq i < n-1$ with c_n equivalent to a configuration that includes c.*

Theorem 6. *K-control-reachability is undecidable in general.*

Proof sketch. Encode state reachability of an arbitrary Minsky machine with counters x and y using K-control-reachability of a suitable phaser program. The program (see [10]) has five tasks: `main`, `xTask`, `yTask`, `child1` and `child2`. Machine states are captured with shared variables and counter values with phasers `xPh` for counter x (resp. `yPh` for counter y). Then, (1) spawn an instance of `xTask` (resp. `yTask`) and register it to `xPh` (resp. `yPh`) for increments, and (2) perform a wait on `xPh` (resp. `yPh`) to test for zero. Decrementing a counter, say x, involves asking an `xTask`, via shared variables, to exit (hence, to deregister from `xPh`). However, more than one task might participate in the decrement operation. For this reason, each participating task builds a path from `xPh` to `child2` with two phasers. If more than one `xTask` participates in the decrement, then the number of reachable phasers of an intermediary configuration will be at least five.

Theorem 7. *Control reachability is undecidable if atomic statements are allowed even if only a finite number of phasers is generated.*

Proof sketch. Encode state reachability of an arbitrary Minsky machine with counters x and y using a phaser program with atomic statements. The phaser program (see [10]) has three tasks: `main`, `xTask` and `yTask` and encodes machine states with shared variables. The idea is to associate a phaser `xPh` to counter x (resp. `yPh` to y) and to perform a signal followed by a wait on `xPh` (resp. `yPh`) to test for zero. Incrementing and decrementing is performed by asking spawned tasks to spawn a new instance or to deregister. Atomic-next statements are used to ensure exactly one task is spawned or deregistered.

Finally, even with finite numbers of tasks and phasers, but with arbitrary gap-bounds, we can show [9] the following.

Theorem 8. *Plain reachability is undecidable if generated gaps are not bounded even when restricting to non-atomic programs with finite numbers of phasers.*

Table 1. Findings summary: *ctrl* stands for control reachability and *plain* for plain reachability; *atomic* stands for allowing the `v.next(){stmt}` atomic instruction and *non-atomic* for forbidding it (resulting in non-atomic programs). Decidable problems are marked with ✓ and undecidable ones with ✗.

	Arbitrary numbers of tasks			
	Finite dimension		K-reachability	Arbitrary dimension
Bounded gaps	ctrl atomic ✗ (Theorem 7)	plain non-atomic ✓ (Theorem 5)	ctrl non-atomic ✗ (Theorem 6)	ctrl non-atomic ✗ (Theorem 2)
Arbitrary gaps	ctrl non-atomic ✓ (Theorem 4)	plain non-atomic ✗ (From [9])		

7 Conclusion

We have studied parameterized plain (e.g., deadlocks) and control (e.g., assertions) reachability problems. We have proposed an exact verification procedure for non-atomic programs. We summarize our findings in Table 1. The procedure is guaranteed to terminate, even for programs that may generate arbitrary many tasks but finitely many phasers, when checking control reachability or when checking plain reachability with bounded gaps. These results were obtained using a non-trivial symbolic representation for which termination had required showing an $_{\exists}\preceq_{\forall}$ preorder on multisets on gaps on natural numbers to be a $\mathcal{WQO}$. We are working on a tool that implements the procedure to verify phaser programs that dynamically spawn tasks. We believe our general decidability results are useful to reason about synchronization constructs other than phasers. For instance, a traditional static barrier can be captured with one phaser and with bounded gaps (in fact one). Similarly, one phaser with one producer and arbitrary many consumers can capture futures where "gets" are modeled with waits. Also, test-and-set operations can model atomic instructions and may result in undecidability of reachability. This suggests more general applications of the work are to be investigated.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science, LICS 1996, pp. 313–321. IEEE (1996)
2. Anderson, P., Chase, B., Mercer, E.: JPF verification of Habanero Java programs. SIGSOFT Softw. Eng. Notes **39**(1), 1–7 (2014). https://doi.org/10.1145/2557833.2560582. http://doi.acm.org/10.1145/2557833.2560582
3. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71316-6_22
4. Cavé, V., Zhao, J., Shirako, J., Sarkar, V.: Habanero-Java: the new adventures of old x10. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, pp. 51–61. ACM (2011)
5. Charles, P., et al.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**(10), 519–538 (2005). https://doi.org/10.1145/1103845.1094852. http://doi.acm.org/10.1145/1103845.1094852
6. Cogumbreiro, T., Hu, R., Martins, F., Yoshida, N.: Dynamic deadlock verification for general barrier synchronisation. In: 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 150–160. ACM, New York (2015). https://doi.org/10.1145/2688500.2688519. http://doi.acm.org/10.1145/2688500.2688519
7. Cogumbreiro, T., Shirako, J., Sarkar, V.: Formalization of Habanero phasers using Coq. J. Log. Algebraic Methods Program. **90**, 50–60 (2017). https://doi.org/10.1016/j.jlamp.2017.02.006. http://www.sciencedirect.com/science/article/pii/S2352220816300839
8. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere!. Theoret. Comput. Sci. **256**(1), 63–92 (2001). https://doi.org/10.1016/S0304-3975(00)00102-X. http://www.sciencedirect.com/science/article/pii/S030439750000102X
9. Ganjei, Z., Rezine, A., Eles, P., Peng, Z.: Safety verification of phaser programs. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design, FMCAD 2017, pp. 68–75. FMCAD Inc., Austin (2017). http://dl.acm.org/citation.cfm?id=3168451.3168471
10. Ganjei, Z., Rezine, A., Henrio, L., Eles, P., Peng, Z.: On reachability in parameterized phaser programs. arXiv:1811.07142 (2019)
11. Havelund, K., Pressburger, T.: Model checking Java programs using Java pathfinder. Int. J. Softw. Tools Technol. Transfer (STTT) **2**(4), 366–381 (2000)
12. Jancar, P.: A note on well quasi-orderings for powersets. Inf. Process. Lett. **72**(5–6), 155–160 (1999). https://doi.org/10.1016/S0020-0190(99)00149-0
13. Marcone, A.: Fine analysis of the quasi-orderings on the power set. Order **18**(4), 339–347 (2001). https://doi.org/10.1023/A:1013952225669
14. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Inc., Englewood Cliffs (1967)
15. Shirako, J., Peixotto, D.M., Sarkar, V., Scherer, W.N.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: 22nd Annual International Conference on Supercomputing, pp. 277–288. ACM (2008)
16. Valiant, L.G.: A bridging model for parallel computation. CACM **33**(8), 103 (1990)
17. Willebeek-LeMair, M.H., Reeves, A.P.: Strategies for dynamic load balancing on highly parallel computers. IEEE Trans. Parallel Distrib. Syst. **4**(9), 979–993 (1993)

Abstract Dependency Graphs and Their Application to Model Checking

Søren Enevoldsen, Kim Guldstrand Larsen, and Jiří Srba(✉)

Department of Computer Science,
Aalborg University, Selma Lagerlofs Vej 300,
9220 Aalborg East, Denmark
srba@cs.aau.dk

Abstract. Dependency graphs, invented by Liu and Smolka in 1998, are oriented graphs with hyperedges that represent dependencies among the values of the vertices. Numerous model checking problems are reducible to a computation of the minimum fixed-point vertex assignment. Recent works successfully extended the assignments in dependency graphs from the Boolean domain into more general domains in order to speed up the fixed-point computation or to apply the formalism to a more general setting of e.g. weighted logics. All these extensions require separate correctness proofs of the fixed-point algorithm as well as a one-purpose implementation. We suggest the notion of *abstract dependency graphs* where the vertex assignment is defined over an abstract algebraic structure of Noetherian partial orders with the least element. We show that existing approaches are concrete instances of our general framework and provide an open-source C++ library that implements the abstract algorithm. We demonstrate that the performance of our generic implementation is comparable to, and sometimes even outperforms, dedicated special-purpose algorithms presented in the literature.

1 Introduction

Dependency Graphs (DG) [1] have demonstrated a wide applicability with respect to verification and synthesis of reactive systems, e.g. checking behavioural equivalences between systems [2], model checking systems with respect to temporal logical properties [3–5], as well as synthesizing missing components of systems [6]. The DG approach offers a general and often performance-optimal way to solve these problem. Most recently, the DG approach to CTL model checking of Petri nets [7], implemented in the model checker TAPAAL [8], won the gold medal at the annual Model Checking Contest 2018 [9].

A DG consists of a finite set of vertices and a finite set of hyperedges that connect a vertex to a number of children vertices. The computation problem is to find a point-wise minimal assignment of vertices to the Boolean values 0 and 1 such that the assignment is stable: whenever there is a hyperedge where all children have the value 1 then also the father of the hyperedge has the value 1. The main contribution of Liu and Smolka [1] is a linear-time, on-the-fly algorithm to find such a minimum stable assignment.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 316–333, 2019.
https://doi.org/10.1007/978-3-030-17462-0_18

Recent works successfully extend the DG approach from the Boolean domain to more general domains, including synthesis for timed systems [10], model checking for weighted systems [3] as well as probabilistic systems [11]. However, each of these extensions have required separate correctness arguments as well as ad-hoc specialized implementations that are to a large extent similar with other implementations of dependency graphs (as they are all based on the general principle of computing fixed points by local exploration). The contribution of our paper is a notion of Abstract Dependency Graph (ADG) where the values of vertices come from an abstract domain given as an Noetherian partial order (with least element). As we demonstrate, this notion of ADG covers many existing extensions of DG as concrete instances. Finally, we implement our abstract algorithms in C++ and make it available as an open-source library. We run a number of experiments to justify that our generic approach does not sacrifice any significant performance and sometimes even outperforms existing implementations.

Related Work. The aim of Liu and Smolka [1] was to find a unifying formalism allowing for a local (on-the-fly) fixed-point algorithm running in linear time. In our work, we generalize their formalism from the simple Boolean domain to general Noetherian partial orders over potentially infinite domains. This requires a non-trivial extension to their algorithm and the insight of how to (in the general setting) optimize the performance, as well as new proofs of the more general loop invariants and correctness arguments.

Recent extensions of the DG framework with certain-zero [7], integer [3] and even probabilistic [11] domains generalized Liu and Smolka's approach, however they become concrete instances of our abstract dependency graphs. The formalism of Boolean Equation Systems (BES) provides a similar and independently developed framework [12–15] pre-dating that of DG. However, BES may be encoded as DG [1] and hence they also become an instance of our abstract dependency graphs.

2 Preliminaries

A set D together with a binary relation $\sqsubseteq \subseteq D \times D$ that is reflexive ($x \sqsubseteq x$ for any $x \in D$), transitive (for any $x, y, z \in D$, if $x \sqsubseteq y$ and $y \sqsubseteq x$ then also $x \sqsubseteq z$) and anti-symmetric (for any $x, y \in D$, if $x \sqsubseteq y$ and $y \sqsubseteq x$ then $x = y$) is called a *partial order* and denoted as a pair $(D, \sqsubseteq)$. We write $x \sqsubset y$ if $x \sqsubseteq y$ and $x \neq y$. A function $f : D \to D'$ from a partial order $(D, \sqsubseteq)$ to a partial order $(D', \sqsubseteq')$ is *monotonic* if whenever $x \sqsubseteq y$ for $x, y \in D$ then also $f(x) \sqsubseteq' f(y)$. We shall now define a particular partial order that will be used throughout this paper.

Definition 1 (NOR). Noetherian Ordering Relation with least element *(NOR) is a triple* $\mathcal{D} = (D, \sqsubseteq, \bot)$ *where* $(D, \sqsubseteq)$ *is a partial order,* $\bot \in D$ *is its least element such that for all* $d \in D$ *we have* $\bot \sqsubseteq d$*, and* $\sqsubseteq$ *satisfies the ascending chain condition: for any infinite chain* $d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \ldots$ *there is an integer* k *such that* $d_k = d_{k+j}$ *for all* $j > 0$.

We can notice that any finite partial order with a least element is a NOR; however, there are also such relations with infinitely many elements in the domain as shown by the following example.

Example 1. Consider the partial order $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \geq, \infty)$ over the set of natural numbers extended with ∞ and the natural larger-than-or-equal comparison on integers. As the relation is reversed, this implies that ∞ is the least element of the domain. We observe that $\mathcal{D}$ is NOR. Consider any infinite sequence $d_1 \geq d_2 \geq d_3 \ldots$. Then either $d_i = \infty$ for all i, or there exists i such that $d_i \in \mathbb{N}^0$. Clearly, the sequence must in both cases eventually stabilize, i.e. there is a number k such that $d_k = d_{k+j}$ for all $j > 0$.

New NORs can be constructed by using the Cartesian product. Let $\mathcal{D}_i = (D_i, \sqsubseteq_i, \bot_i)$ for all i, $1 \leq i \leq n$, be NORs. We define $\mathcal{D}^n = (D^n, \sqsubseteq^n, \bot^n)$ such that $D^n = D_1 \times D_2 \times \cdots \times D_n$ and where $(d_1, \ldots, d_n) \sqsubseteq^n (d'_1, \ldots, d'_n)$ if $d_i \sqsubseteq_i d'_i$ for all i, $1 \leq i \leq k$, and where $\bot^n = (\bot_1, \ldots, \bot_n)$.

Proposition 1. *Let $\mathcal{D}_i$ be a NOR for all i, $1 \leq i \leq n$. Then $\mathcal{D}^n = (D^n, \sqsubseteq^n, \bot^n)$ is also a NOR.*

In the rest of this paper, we consider only NOR $(D, \sqsubseteq, \bot)$ that are *effectively computable*, meaning that the elements of D can be represented by finite strings, and that given the finite representations of two elements x and y from D, there is an algorithm that decides whether $x \sqsubseteq y$. Similarly, we consider only functions $f : D \to D'$ from an effectively computable NOR $(D, \sqsubseteq, \bot)$ to an effectively computable NOR $(D', \sqsubseteq', \bot')$ that are *effectively computable*, meaning that there is an algorithm that for a given finite representation of an element $x \in D$ terminates and returns the finite representation of the element $f(x) \in D'$. Let $\mathcal{F}(\mathcal{D}, n)$, where $\mathcal{D} = (D, \sqsubseteq, \bot)$ is an effectively computable NOR and n is a natural number, stand for the collection of all effectively computable functions $f : D^n \to D$ of arity n and let $\mathcal{F}(\mathcal{D}) = \bigcup_{n \geq 0} \mathcal{F}(\mathcal{D}, n)$ be a collection of all such functions.

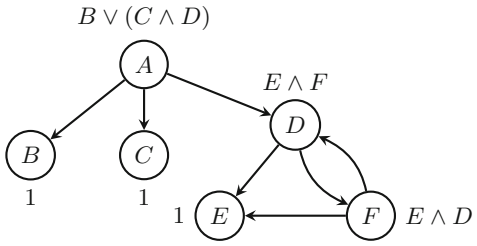
For a set X, let X^* be the set of all finite strings over X. For a string $w \in X^*$ let $|w|$ denote the length of w and for every i, $1 \leq i \leq |w|$, let w^i stand for the i'th symbol in w.

3 Abstract Dependency Graphs

We are now ready to define the notion of an abstract dependency graph.

Definition 2 (Abstract Dependency Graph). *An* abstract dependency graph *(ADG) is a tuple $G = (V, E, \mathcal{D}, \mathcal{E})$ where*

- *V is a finite set of vertices,*
- *$E : V \to V^*$ is an edge function from vertices to sequences of vertices such that $E(v)^i \neq E(v)^j$ for every $v \in V$ and every $1 \leq i < j \leq |E(v)|$, i.e. the co-domain of E contains only strings over V where no symbol appears more than once,*



(a) Abstract dependency graph

	A	B	C	D	E	F
$A_\perp$	0	0	0	0	0	0
$F(A_\perp)$	0	1	1	0	1	0
$F^2(A_\perp)$	1	1	1	0	1	0
$F^3(A_\perp)$	1	1	1	0	1	0

(b) Fixed-point computation

Fig. 1. Abstract dependency graph over NOR $(\{0,1\}, \leq, 0)$

- $\mathcal{D}$ *is an effectively computable NOR, and*
- $\mathcal{E}$ *is a labelling function* $\mathcal{E} : V \to \mathcal{F}(\mathcal{D})$ *such that* $\mathcal{E}(v) \in \mathcal{F}(\mathcal{D}, |E(v)|)$ *for each* $v \in V$*, i.e. each edge* $E(v)$ *is labelled by an effectively computable function* f *of arity that corresponds to the length of the string* $E(v)$*.*

Example 2. An example of ADG over the NOR $\mathcal{D} = (\{0,1\}, \{(0,1)\}, 0)$ is shown in Fig. 1a. Here 0 (interpreted as false) is below the value 1 (interpreted as true) and the monotonic functions for vertices are displayed as vertex annotations. For example $E(A) = B{\cdot}C{\cdot}D$ and $\mathcal{E}(A)$ is a ternary function such that $\mathcal{E}(A)(x, y, z) = x \vee (y \wedge z)$, and $E(B) = \epsilon$ (empty sequence of vertices) such that $\mathcal{E}(B) = 1$ is a constant labelling function. Clearly, all functions used in our example are monotonic and effectively computable.

Let us now assume a fixed ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ over an effectively computable NOR $\mathcal{D} = (D, \sqsubseteq, \perp)$. We first define an assignment of an ADG.

Definition 3 (Assignment). *An* assignment *on* G *is a function* $A : V \to D$*.*

The set of all assignments is denoted by $\mathcal{A}$. For $A, A' \in \mathcal{A}$ we define $A \leq A'$ iff $A(v) \sqsubseteq A'(v)$ for all $v \in V$. We also define the bottom assignment $A_\perp(v) = \perp$ for all $v \in V$ that is the least element in the partial order $(\mathcal{A}, \leq)$. The following proposition is easy to verify.

Proposition 2. *The partial order* $(\mathcal{A}, \leq, A_\perp)$ *is a NOR.*

Finally, we define the *minimum fixed-point assignment* A_{min} for a given ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ as the minimum fixed point of the function $F : \mathcal{A} \to \mathcal{A}$ defined as follows: $F(A)(v) = \mathcal{E}(v)(A(v_1), A(v_2), \ldots, A(v_k))$ where $E(v) = v_1 v_2 \ldots v_k$.

In the rest of this section, we shall argue that A_{min} of the function F exists by following the standard reasoning about fixed points of monotonic functions [16].

Lemma 1. *The function* F *is monotonic.*

Let us define the notation of multiple applications of the function F by $F^0(A) = A$ and $F^i(A) = F(F^{i-1}(A))$ for $i > 0$.

Lemma 2. *For all $i \geq 0$ the assignment $F^i(A_\perp)$ is effectively computable, $F^i(A_\perp) \leq F^j(A_\perp)$ for all $i \leq j$, and there exists a number k such that $F^k(A_\perp) = F^{k+j}(A_\perp)$ for all $j > 0$.*

We can now finish with the main observation of this section.

Theorem 1. *There exists a number k such that $F^j(A_\perp) = A_{min}$ for all $j \geq k$.*

Example 3. The computation of the minimum fixed point for our running example from Fig. 1a is given in Fig. 1b. We can see that starting from the assignment where all nodes take the least element value 0, in the first iteration all constant functions increase the value of the corresponding vertices to 1 and in the second iteration the value 1 propagates from the vertex B to A, because the function $B \vee (C \wedge D)$ that is assigned to the vertex A evaluates to true due to the fact that $F(A_\perp)(B) = 1$. On the other hand, the values of the vertices D and F keep the assignment 0 due to the cyclic dependencies between the two vertices. As $F^2(A_\perp) = F^3(A_\perp)$, we know that we found the minimum fixed point.

As many natural verification problems can be encoded as a computation of the minimum fixed point on an ADG, the result in Theorem 1 provides an algorithmic way to compute such a fixed point and hence solve the encoded problem. The disadvantage of this *global* algorithm is that it requires that the whole dependency graph is a priory generated before the computation can be carried out and this approach is often inefficient in practice [3]. In the following section, we provide a *local*, on-the-fly algorithm for computing the minimum fixed-point assignment of a specific vertex, without the need to always explore the whole abstract dependency graph.

4 On-the-Fly Algorithm for ADGs

The idea behind the algorithm is to progressively explore the vertices of the graph, starting from a given root vertex for which we want to find its value in the minimum fixed-point assignment. To search the graph, we use a waiting list that contains configurations (vertices) whose assignment has the potential of being improved by applying the function $\mathcal{E}$. By repeated applications of $\mathcal{E}$ on the vertices of the graph in some order maintained by the algorithm, the minimum fixed-point assignment for the root vertex can be identified without necessarily exploring the whole dependency graph.

To improve the performance of the algorithm, we make use of an optional user-provided function $\textsc{Ignore}(A, v)$ that computes, given a current assignment A and a vertex v of the graph, the set of vertices on an edge $E(v)$ whose current and any potential future value no longer effect the value of $A_{min}(v)$. Hence, whenever a vertex v' is in the set $\textsc{Ignore}(A, v)$, there is no reason to explore the subgraph rooted by v' for the purpose of computing $A_{min}(v)$ since an improved assignment value of v' cannot influence the assignment of v. The soundness property of the ignore function is formalized in the following definition. As before, we assume a fixed ADG $G = (V, E, \mathcal{D}, \mathcal{E})$ over an effectively computable NOR $\mathcal{D} = (D, \sqsubseteq, \perp)$.

Definition 4 (Ignore Function). *A function:* $\textsc{Ignore} : \mathcal{A} \times V \to 2^V$ *is* sound *if for any two assignments* $A, A' \in \mathcal{A}$ *where* $A \leq A'$ *and every* i *such that* $E(v)^i \in \textsc{Ignore}(A, v)$ *holds that*

$$\begin{aligned}&\mathcal{E}(v)(A'(v_1), A'(v_2), \ldots, A(v_i), \ldots, A'(v_{|E(v)-1|}), A'(v_{|E(v)|}))\\ &= \mathcal{E}(v)(A'(v_1), A'(v_2), \ldots, A'(v_i), \ldots, A'(v_{|E(v)-1|}), A'(v_{|E(v)|})).\end{aligned}$$

From now on, we shall consider only sound and effectively computable ignore functions. Note that there is always a trivially sound Ignore function that returns for every assignment and every vertex the empty set. A more interesting and universally sound ignore function may be defined by

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } d \leq A(v) \text{ for all } d \in D \\ \emptyset & \text{otherwise} \end{cases}$$

that returns the set of all vertices on an edge $E(v)$ once $A(v)$ reached its maximal possible value. This will avoid the exploration of the children of the vertex v once the value of v in the current assignment cannot be improved any more. Already this can have a significant impact on the improved performance of the algorithm; however, for concrete instances of our general framework, the user can provide more precise and case-specific ignore functions in order to tune the performance of the fixed-point algorithm, as shown by the next example.

Example 4. Consider the ADG from Fig. 1a in an assignment where the value of B is already known to be 1. As the vertex A has the labelling function $B \vee (C \wedge D)$, we can see that the assignment of A will get the value 1, irrelevant of what are the assignments for the vertices C and D. Hence, in this assignment, we can move the vertices C and D to the ignore set of A and avoid the exploration of the subgraphs rooted by C and D.

The following lemma formalizes the fact that once the ignore function of a vertex contains all its children and the vertex value has been relaxed by applying the associated monotonic function, then its current assignment value is equal to the vertex value in the minimum fixed-point assignment.

Lemma 3. *Let* A *be an assignment such that* $A \leq A_{min}$. *If* $v_i \in \textsc{Ignore}(A, v)$ *for all* $1 \leq i \leq k$ *where* $E(v) = v_1 \cdots v_k$ *and* $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ *then* $A(v) = A_{min}(v)$.

In Algorithm 1 we now present our local (on-the-fly) minimum fixed-point computation. The algorithm uses the following internal data structures:

- A is the currently computed assignment that is initialized to $A_\perp$,
- W is the waiting list containing the set of pending vertices to be explored,
- Passed is the set of explored vertices, and
- $Dep : V \to 2^V$ is a function that for each vertex v returns a subset of vertices that should be reevaluated whenever the assignment value of v improves.

```
   Input: An effectively computable ADG G = (V, E, D, E) and v_0 ∈ V.
   Output: A_min(v_0)
 1 A := A_⊥ ; Dep(v) := ∅ for all v
 2 W := {v_0} ; Passed := ∅
 3 while W ≠ ∅ do
 4 |   let v ∈ W ; W := W \ {v}
 5 |   UpdateDependents (v)
 6 |   if v = v_0 or Dep(v) ≠ ∅ then
 7 |   |   let v_1 v_2 ··· v_k = E(v)
 8 |   |   d := E(v)(A(v_1), ..., A(v_k))
 9 |   |   if A(v) ⊏ d then
10 |   |   |   A(v) := d
11 |   |   |   W := W ∪ {u ∈ Dep(v) | v ∉ Ignore(A, u)}
12 |   |   |   if v = v_0 and {v_1, ..., v_k} ⊆ Ignore(A, v_0) then
13 |   |   |   |   "break out of the while loop"
14 |   |   if v ∉ Passed then
15 |   |   |   Passed := Passed ∪ {v}
16 |   |   |   for all v_i ∈ {v_1, ..., v_k} \ Ignore(A, v) do
17 |   |   |   |   Dep(v_i) := Dep(v_i) ∪ {v}
18 |   |   |   |   W := W ∪ {v_i}
19 return A(v_0)
20 Procedure UpdateDependents(v):
21 |   C := {u ∈ Dep(v) | v ∈ Ignore(A, u)}
22 |   Dep(v) := Dep(v) \ C
23 |   if Dep(v) = ∅ and C ≠ ∅ then
24 |   |   Passed := Passed \ {v}
25 |   |   UpdateDependentsRec (v)
26 Procedure UpdateDependentsRec(v):
27 |   for v' ∈ E(v) do
28 |   |   Dep(v') := Dep(v') \ {v}
29 |   |   if Dep(v') = ∅ then
30 |   |   |   UpdateDependentsRec (v')
31 |   |   |   Passed := Passed \ {v'}
```

Algorithm 1. Minimum Fixed-Point Computation on an ADG

The algorithm starts by inserting the root vertex v_0 into the waiting list. In each iteration of the while-loop it removes a vertex v from the waiting list and performs a check whether there is some other vertex that depends on the value of v. If this is not the case, we are not going to explore the vertex v and recursively propagate this information to the children of v. After this, we try to improve the current assignment of $A(v)$ and if this succeeds, we update the waiting list by adding all vertices that depend on the value of v to W, and we test if the algorithm can early terminate (should the root vertex v_0 get its final value). Otherwise, if the vertex v has not been explored yet, we add all its children to the waiting list and update the dependencies. We shall now state the termination and correctness of our algorithm.

Lemma 4 (Termination). *Algorithm 1 terminates.*

Lemma 5 (Soundness). *Algorithm 1 at all times satisfies* $A \leq A_{min}$.

Lemma 6 (While-Loop Invariant). *At the beginning of each iteration of the loop in line 1 of Algorithm 1, for any vertex* $v \in V$ *it holds that either:*

1. $A(v) = A_{min}(v)$, *or*
2. $v \in W$, *or*
3. $v \neq v_0$ *and* $Dep(v) = \emptyset$, *or*
4. $A(v) = \mathcal{E}(v)(A(v_1), \ldots, A(v_k))$ *where* $v_1 \cdots v_k = E(v)$ *and for all* i, $1 \leq i \leq k$, *whenever* $v_i \notin \textsc{Ignore}(A, v)$ *then also* $v \in Dep(v_i)$.

Theorem 2. *Algorithm 1 terminates and returns the value* $A_{min}(v_0)$.

5 Applications of Abstract Dependency Graphs

We shall now describe applications of our general framework to previously studied settings in order to demonstrate the direct applicability of our framework. Together with an efficient implementation of the algorithm, this provides a solution to many verification problems studied in the literature. We start with the classical notion of dependency graphs suggested by Liu and Smolka.

5.1 Liu and Smolka Dependency Graphs

In the dependency graph framework introduced by Liu and Smolka [17], a dependency graph is represented as $G = (V, H)$ where V is a finite set of vertices and $H \subseteq V \times 2^V$ is the set of *hyperedges*. An *assignment* is a function $A : V \to \{0, 1\}$. A given assignment is a *fixed-point assignment* if $(A)(v) = \max_{(v,T)\in H} \min_{v'\in T} A(v')$ for all $v \in V$. In other words, A is a fixed-point assignment if for every hyperedge (v, T) where $T \subseteq V$ holds that if $A(v') = 1$ for every $v' \in T$ then also $A(v) = 1$. Liu and Smolka suggest both a global and a local algorithm [17] to compute the minimum fixed-point assignment for a given dependency graph.

We shall now argue how to instantiate their framework into abstract dependency graphs. Let (V, H) be a fixed dependency graph. We consider a NOR $\mathcal{D} = (\{0, 1\}, \leq, 0)$ where $0 < 1$ and construct an abstract dependency graph $G' = (V, E, \mathcal{D}, \mathcal{E})$. Here $E : V \to V^*$ is defined

$$E(v) = v_1 \cdots v_k \text{ s.t. } \{v_1, \ldots, v_k\} = \bigcup_{(v,T)\in H} T$$

such that $E(v)$ contains (in some fixed order) all vertices that appear on at least one hyperedge rooted with v. The labelling function $\mathcal{E}$ is now defined as expected

$$\mathcal{E}(v)(d_1, \ldots, d_k) = \max_{(v,T)\in H} \min_{v_i\in T} d_i$$

mimicking the computation in dependency graphs. For the efficiency of fixed-point computation in abstract dependency graphs it is important to provide an IGNORE function that includes as many vertices as possible. We shall use the following one

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } \exists (v, T) \in H. \forall u \in T. A(u) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

meaning that once there is a hyperedge with all the target vertices with value 1 (that propagates the value 1 to the root of the hyperedge), then the vertices of all other hyperedges can be ignored. This ignore function is, as we observed when running experiments, more efficient than this simpler one

$$\textsc{Ignore}(A, v) = \begin{cases} \{E(v)^i \mid 1 \leq i \leq |E(v)|\} & \text{if } A(v) = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

because it avoids the exploration of vertices that can be ignored before the root v is picked from the waiting list. Our encoding hence provides a generic and efficient way to model and solve problems described by Boolean equations [18] and dependency graphs [17].

5.2 Certain-Zero Dependency Graphs

Liu and Smolka's on-the-fly algorithm for dependency graphs significantly benefits from the fact that if there is a hyperedge with all target vertices having the value 1 then this hyperedge can propagate this value to the source of the hyperedge without the need to explore the remaining hyperedges. Moreover, the algorithm can early terminate should the root vertex v_0 get the value 1. On the other hand, if the final value of the root is 0 then the whole graph has to be explored and no early termination is possible. Recently, it has been noticed [19] that the speed of fixed-point computation by Liu and Smolka's algorithm can been considerably improved by considering also certain-zero value in the assignment that can, in certain situations, propagate from children vertices to their parents and once it reaches the root vertex, the algorithm can early terminate.

We shall demonstrate that this extension can be directly implemented in our generic framework, requiring only a minor modification of the abstract dependency graph. Let $G = (V, H)$ be a given dependency graph. We consider now a NOR $\mathcal{D} = (\{\bot, 0, 1\}, \sqsubseteq, \bot)$ where $\bot \sqsubset 0$ and $\bot \sqsubset 1$ but 0 and 1, the 'certain' values, are incomparable. We use the labelling function

$$\mathcal{E}(v)(d_1, \ldots, d_k) = \begin{cases} 1 & \text{if } \exists (v, T) \in H. \forall v_i \in T. d_i = 1 \\ 0 & \text{if } \forall (v, T) \in H. \exists v_i \in T. d_i = 0 \\ \bot & \text{otherwise} \end{cases}$$

so that it rephrases the method described in [19]. In order to achieve a competitive performance, we use the following ignore function.

$$\textsc{Ignore}(A,v) = \begin{cases} \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \exists (v,T) \in H. \forall u \in T. A(u) = 1 \\ \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \forall (v,T) \in H. \exists u \in T. A(u) = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

Our experiments presented in Sect. 6 show a clear advantage of the certain-zero algorithm over the classical one, as also demonstrated in [19].

5.3 Weighted Symbolic Dependency Graphs

In this section we show an application that instead of a finite NOR considers an ordering with infinitely many elements. This allows us to encode e.g. the model checking problem for weighted CTL logic as demonstrated in [3,20]. The main difference, compared to the dependency graphs in Sect. 5.1, is the addition of cover-edges and hyperedges with weight.

A *weighted symbolic dependency graph*, as introduced in [20], is a triple $G = (V, H, C)$, where V is a finite set of vertices, $H \subseteq V \times 2^{(\mathbb{N}^0 \times V)}$ is a finite set of hyperedges and $C \subseteq V \times \mathbb{N}^0 \times V$ a finite set of cover-edges. We assume the natural ordering relation $>$ on natural numbers such that $\infty > n$ for any $n \in \mathbb{N}^0$. An *assignment* $A : V \to \mathbb{N}^0 \cup \{\infty\}$ is a mapping from configurations to values. A *fixed-point assignment* is an assignment A such that

$$A(v) = \begin{cases} 0 & \text{if there is } (v,w,u) \in C \text{ such that } A(u) \le w \\ \min\limits_{(v,T)\in H} \big(\max\{A(u) + w \mid (w,u) \in T\}\big) & \text{otherwise} \end{cases}$$

where we assume that $\max \emptyset = 0$ and $\min \emptyset = \infty$. As before, we are interested in computing the value $A_{min}(v_0)$ for a given vertex v_0 where A_{min} is the minimum fixed-point assignment.

In order to instantiate weighted symbolic dependency graphs in our framework, we use the NOR $\mathcal{D} = (\mathbb{N}^0 \cup \{\infty\}, \ge, \infty)$ as introduced in Example 1 and define an abstract dependency graph $G' = (V, E, \mathcal{D}, \mathcal{E})$. We let $E : V \to V^*$ be defined as $E(v) = v_1 \cdots v_m c_1 \cdots c_n$ where $\{v_1, \ldots, v_m\} = \bigcup_{(v,T)\in H} \bigcup_{(w,v_i)\in T} \{v_i\}$ is the set (in some fixed order) of all vertices that are used in hyperedges and $\{c_1, \ldots, c_n\} = \bigcup_{(v,w,u)\in C} \{u\}$ is the set (in some fixed order) of all vertices connected to cover-edges. Finally, we define the labelling function $\mathcal{E}$ as

$$\mathcal{E}(v)(d_1, \ldots, d_m, e_1, \ldots, e_n) =$$

$$\begin{cases} 0 & \text{if } \exists (v,w,c_i) \in C.\ w \ge e_i \\ \min_{(v,T)\in H} \max_{(w,v_i)\in T} w + d_i & \text{otherwise.} \end{cases}$$

In our experiments, we consider the following ignore function.

$$\textsc{Ignore}(A,v) = \begin{cases} \{E(v)^i \mid 1 \le i \le |E(v)|\} & \text{if } \exists (v,w,u) \in C. A(u) \le w \\ \{E(v)^i \mid 1 \le i \le |E(v)|, A(E(v)^i) = 0\} & \text{otherwise} \end{cases}$$

```
struct Value {
  bool operator==(const Value&);
  bool operator!=(const Value&);
  bool operator<(const Value&);
};

struct VertexRef {
  bool operator==(const VertexRef&);
  bool operator<(const VertexRef&);
};

struct ADG {
  using Value = Value;
  using VertexRef = VertexRef;
  using EdgeTuple = vector<VertexRef>;
  static Value BOTTOM;
  VertexRef initialVertex();
  EdgeTuple getEdge(VertexRef& v);
  using VRA = typename algorithm:VertexRefAssignment<ADG>;
  Value compute(const VRA*, const VRA**, size_t n);
  void updateIgnored(const VRA*, const VRA**, size_t n, vector<bool>& ignore);
  bool ignoreSingle(const VRA* v, const VRA* u);
};
```

Fig. 2. The C++ interface

This shows that also the formalism of weighted symbolic dependency graphs can be modelled in our framework and the experimental evaluation documents that it outperforms the existing implementation.

6 Implementation and Experimental Evaluation

The algorithm is implemented in C++ and the signature of the user-provided interface in order to use the framework is shown in Fig. 2. The structure `ADG` is the main interface the algorithm uses. It assumes the definition of the type `Value` that represents the NOR, and the type `VertexRef` that represents a light-weight reference to a vertex and the bottom element. The type aliased as `VRA` contains both a `Value` and a `VertexRef` and represents the assignment of a vertex. The user must also provide the implementation of the functions: `initialVertex` that returns the root vertex v_0, `getEdge` that returns ordered successors for a given vertex, `compute` that computes $\mathcal{E}(v)$ for a given assignment of v and its successors, and `updateIgnored` that receives the assignment of a vertex and its successors and sets the ignore flags.

We instantiated this interface to three different applications as discussed in Sect. 5. The source code of the algorithm and its instantiations is available at https://launchpad.net/adg-tool/.

We shall now present a number of experiments showing that our generic implementation of abstract dependency graph algorithm is competitive with single-purpose implementations mentioned in the literature. The first two experiments (bisimulation checking for CCS processes and CTL model checking of Petri nets) were run on a Linux cluster with AMD Opteron 6376 processors running Ubuntu 14.04. We marked an experiment as OOT if it run for more than one hour and OOM if it used more than 16 GB of RAM. The final experiment for WCTL model checking required to be executed on a personal computer as the tool we compare to is written in JavaScript, so each problem instance was run on

Size	Time [s]			Memory [MB]		
	DG	ADG	Speedup	DG	ADG	Reduction
Lossy Alternating Bit Protocol – Bisimilar						
3	83.03	78.08	+6%	71	58	22%
4	2489.08	2375.10	+5%	995	810	23%
Lossy Alternating Bit Protocol — Nonbisimilar						
4	6.04	5.07	+19%	25	18	39%
5	4.10	5.08	−19%	69	61	13%
6	9.04	6.06	+49%	251	244	3%
Ring Based Leader-Election — Bisimilar						
8	21.09	18.06	+17%	31	23	35%
9	190.01	186.05	+2%	79	71	11%
10	2002.05	1978.04	+1%	298	233	28%
Ring Based Leader-Election — Nonbisimilar						
8	4.09	2.01	+103%	59	52	13%
9	16.02	15.07	+6%	185	174	6%
10	125.06	126.01	−1%	647	638	1%

Fig. 3. Weak bisimulation checking comparison

a Lenovo ThinkPad T450s laptop with an Intel Core i7-5600U CPU @ 2.60 GHz and 12 GB of memory.

6.1 Bisimulation Checking for CCS Processes

In our first experiment, we encode using ADG a number of weak bisimulation checking problems for the process algebra CCS. The encoding was described in [2] where the authors use classical Liu and Smolka's dependency graphs to solve the problems and they also provide a C++ implementation (referred to as DG in the tables). We compare the verification time needed to answer both positive and negative instances of the test cases described in [2].

Figure 3 shows the results where DG refers to the implementation from [2] and ADG is our implementation using abstract dependency graphs. It displays the verification time in seconds and peak memory consumptions in MB for both implementations as well as the relative improvement in percents. We can see that the performance of both algorithms is comparable, slightly in favour of our algorithm, sometimes showing up to 103% speedup like in the case of nonbisimilar processes in leader election of size 8. For nonbisimilar processes modelling alternating bit protocol of size 5 we observe a 19% slowdown caused by the different search strategies so that the counter-example to bisimilarity is found faster by the implementation from [2]. Memory-wise, the experiments are slightly in favour of our implementation.

We further evaluated the performance for weak simulation checking on task graph scheduling problems. We verified 180 task graphs from the Standard Task Graph Set as used in [2] where we check for the possibility to complete all tasks

Name	VerifyPN	ADG	Speedup
VerifyPN/ADG *Best 2*			
Diffusion2D-PT-D05N350:12	OOM	42.07	∞
Diffusion2D-PT-D05N350:01	332.70	0.01	+3326900%
VerifyPN/ADG *Middle 7*			
IOTPpurchase-PT-C05M04P03D02:08	4.15	2.13	+95%
Solitaire-PT-SqrNC5x5:09	340.31	180.47	+89%
Railroad-PT-010:08	155.34	83.92	+85%
IOTPpurchase-PT-C05M04P03D02:13	0.16	0.09	+78%
PolyORBLF-PT-S02J04T06:11	2.66	1.67	+59%
Diffusion2D-PT-D10N050:01	168.17	110.59	+52%
MAPK-PT-008:05	454.50	325.24	+40%
VerifyPN/ADG *Worst 2*			
ResAllocation-PT-R020C002:06	0.02	OOM	$-\infty$
MAPK-PT-008:06	0.01	OOM	$-\infty$

Fig. 4. Time comparison for CTL model checking (in seconds)

within a fixed number of steps. Both DG and ADG solved 35 task graphs using the classical Liu Smolka approach. However, once we allow for the certain-zero optimization in our approach (requiring to change only a few lines of code in the user-defined functions), we can solve 107 of the task graph scheduling problems.

6.2 CTL Model Checking of Petri Nets

In this experiment, we compare the performance of the tool TAPAAL [8] and its engine VerifyPN [21], version 2.1.0, on the Petri net models and CTL queries from the 2016 Model Checking Contest [22]. From the database of models and queries, we selected all those that do not contain logical negation in the CTL query (as they are not supported by the current implementation of abstract dependency graphs). This resulted in 267 model checking instances[1].

The results comparing the speed of model checking are shown in Fig. 4. The 267 model checking executions are ordered by the ratio of the verification time of VerifyPN vs. our implementation referred to as ADG. In the table we show the best two instances for our tool, the middle seven instances and the worst two instances. The results significantly vary on some instances as both algorithms are on-the-fly with early termination and depending on the search strategy the verification times can be largely different. Nevertheless, we can observe that on the average (middle) experiment IOTPpurchase-PT-C05M04P03D02:13, we are 78% faster than VerifyPN. However, we can also notice that in the two worst cases, our implementation runs out of memory.

[1] During the experiments we turned off the query preprocessing using linear programming as it solves a large number of queries by applying logical equivalences instead of performing the state-space search that we are interested in.

Name	VerifyPN	ADG	Reduction
VerifyPN/ADG *Best 2*			
Diffusion2D-PT-D05N350:12	OOM	4573	$+\infty$
Diffusion2D-PT-D05N350:01	9882	7	141171%
VerifyPN/ADG *Middle 7*			
PolyORBLF-PT-S02J04T06:13	17	23	−35%
ParamProductionCell-PT-0:02	1846	2556	−38%
ParamProductionCell-PT-0:07	1823	2528	−39%
ParamProductionCell-PT-4:13	1451	2064	−42%
SharedMemory-PT-000010:12	21	30	−43%
Angiogenesis-PT-15:04	51	74	−45%
Peterson-PT-3:03	1910	2792	−46%
VerifyPN/ADG *Worst 2*			
ParamProductionCell-PT-5:13	6	OOT	$-\infty$
ParamProductionCell-PT-0:10	6	OOT	$-\infty$

Fig. 5. Memory comparison for CTL model checking (in MB)

In Fig. 5 we present an analogous table for the peak memory consumption of the two algorithms. In the middle experiment ParamProductionCell-PT-4:13 we use 42% extra memory compared to VerifyPN. Hence we have a trade-off between the verification speed and memory consumption where our implementation is faster but consumes more memory. We believe that this is due to the use of the waiting list where we store directly vertices (allowing for a fast access to their assignment), compared to storing references to hyperedges in the VerifyPN implementation (saving the memory). Given the 16 GB memory limit we used in our experiments, this results in the fact that we were able to solve only 144 instances, compared to 218 answers provided by VerifyPN and we run 102 times out of memory while VerifyPN did only 45 times.

6.3 Weighted CTL Model Checking

Our last experiment compares the performance on the model checking of weighted CTL against weighted Kripke structures as used in the WKTool [3]. We implemented the weighted symbolic dependency graphs in our generic interface and run the experiments on the benchmark from [3]. The measurements for a few instances are presented in Fig. 6 and clearly show significant speedup in favour of our implementation. We remark that because WKTool is written in JavaScript, it was impossible to gather its peek memory consumption.

Instance	Time [s]			Satisfied?
	WKTool	ADG	Speedup	
Alternating Bit Protocol: EF$[\leq Y]$ *delivered* $= X$				
B=5 X=7 Y=35	7.10	0.83	+755%	yes
B=5 X=8 Y=40	4.17	1.05	+297%	yes
B=6 X=5 Y=30	7.58	1.44	+426%	yes
Alternating Bit Protocol: EF (*send0* && *deliver1*) $\|\|$ (*send1* && *deliver0*)				
B=5, M=7	7.09	1.39	+410%	no
B=5, M=8	4.64	1.60	+190%	no
B=6, M=5	7.75	2.37	+227%	no
Leader Election: EF leader > 1				
N=10	5.88	1.98	+197%	no
N=11	25.19	9.35	+169%	no
N=12	117.00	41.57	+181%	no
Leader Election: EF$[\leq X]$ *leader*				
N=11 X=11	24.36	2.47	+886%	yes
N=12 X=12	101.22	11.02	+819%	yes
N=11 X=10	25.42	9.00	+182%	no
Task Graphs: EF$[\leq 10]$ *done* $= 9$				
T=0	26.20	22.17	+18%	no
T=1	6.13	5.04	+22%	no
T=2	200.69	50.78	+295%	no

Fig. 6. Speed comparison for WCTL (B–buffer size, M–number of messages, N–number of processes, T–task graph number)

7 Conclusion

We defined a formal framework for minimum fixed-point computation on dependency graphs over an abstract domain of Noetherian orderings with the least element. This framework generalizes a number of variants of dependency graphs recently published in the literature. We suggested an efficient, on-the-fly algorithm for computing the minimum fixed-point assignment, including performance optimization features, and we proved the correctness of the algorithm.

On a number of examples, we demonstrated the applicability of our framework, showing that its performance is matching those of specialized algorithms already published in the literature. Last but not least, we provided an open source C++ library that allows the user to specify only a few domain-specific functions in order to employ the generic algorithm described in this paper. Experiential results show that we are competitive with e.g. the tool TAPAAL, winner of the 2018 Model Checking Contest in the CTL category [9], showing 78% faster performance on the median instance of the model checking problem, at the expense of 42% higher memory consumption.

In the future work, we shall apply our approach to other application domains (in particular probabilistic model checking), develop and test generic heuristic search strategies as well as provide a parallel/distributed implementation of our general algorithm (that is already available for some of its concrete instances [7, 23]) in order to further enhance the applicability of the framework.

Acknowledgments. The work was funded by the center IDEA4CPS, Innovation Fund Denmark center DiCyPS and ERC Advanced Grant LASSO. The last author is partially affiliated with FI MU in Brno.

References

1. Liu, X., Smolka, S.A.: Simple linear-time algorithms for minimal fixed points (extended abstract). In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 53–66. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055040
2. Dalsgaard, A.E., Enevoldsen, S., Larsen, K.G., Srba, J.: Distributed computation of fixed points on dependency graphs. In: Fränzle, M., Kapur, D., Zhan, N. (eds.) SETTA 2016. LNCS, vol. 9984, pp. 197–212. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47677-3_13
3. Jensen, J.F., Larsen, K.G., Srba, J., Oestergaard, L.K.: Efficient model checking of weighted CTL with upper-bound constraints. Int. J. Softw. Tools Technol. Transfer (STTT) **18**(4), 409–426 (2016)
4. Keiren, J.J.A.: Advanced reduction techniques for model checking. Ph.D thesis, Eindhoven University of Technology (2013)
5. Christoffersen, P., Hansen, M., Mariegaard, A., Ringsmose, J.T., Larsen, K.G., Mardare, R.: Parametric verification of weighted systems. In: André, É., Frehse, G. (eds.) SynCoP 2015, OASIcs, vol. 44, pp. 77–90. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015)
6. Larsen, K.G., Liu, X.: Equation solving using modal transition systems. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS 1990), Philadelphia, Pennsylvania, USA, 4–7 June 1990, pp. 108–117. IEEE Computer Society (1990)
7. Dalsgaard, A.E., et al.: Extended dependency graphs and efficient distributed fixed-point computation. In: van der Aalst, W., Best, E. (eds.) PETRI NETS 2017. LNCS, vol. 10258, pp. 139–158. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57861-3_10
8. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K.Y., Møller, M.H., Srba, J.: TAPAAL 2.0: integrated development environment for timed-arc Petri nets. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 492–497. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_36
9. Kordon, F., et al.: Complete Results for the 2018 Edition of the Model Checking Contest, June 2018. http://mcc.lip6.fr/2018/results.php
10. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_9

11. Mariegaard, A., Larsen, K.G.: Symbolic dependency graphs for PCTL$^{\geq}_{\leq}$ model-checking. In: Abate, A., Geeraerts, G. (eds.) FORMATS 2017. LNCS, vol. 10419, pp. 153–169. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65765-3_9
12. Larsen, K.G.: Efficient local correctness checking. In: von Bochmann, G., Probst, D.K. (eds.) CAV 1992. LNCS, vol. 663, pp. 30–43. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-56496-9_4
13. Andersen, H.R.: Model checking and boolean graphs. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 1–19. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55253-7_1
14. Mader, A.: Modal μ-calculus, model checking and Gauß elimination. In: Brinksma, E., Cleaveland, W.R., Larsen, K.G., Margaria, T., Steffen, B. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 72–88. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60630-0_4
15. Mateescu, R.: Efficient diagnostic generation for Boolean equation systems. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 251–265. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46419-0_18
16. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955)
17. Liu, X., Ramakrishnan, C.R., Smolka, S.A.: Fully local and efficient evaluation of alternating fixed points. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 5–19. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054161
18. Andersen, H.R.: Model checking and boolean graphs. Theoret. Comput. Sci. **126**(1), 3–30 (1994)
19. Dalsgaard, A.E., et al.: A distributed fixed-point algorithm for extended dependency graphs. Fundamenta Informaticae **161**(4), 351–381 (2018)
20. Jensen, J.F., Larsen, K.G., Srba, J., Oestergaard, L.K.: Local model checking of weighted CTL with upper-bound constraints. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 178–195. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39176-7_12
21. Jensen, J.F., Nielsen, T., Oestergaard, L.K., Srba, J.: TAPAAL and reachability analysis of P/T nets. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. LNCS, vol. 9930, pp. 307–318. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53401-4_16
22. Kordon, F., et al.: Complete Results for the 2016 Edition of the Model Checking Contest, June 2016. http://mcc.lip6.fr/2016/results.php
23. Joubert, C., Mateescu, R.: Distributed local resolution of Boolean equation systems. In: 13th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2005), 6–11 February 2005, Lugano, Switzerland, pp. 264–271. IEEE Computer Society (2005)

Tool Demos

nonreach – A Tool for Nonreachability Analysis

Florian Meßner and Christian Sternagel(✉)

University of Innsbruck, Innsbruck, Austria
{florian.g.messner,
christian.sternagel}@uibk.ac.at

Abstract. We introduce nonreach, an automated tool for nonreachability analysis that is intended as a drop-in addition to existing termination and confluence tools for term rewriting. Our preliminary experimental data suggests that nonreach can improve the performance of existing termination tools.

Keywords: Term rewriting · Nonreachability analysis · Narrowing · Termination · Confluence · Infeasibility

1 Introduction

Nonreachability analysis is an important part of automated tools like TTT2 [1] (for proving termination of rewrite systems) and ConCon [2] (for proving confluence of conditional rewrite systems). Many similar systems compete against each other in the annual termination (TermComp)[1] and confluence (CoCo)[2] competitions, both of which will run as part of TACAS's TOOLympics[3] in 2019.

Our intention for nonreach is to become a valuable component of all of the above mentioned tools by providing a fast and powerful back end for reachability analysis. This kind of analysis is illustrated by the following example.

Example 1. Suppose we have a simple program for multiplication represented by a term rewrite system (TRS, for short) consisting of the following rules:

$$\mathsf{add}(\mathsf{0}, y) \to y \qquad \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y))$$
$$\mathsf{mul}(\mathsf{0}, y) \to \mathsf{0} \qquad \mathsf{mul}(\mathsf{s}(x), y) \to \mathsf{add}(\mathsf{mul}(x, y), y)$$

For checking termination we have to make sure that there is no infinite sequence of recursive calls. One specific subproblem for doing so is to check whether it is

This work is supported by the Austrian Science Fund (FWF): project P27502.

[1] http://termination-portal.org/wiki/Termination_Competition.

[2] http://project-coco.uibk.ac.at/.

[3] https://tacas.info/toolympics.php.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 337–343, 2019.
https://doi.org/10.1007/978-3-030-17462-0_19

possible to reach $t = \mathsf{MUL}(\mathsf{s}(y), z)$[4] from $s = \mathsf{ADD}(\mathsf{mul}(w, x), x)$ for arbitrary instantiations of the variables w, x, y, and z. In other words, we have to check *nonreachability* of the target t from the source s.

In the remainder we will: comment on the role we intend for nonreach (Sect. 2), describe how nonreach is built and used (Sect. 3), give an overview of the techniques that went into nonreach (Sect. 4), and finally provide some experimental data (Sect. 5).

2 Role

When looking into implementations of current termination and confluence tools it soon becomes apparent that many tools use the same techniques for proving nonreachability. In light of this observation, one of our main goals for nonreach was to provide a dedicated stand-alone tool for nonreachability that can be reused for example in termination and confluence tools and can in principle replace many existing implementations.

In order to make such a reuse desirable for authors of other tools two things are important: (1) we have to provide a simple but efficient interface, and (2) we should support all existing techniques that can be implemented efficiently.[5]

At the time of writing, we already successfully use nonreach as back end in the termination tool $\mathsf{T_TT_2}$ [1]. To this end, we incorporated support for external nonreachability tools into $\mathsf{T_TT_2}$, with interaction purely via standard input and output. More specifically, while at the moment only YES/NO/MAYBE answers are required, the interface is general enough to support more detailed certificates corroborating such answers. The external tool is launched once per termination proof and supplied with those nonreachability problems that could not be handled by the existing techniques of $\mathsf{T_TT_2}$. Our next goal is to achieve the same for the confluence tool ConCon [2].

Furthermore, a new *infeasibility* category will be part of CoCo 2019.[6] Infeasibility is a concept from conditional term rewriting but can be seen as a variant of nonreachability [3]. Thus, we plan to enter the competition with nonreach.

Another potential application of nonreach is dead code detection or showing that some error can never occur.

3 Installation and Usage

Our tool nonreach is available from a public *bitbucket*[7] repository which can be obtained using the following command:

[4] We differentiate between recursive calls and normal argument evaluation by capitalization of function symbols.

[5] Efficiency is especially important under the consideration that, for example, termination tools may sometimes have to check thousands of nonreachability problems within a single termination proof.

[6] http://project-coco.uibk.ac.at/2019/categories/infeasibility.php.

[7] https://bitbucket.org/fmessner/nonreach.

```
git clone git@bitbucket.org:fmessner/nonreach.git
```

To compile and run nonreach, you need an up to date installation of *Haskell's stack*.[8] The source code is compiled by invoking `stack build` in the project directory containing the `stack.yaml` file. In order to install the executable in the local `bin` path (`~/.local/bin/` on Linux), run `stack install` instead.

Usage. The execution of nonreach is controlled by several command line flags. The only mandatory part is a rewrite system (with respect to which nonreachability should be checked). This may be passed either as literal string (flag `-d "..."`) or as file (flag `-f filename`). Either way, the input follows the formats for (conditional) rewrite systems that are used for TermComp and CoCo.

In addition to a rewrite system we may provide the nonreachability problems to be worked on (if we do not provide any problems, nonreach will wait indefinitely). For a single nonreachability problem the simple format `s -> t` is used, where s and t are terms and we are interested in nonreachability of the target t from the source s. Again, there are several ways to pass problems to nonreach:

- We can provide white-space-separated lists of problems either literally on the command line (flag `-P "..."`) or through a file (flag `-p filename`).
- Alternatively, a single infeasibility problem can be provided as part of the input rewrite system as specified by the new infeasibility category of CoCo 2019.
- Otherwise, nonreach waits for individual problems on standard input.

For each given problem nonreach produces one line of output: In its default mode the output is `NO` whenever nonreachability can be established and either `MAYBE` or `TIMEOUT`, otherwise. When given an infeasibility problem, the output is `YES` if the problem is infeasible and either `MAYBE` or `TIMEOUT`, otherwise.

Further flags may be used to specify a timeout (in microseconds; flag `-t`) or to give finer control over the individual techniques that are implemented in nonreach (we will mention those in Sect. 4).

It is high time for an example. Let us check Example 1 using nonreach.

Example 2. Assuming that the TRS of Example 1 is in a file `mul.trs` we can have the following interaction (where we indicate user input by a preceding `>`):

```
nonreach -f mul.trs
> ADD(mul(w,x),x) -> MUL(s(y),z)
NO
```

4 Design and Techniques

In this section we give a short overview of the general design decisions and specific nonreachability techniques that went into nonreach.

[8] https://docs.haskellstack.org/en/stable/README.

Design. Efficiency was at the heart of our concern. On the one hand, from a user-interface perspective, this was the reason to provide the possibility that a single invocation of nonreach for a fixed TRS can work on arbitrarily many reachability problems. On the other hand, this lead us to mostly concentrate on techniques that are known to be fast in practice. The selection of techniques we present below (with the exception of narrowing) satisfies this criterion.

Techniques. Roughly speaking, nonreach uses two different kinds of techniques: (1) *transformations* that result in disjunctions or conjunctions of easier nonreachability problems, and (2) actual *nonreachability checks*. We use the notation $s \twoheadrightarrow t$ for a nonreachability problem with source s and target t.

Reachability checks. The first check we recapitulate is implemented by most termination tools and based on the idea of computing the topmost part of a term that does not change under rewriting (its *cap*) [4]. If the cap of the source s does not unify with the target t, then there are no substitutions σ and τ such that $s\sigma \to^* t\tau$. There are different algorithms to underapproximate such caps. We use etcap, developed by Thiemann together with the second author [5], due to its linear time complexity (by reducing unification to *ground context matching*) but nevertheless simple implementation. With etcap subterms are matched bottom-up with left-hand sides of rules. In case of a match, the corresponding subterm is potentially rewritable and thus replaced by a *hole*, written $\Box$, representing a fresh variable. We illustrate etcap on the first two (addition) rules of Example 1.

Example 3. We have $\mathsf{etcap}(\mathsf{s}(\mathsf{0})) = \mathsf{s}(\mathsf{0})$ and $\mathsf{etcap}(\mathsf{s}(\mathsf{add}(\mathsf{s}(z), \mathsf{s}(\mathsf{0})))) = \mathsf{s}(\Box)$, since the subterm headed by add matches the second addition rule. Using etcap, we obtain the following nonreachability check

$$\mathsf{reach}_{\mathsf{etcap}}(s \twoheadrightarrow t) = \begin{cases} \mathtt{MAYBE} & \text{if } \mathsf{etcap}(s) \sim t \\ \mathtt{NO} & \text{otherwise} \end{cases}$$

where $\sim$ denotes unifiability of terms.

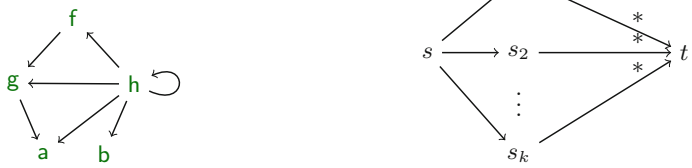
The second reachability check implemented in nonreach [3,6] is based on the so called *symbol transition graph* (SG for short) of a TRS. Here, the basic idea is to build a graph that encodes the dependencies between root symbols induced by the rules of a TRS. This is illustrated by the following example:

Example 4. Given the TRS $\mathcal{R}$ consisting of the four rules

$$\mathsf{f}(x, x) \to \mathsf{g}(x) \qquad \mathsf{g}(x) \to \mathsf{a} \qquad \mathsf{h}(\mathsf{a}) \to \mathsf{b} \qquad \mathsf{h}(x) \to x$$

we generate the corresponding SG shown in Fig. 1(a) on page 5.

For each rule we generate an edge from the node representing the root symbol of the left-hand side to the node representing the root symbol of the right-hand side. Since in the last rule, the right-hand side is a variable (which can in principle



(a) SG of TRS R from Example 4. (b) Decomposition by narrowing.

Fig. 1. A symbol transition graph and the idea of narrowing.

be turned into an arbitrary term by applying a substitution), we have to add edges from h to all nodes (including h itself).

From the graph it is apparent that $\mathsf{f}(x, y)$ is not reachable from $\mathsf{g}(z)$, no matter the substitutions for x, y and z, since there is no path from g to f.

We obtain the following SG based nonreachability check:

$$\mathsf{reach}_{\mathsf{stg}}(s \twoheadrightarrow t) = \begin{cases} \texttt{MAYBE} & \text{if there is a path from } \mathsf{root}(s) \text{ to } \mathsf{root}(t) \text{ in the graph} \\ \texttt{NO} & \text{otherwise} \end{cases}$$

Which reachability checks are applied by nonreach can be controlled by its `-s` flag, which expects a list of checks (in Haskell syntax). Currently the two checks `TCAP` and `STG` are supported.

Transformations. We call the first of our transformations *(problem) decomposition*. This technique relies on root-reachability checks to decompose a problem into a *conjunction* of strictly smaller subproblems. Thus, we are done if any of these subproblems is nonreachable.

Decomposition of a problem $s \twoheadrightarrow t$ only works if source and target are of the form $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$, respectively. Now the question is whether t can be reached from s involving at least one root-rewrite step. If this is not possible, we know that for reachability from s to t, we need all t_i to be reachable from the corresponding s_i.

For the purpose of checking root-nonreachability, we adapt the two reachability checks from above:

$$\mathsf{rootreach}_{\mathsf{etcap}}(s, t) = \begin{cases} \texttt{True} & \text{if there is a rule } \ell \to r \text{ such that } \mathsf{etcap}(s) \sim \ell \\ \texttt{False} & \text{otherwise} \end{cases}$$

$$\mathsf{rootreach}_{\mathsf{stg}}(s, t) = \begin{cases} \texttt{True} & \text{if there is a non-empty path from } \mathsf{root}(s) \\ & \text{to } \mathsf{root}(t) \text{ in the SG} \\ \texttt{False} & \text{otherwise} \end{cases}$$

If at least one of these checks returns `False`, we can decompose the initial problem $s \twoheadrightarrow t$ into a conjunction of problems $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$.

The second transformation is based on *narrowing*. Without going into too much technical detail let us start from the following consideration. Given a reachability problem $s \twoheadrightarrow t$, assume that t is reachable from s. Then either the corresponding rewrite sequence is empty (in which case s and t are unifiable) or there is at least one initial rewrite step. Narrowing is the tool that allows us to capture all possible first steps (one for each rule $l \to r$ and each subterm of s that unifies with l), of the form $s \to s_i \to^* t$. This idea is captured in Fig. 1(b).

Now, decomposition (which is always applicable and thus has to be handled with care), transforms a given reachability problem $s \twoheadrightarrow t$ into a disjunction of new problems (that is, this time we have to show `NO` for all of these problems in order to conclude `NO` for the initial one) $s \stackrel{?}{\sim} t$ or $s_1 \twoheadrightarrow t$ or $\ldots$ or $s_k \twoheadrightarrow t$, where the first one is a unifiability problem and the remaining ones are again reachability problems.

The maximal number of narrowing applications is specified by the `-l` flag.

5 Experiments

In order to obtain test data for nonreach we started from the *Termination Problem Data Base* (TPDB, for short),[9] the data base of problems that are used also in TermComp. Then we used a patched version of the termination tool TTT2 to obtain roughly 20,000 non-trivial reachability problems with respect to 730 TRSs. These are available from a public *bitbucket* repository[10] alongside a small script, that runs nonreach on all the problems and displays the overall results. All these problems could, at the time of creating the dataset, not be solved with the reachability checks used in the competition strategy of TTT2. The current version[11] of nonreach in its default configuration can prove nonreachability of 369 problems of this set. While this does not seem much, we strongly suspect that the majority of problems in our set are actually reachable.

References

1. Korp, M., Sternagel, C., Zankl, H., Middeldorp, A.: Tyrolean termination tool 2. In: Treinen, R. (ed.) RTA 2009. LNCS, vol. 5595, pp. 295–304. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02348-4_21
2. Sternagel, T., Middeldorp, A.: Conditional confluence (system description). In: Dowek, G. (ed.) RTA 2014. LNCS, vol. 8560, pp. 456–465. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08918-8_31
3. Sternagel, C., Yamada, A.: Reachability analysis for termination and confluence of rewriting. In: Vojnar, T., Zhang, L. (eds.) TACAS 2019. LNCS, vol. 11427, pp. 262–278 (2019)
4. Giesl, J., Thiemann, R., Schneider-Kamp, P.: Proving and disproving termination of higher-order functions. In: Gramlich, B. (ed.) FroCoS 2005. LNCS (LNAI), vol. 3717, pp. 216–231. Springer, Heidelberg (2005). https://doi.org/10.1007/11559306_12

[9] http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB.
[10] https://bitbucket.org/fmessner/nonreach-testdata.
[11] https://bitbucket.org/fmessner/nonreach/src/77945a5/?at=master.

5. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 452–468. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_31
6. Yamada, A.: Reachability for termination. In: 4th AJSW (2016). http://cl-informatik.uibk.ac.at/users/ayamada/AJSW2016-slides.pdf

The Quantitative Verification Benchmark Set

Arnd Hartmanns[1(✉)], Michaela Klauck[2], David Parker[3], Tim Quatmann[4], and Enno Ruijters[1]

[1] University of Twente, Enschede, The Netherlands
a.hartmanns@utwente.nl
[2] Saarland University, Saarbrücken, Germany
[3] University of Birmingham, Birmingham, UK
[4] RWTH Aachen, Aachen, Germany

Abstract. We present an extensive collection of quantitative models to facilitate the development, comparison, and benchmarking of new verification algorithms and tools. All models have a formal semantics in terms of extensions of Markov chains, are provided in the JANI format, and are documented by a comprehensive set of metadata. The collection is highly diverse: it includes established probabilistic verification and planning benchmarks, industrial case studies, models of biological systems, dynamic fault trees, and Petri net examples, all originally specified in a variety of modelling languages. It archives detailed tool performance data for each model, enabling immediate comparisons between tools and among tool versions over time. The collection is easy to access via a client-side web application at qcomp.org with powerful search and visualisation features. It can be extended via a Git-based submission process, and is openly accessible according to the terms of the CC-BY license.

1 Introduction

Quantitative verification is the analysis of formal models and requirements that capture probabilistic behaviour, hard and soft real-time aspects, or complex continuous dynamics. Its applications include probabilistic programs, safety-critical and fault-tolerant systems, biological processes, queueing systems, and planning in uncertain environments. Quantitative verification tools can, for example, compute the worst-case probability of failure within a time bound, the minimal expected cost to achieve a goal, or a Pareto-optimal control strategy balancing energy consumption versus the probability of unsafe behaviour. Two prominent such tools are PRISM [15] for probabilistic and UPPAAL [17] for real-time systems.

Over the past decade, various improvements and extensions have been made to quantitative model checking algorithms, with different approaches implemented in an increasing number of tools, e.g. [7,8,11,13,18]. Researchers, tool

Authors are listed alphabetically. This work was supported by DFG grant 389792660 (part of CRC 248), ERC Advanced Grants 695614 (POWVER) and 781914 (FRAPPANT), NWO and BetterBe grant 628.010.006, and NWO VENI grant 639.021.754.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 344–350, 2019.
https://doi.org/10.1007/978-3-030-17462-0_20

developers, non-academic users, and reviewers can all greatly benefit from a common set of realistic and challenging examples that new algorithms and tools are consistently benchmarked and compared on and that may indicate the practicality of a new method or tool. Such sets, and the associated push to standardised semantics, formats, and interfaces, have proven their usefulness in other areas such as software verification [4] and SMT solving [3].

In quantitative verification, the Prism Benchmark Suite (PBS) [16] has served this role for the past seven years. It provides 24 distinct examples in the Prism language covering discrete- and continuous time Markov chains (DTMC and CTMC), discrete-time Markov decision processes (MDP), and probabilistic timed automata (PTA). To date, it has been used in over 60 scientific papers. Yet several developments over the past seven years are not adequately reflected or supported by the PBS. New tools (1) support other modelling languages and semantics (in particular, several tools have converged on the Jani model exchange format [6]), and (2) exploit higher-level formalisms like Petri nets or fault trees. In addition, (3) today's quantitative verification tools employ a wide range of techniques, whereas the majority of models in the PBS work best with Prism's original BDD-based approach. Furthermore, (4) probabilistic verification and planning have been connected (e.g. [14]), and (5) MDP have gained in prominence through recent breakthroughs in AI and learning.

We present the Quantitative Verification Benchmark Set (QVBS): a new and growing collection of currently 72 models (Sect. 2) in the Jani format, documented by comprehensive metadata. It includes all models from the PBS plus a variety of new examples originally specified in significantly different modelling languages. It also covers decision processes in continuous stochastic time via Markov automata (MA [9]). The QVBS aggregates performance results obtained by different tools on its models (Sect. 3). All data is accessible via a client-side web application with powerful search and visualisation capabilities (Sect. 4).

2 A Collection of Quantitative Models

The Quantitative Verification Benchmark Set is characterised by commonality and diversity. All models are available in the Jani model exchange format [6], and they all have a well-defined formal semantics in terms of five related automata-based probabilistic models based on Markov chains. At the same time, the models of the QVBS originate from a number of different application domains, were specified in six modelling languages (with the original models plus information on the Jani conversion process being preserved in the QVBS), and pose different challenges including state space explosion, numeric difficulties, and rare events.

Syntax and semantics. The QVBS accepts any interesting model with a Jani translation to the DTMC, CTMC, MDP, MA, and PTA model types. Its current models were originally specified in Galileo for fault trees [20], GreatSPN [2] for Petri nets, the Modest language [5], PGCL for probabilistic programs [10], PPDDL for planning domains [21], and the Prism language [15]. By also storing

Table 1. Sources and domains of models

		source			application domain					
	all	PBS	IPPC	TA	com	rda	dpe	pso	bio	sec
all	72	24	10	7	12	9	17	16	6	5
DTMC	9	7			2	3	1			2
CTMC	13	7					4	1	6	
MDP	25	5	10		5	5		13		
MA	18			7		1	12	2		1
PTA	7	5			5					2

Table 2. Properties and valuations

	properties						parameter valuations				
	all	P	Pb	E	Eb	S	all	10^4	10^6	10^7	$>10^7$
all	229	90	57	52	12	18	589	135	127	94	28
DTMC	20	10	1	9			91	40	23	14	14
CTMC	49	6	22	4	11	6	161	43	52	28	5
MDP	61	40	3	17	1		82	31	24	21	6
MA	61	14	18	17		12	218	7	28	26	3
PTA	38	20	13	5			37	14		5	

the original model, structural information (such as in Petri nets or fault trees) that is lost by a conversion to an automata-based model is preserved for tools that can exploit it. We plan to broaden the scope to e.g. stochastic timed automata [5] or stochastic hybrid systems [1] in coordination with interested tool authors.

Sources and application domains. 41 of the QVBS's current 72 models stem from existing smaller and more specialised collections: 24 from the PRISM Benchmark Suite (PBS) [16], 10 from the probabilistic/uncertainty tracks of the 2006 and 2008 International Planning Competitions (IPPC) [21], and 7 repairable dynamic fault trees from the Twente Arberretum (TA) [19]. 65 of the models can be categorised as representing systems from six broad application domains: models of communication protocols (com), of more abstract randomised and distributed algorithms (rda), for dependability and performance evaluation (dpe), of planning, scheduling and operations management scenarios (pso), of biological processes (bio), and of mechanisms for security and privacy (sec). We summarise the sources and application domains of the QVBS models in Table 1.

Metadata. Alongside each model, in original and JANI format, we store a comprehensive set of structured JSON metadata to facilitate browsing and data mining the benchmark set. This includes basic information such as a description of the model, its version history, and references to the original source and relevant literature. Almost all models are parameterised such that the difficulty of analysing the model can be varied: some parameters influence the size of the state spaces, others may be time bounds used in properties, etc. The metadata documents all parameters and the ranges of admissible values. It includes sets of "proposed" parameter valuations with corresponding state space sizes and reference results. Each model contains a set of properties to be analysed; they are categorised into probabilistic unbounded and bounded reachability (P and Pb), unbounded and bounded expected rewards (E and Eb), and steady-state queries (S). Table 2 summarises the number of properties of each type (left), and the number of suggested parameter valuations (right) per resulting state space size (if available), where e.g. column "10^6" lists the numbers of valuations yielding 10^4 to 10^6 states.

3 An Archive of Results

The Quantitative Verification Benchmark Set collects not only models, but also *results*: the values of the properties that have been checked and performance data on runtime and memory usage. For every model, we archive results obtained with different tools/tool versions and settings on different hardware in a structured JSON format. The aim is to collect a "big dataset" of performance information that can be mined for patterns over tools, models, and time. It also gives developers of new tools and algorithms a quick indication of the relative performance of their implementation, saving the often cumbersome process of installing and running many third-party tools locally. Developers of existing tools may profit from an archive of the performance of their own tool, helping to highlight performance improvements—or pinpoint regressions—over time. The QVBS includes a graphical interface to aggregate and visualise this data (see Sect. 4 below).

4 Accessing the Benchmark Set

The models and results data of the Quantitative Verification Benchmark Set are managed in a Git repository at github.com/ahartmanns/qcomp. A user-friendly interface is provided at qcomp.org/benchmarks via a web application that dynamically loads the JSON data and presents it in two views:

The model browser presents a list of all models with key metadata. The list can be refined by a full-text search over the models' names, descriptions and notes, and by filters for model type, original modelling language, property types, and state space size. For example, a user could request the list of all MODEST MDP models with an expected-reward property and at least ten million states. Every model can be opened in a detail view that links to the JANI and original files, shows all metadata including parameters, proposed valuations, and properties with reference results, and provides access to all archived results. Figure 1 shows the model browser filtered to GREATSPN models that include a bounded probabilistic reachability property. The flexible-manufacturing model is open in detail view.

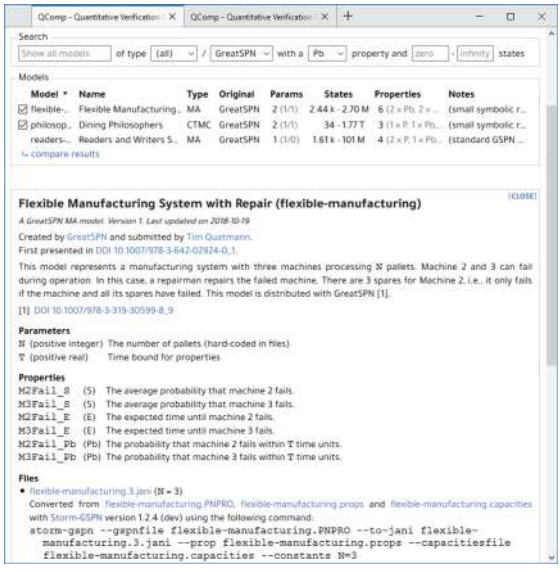


Fig. 1. The model browser and detail view

The results browser is accessed by selecting one or more models in the model browser and opening the "compare results" link. It provides a flexible, summarising view of the performance data collected from all archived results for the selected models. The data can be filtered to include select properties or parameter valuations only. It is visualised as a table or different types of charts, including bar charts and scatter plots. Figure 2 shows the result browser for the beb and breakdown-queues models, comparing the performance of MCSTA [13] with default settings to STORM [8] in its slower "exact" mode. The performance data can optionally be normalised by the benchmark scores of the CPU used to somewhat improve comparability, although this still disregards many other important factors (like memory bandwidth and storage latency), of course.

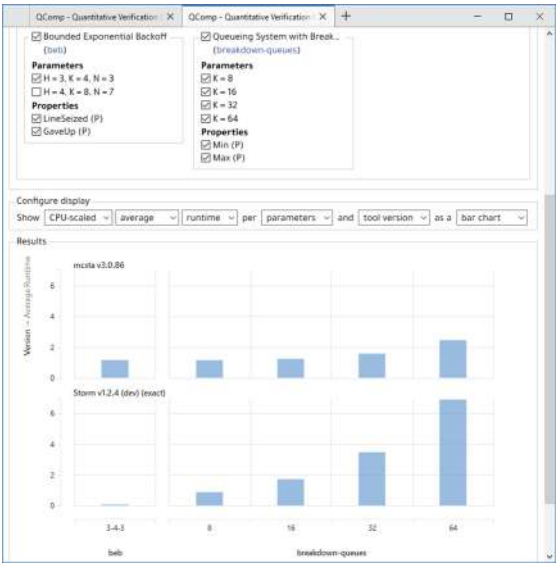


Fig. 2. The results browser showing a bar chart

The web application is entirely client-side: all data is loaded into the user's browser as needed. All aggregation, filtering, and visualisation is implemented in Javascript. The application thus has no requirements on the server side. It is part of the Git repository and can be downloaded and opened offline by anyone.

5 Conclusion

Building upon the successful foundation of the PRISM Benchmark Suite, the new Quantitative Verification Benchmark Set not only expands the number and diversity of easily accessible benchmarks, but also professionalises the collection and provision of benchmark data through its JSON-based formats for metadata and results. We expect its associated web application to become a valuable tool for researchers, tool authors, and users alike. The QVBS is also an *open* dataset: all content is available under the CC-BY license, and new content—new models, updates, and results—can be contributed via a well-defined Git-based process. The Quantitative Verification Benchmark Set is the sole source of models for QCOMP 2019 [12], the first friendly competition of quantitative verification tools.

Acknowledgments. The authors thank Paul Gainer (University of Liverpool), Sebastian Junges (RWTH Aachen), Joachim Klein (Technische Universität Dresden), Matthias Volk (RWTH Aachen), and Zhen Zhang (Utah State University) for submitting models to the QVBS, Gethin Norman (University of Glasgow) for his contributions to the Prism Benchmark Suite, and Marcel Steinmetz (Saarland University) for translating the IPPC benchmarks.

References

1. Abate, A., Blom, H., Cauchi, N., Haesaert, S., Hartmanns, A., Lesser, K., Oishi, M., Sivaramakrishnan, V., Soudjani, S., Vasile, C.I., Vinod, A.P.: ARCH-COMP18 category report: stochastic modelling. In: ARCH Workshop at ADHS. EPiC Series in Computing, vol. 54, pp. 71–103. EasyChair (2018)
2. Amparore, E.G., Balbo, G., Beccuti, M., Donatelli, S., Franceschinis, G.: 30 years of GreatSPN. In: Fiondella, L., Puliafito, A. (eds.) Principles of Performance and Reliability Modeling and Evaluation. SSRE, pp. 227–254. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30599-8_9
3. Barrett, C., Fontaine, P., Tinelli, C.: SMT-LIB benchmarks. http://smtlib.cs.uiowa.edu/benchmarks.shtml
4. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
5. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: MoDeST: a compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. **32**(10), 812–830 (2006)
6. Budde, C.E., Dehnert, C., Hahn, E.M., Hartmanns, A., Junges, S., Turrini, A.: JANI: quantitative model and tool interaction. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 151–168. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_9
7. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: Uppaal Stratego. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 206–211. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_16
8. Dehnert, C., Junges, S., Katoen, J.-P., Volk, M.: A Storm is coming: a modern probabilistic model checker. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 592–600. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_31
9. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351. IEEE Computer Society (2010)
10. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: FOSE, pp. 167–181. ACM (2014)
11. Hahn, E.M., Li, Y., Schewe, S., Turrini, A., Zhang, L.: iscasMc: a web-based probabilistic model checker. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 312–317. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-06410-9_22
12. Hartmanns, A., Hensel, C., Klauck, M., Klein, J., Kretínský, J., Parker, D., Quatmann, T., Ruijters, E., Steinmetz, M.: The 2019 comparison of tools for the analysis of quantitative formal models. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 69–92. Springer, Cham (2019)

13. Hartmanns, A., Hermanns, H.: The Modest Toolset: an integrated environment for quantitative modelling and verification. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 593–598. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_51
14. Klauck, M., Steinmetz, M., Hoffmann, J., Hermanns, H.: Compiling probabilistic model checking into probabilistic planning. In: ICAPS. AAAI Press (2018)
15. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
16. Kwiatkowska, M.Z., Norman, G., Parker, D.: The PRISM benchmark suite. In: QEST, pp. 203–204. IEEE Computer Society (2012)
17. Larsen, K.G., Lorber, F., Nielsen, B.: 20 years of UPPAAL enabled industrial model-based validation and beyond. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 212–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_18
18. Legay, A., Sedwards, S., Traonouez, L.-M.: Plasma Lab: a modular statistical model checking platform. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9952, pp. 77–93. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47166-2_6
19. Ruijters, E., et al.: The Twente Arberretum. https://dftbenchmarks.utwente.nl/
20. Sullivan, K.J., Dugan, J.B., Coppit, D.: The Galileo fault tree analysis tool. In: FTCS-29, pp. 232–235. IEEE Computer Society (1999)
21. Younes, H.L.S., Littman, M.L., Weissman, D., Asmuth, J.: The first probabilistic track of the International Planning Competition. J. Artif. Intell. Res. **24**, 851–887 (2005)

ILAng: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions

Bo-Yuan Huang(✉), Hongce Zhang, Aarti Gupta, and Sharad Malik

Princeton University, Princeton, USA
{byhuang,hongcez,aartig,sharad}@princeton.edu

Abstract. We present ILAng, a platform for modeling and verification of systems-on-chip (SoCs) using Instruction-Level Abstractions (ILA). The ILA formal model targeting the hardware-software interface enables a clean separation of concerns between software and hardware through a unified model for heterogeneous processors and accelerators. Top-down it provides a specification for functional verification of hardware, and bottom-up it provides an abstraction for software/hardware co-verification. ILAng provides a programming interface for (i) constructing ILA models (ii) synthesizing ILA models from templates using program synthesis techniques (iii) verifying properties on ILA models (iv) behavioral equivalence checking between different ILA models, and between an ILA specification and an implementation. It also provides for translating models and properties into various languages (e.g., Verilog and SMT LIB2) for different verification settings and use of third-party verification tools. This paper demonstrates selected capabilities of the platform through case studies. Data or code related to this paper is available at: [9].

1 Introduction

Modern computing platforms are increasingly heterogeneous, having both programmable processors and application-specific accelerators. These accelerator-rich platforms pose two distinct verification challenges. The first challenge is constructing meaningful specifications for accelerators that can be used to verify the implementation. Higher-level executable models used in early design stages are not suitable for this task. The second challenge is to reason about hardware-software interactions from the viewpoint of software. The traditional approach in system-level/hardware modeling using detailed models, e.g., Register-Transfer Level (RTL) descriptions, is not amenable to scalable formal analysis.

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA. This research is also funded in part by NSF award number 1628926, XPS: FULL: Hardware Software Abstractions: Addressing Specification and Verification Gaps in Accelerator-Oriented Parallelism, and the DARPA POSH Program Project: Upscale: Scaling up formal tools for POSH Open Source Hardware.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 351–357, 2019.
https://doi.org/10.1007/978-3-030-17462-0_21

The Instruction-Level Abstraction (ILA) has been proposed to address these challenges [4]. The ILA model is a uniform formal abstraction for processors and accelerators that captures their software-visible functionality as a set of instructions. It facilitates behavioral equivalence checking of an implementation against its ILA specification. This, in turn, supports accelerator upgrades using the notion of ILA compatibility similar to that of ISA compatibility for processors [4]. It also enables firmware/hardware co-verification [3]. Further, it enables reasoning about memory consistency models for system-wide properties [8].

In this paper, we present ILAng, a platform for Systems-on-chip (SoC) verification where ILAs are used as the formal model for processors and accelerators. ILAng provides for (i) ILA modeling and synthesis, (ii) ILA model verification, and (iii) behavioral equivalence checking between ILA models, and between an ILA specification and an implementation. The tool is open source and available online on https://github.com/Bo-Yuan-Huang/ILAng.

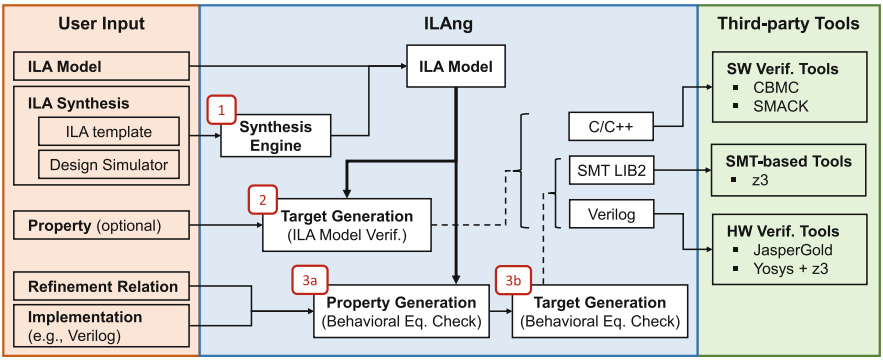


Fig. 1. ILAng work flow

2 The ILAng Platform

Figure 1 illustrates the modeling and verification capabilities of ILAng.

1. It allows constructing ILA formal models using a programming interface. It also allows semi-automated ILA synthesis using program synthesis techniques [5] and a template language [6].
2. It provides a set of utilities for ILA model verification, such as SMT-based transition unrolling and bounded model checking. Further, ILAng is capable of translating ILA formal models into various languages for verification, including Verilog, C, C++, and SMT LIB2, targeting other off-the-shelf verification tools and platforms.
3. It supports behavioral equivalence checking between ILA models and between an ILA specification and an implementation based on the standard commutative diagram approach [1].

2.1 Background

ILA. The ILA is a formal, uniform, modular, and hierarchical model for modeling both programmable processors and accelerators [4].

An ILA model is **modular** (state updates defined per instruction) and can be viewed as a generalization of the processor ISA in the heterogeneous context, where the commands on an accelerator's interface are defined as instructions. Generally, these appear as memory-mapped IO (MMIO) instructions in programs. The MMIO instruction lacks the actual semantics of what the instruction is doing in the accelerator; the ILA instruction captures exactly that.

It is an operational model that captures updates to *program-visible states* (i.e., states that are accessible or observable by programs). For processors, these include the architectural registers, flag bits, data, and program memory. An `ADD` instruction ($R1 = R2 + R3$), for example, of a processor defines the update to the $R1$ program-visible register. For accelerators, the program visible state includes memory-mapped registers, internal buffers, output ports to on-chip interconnect, etc. An ILA instruction for a crypto-accelerator could define the setting of an encryption key or even the encryption of an entire block using this key.

An ILA model is **hierarchical**, where an instruction at a high level can be expressed as a program of *child-instructions* (like micro-instructions). For example, the `START_ENCRYPT` instruction of a crypto-accelerator can be described as a program of reading data, encrypting it, and writing the result.

2.2 Constructing ILAs

ILAng provides a programming interface to define ILA models. For each component, the program-visible (aka architectural) states are specified. For each instruction, the state updates are specified independently as transition relations. Currently, ILAng supports state variables of type Boolean, bit-vector, and array. Uninterpreted functions are also supported for modeling complex operations.

Synthesis Capability. To alleviate the manual effort in constructing ILAs, ILAng provides a synthesis engine for synthesizing ILAs from a partial description called a template [6] using program synthesis techniques [5]. This is shown as Block 1 in Fig. 1. The synthesis algorithm requires two user inputs: an ILA template, and a design simulator/emulator. A template is a partially defined ILA model that specifies the program-visible states, the set of instructions, and also the *possible* operations for each instruction. The simulator can be a software/hardware prototype and is used as an oracle during ILA synthesis. The synthesis algorithm fills in the missing parts in the set of instructions.

2.3 Verification Using ILAs

The ILA formal model is a modular (per instruction) transition system, enabling the use of verification techniques such as model checking. We now discuss a selected list of verification capabilities provided by ILAng.

ILA Model Verification. As shown in Block 2 in Fig. 1, ILAng supports verification of user-provided safety properties on ILAs. It generates verification targets (including the design and property assertions) into different languages, as discussed in Sect. 2.4.

Behavioral Equivalence Checking. The modularity and hierarchy in the ILA models simplify behavioral equivalence checking through decomposition. Based on the standard commutative diagram approach [1], behavioral equivalence checking in ILAng supports two main settings: (i) ILA vs. ILA, and (ii) ILA vs. finite state machine (FSM) model (e.g., RTL implementation). As shown in Blocks 3*a* and 3*b* in Fig. 1, ILAng takes as input the two models (two ILA models, or an ILA model and an implementation model) and the user-provided refinement relation. The refinement relation specifies:

1. how to apply an instruction (e.g., instruction decode),
2. how to flush the state machine if required (e.g., stalling), and
3. how to identify if an instruction has completed (e.g., commit point).

The refinement map is then used with the two models to generate the property assertions using the standard commutative diagram approach [1]. The verification targets (the design and property assertions) are then generated in Verilog or SMT LIB2 for further reasoning, as described in Sect. 2.4.

SMT-Based Verification Utilities

Unrolling. Given an ILA, ILAng provides utilities to unroll the transition relation up to a finite bound, with options for different encodings and simplifications. Users can unroll a sequence of given instructions with a fixed instruction ordering. They can also unroll all possible instructions as a monolithic state machine.

Bounded Model Checking (BMC). With the unrolling utility, ILAng supports BMC of safety properties using an SMT solver. Users can specify initial conditions, invariants, and the properties. They can use a fixed bound for BMC or use on-the-fly BMC that iteratively increases the bound.

2.4 Generating Verification Targets

To utilize off-the-shelf verification tools, ILAng can export ILA models into different languages, including Verilog, C, C++, and SMT-LIB2.

Verilog. Many hardware implementations use RTL descriptions in Verilog. To support ILA vs. FSM equivalence checking, ILAng supports exporting ILA to Verilog. This also allows the use of third-party verification tools, since most such tools support Verilog. The memory module in the exported ILA can be configured as internal registers or external memory modules for different requirements.

C/C++. Given an ILA model, ILAng can generate a hardware simulator (in C or C++) for use in system/software development. This simulator can be verified against an implementation to check that it is a reliable executable model.

SMT LIB2. The ILAng platform is integrated with the SMT solver z3 [2]. It can generate SMT formulas for the transition relation and other verification conditions using the utilities described in Sect. 2.3.

3 Tutorial Case Studies

We demonstrate the applicability of ILAng through two case studies discussed in this section.[1] Table 1 provides information for each case study, including implementation size, ILA (template) size, synthesis time, and verification time (Ubuntu 18.04 VM on Intel i5-8300H with 2 GB memory). Note that these case studies are for demonstration, ILAng is capable of handling designs with significant complexity, as discussed and referenced in Sect. 4.

Table 1. Experimental Results

	Design Statistics		**ILA**			**Verif.**
	Reference	Ref. Size	# of Insts. (parent/child)	ILA Size	Synth. Time (s)	Time (s)
AES_V	RTL Impl. (Verilog)	1756	8/5	324†	110	63
AES_C	Software Model (C)	328	8/7	292†	63	
$Pipe_I$	RTL Impl. (Verilog)	218	4/0	78	-	25

† ILA synthesis template. Note: sizes are LoC w.r.t the corresponding language.

3.1 Advanced Encryption Standard (AES)

In this case study, we showcase the synthesis engine (Sect. 2.2) and the verification utilities (Sect. 2.3) for ILA vs. ILA behavioral equivalence checking.

We synthesized two ILAs for the AES crypto-engine, AES_C and AES_V, based on C and Verilog implementations respectively. They have the same instruction set, but with differences in block-level and round-level algorithms. As shown in Table 1, the sizes of ILAs (synthesis templates) are *significantly smaller* than the final RTL implementation, making this an attractive entry point for verification.

The equivalence between AES_C and AES_V is checked modularly, i.e., per instruction. Table 1 shows the verification time for checking behavioral equivalence using the SMT solver z3.

3.2 A Simple Instruction Execution Pipeline

In this case study, we demonstrate the steps in manually defining an ILA (Sect. 2.2) and exporting it in Verilog (Sect. 2.4) for ILA vs. FSM behavioral equivalence checking using existing hardware verification tools.

[1] All tutorial case studies are available in the submitted artifact and on GitHub.

This pipeline case study is a simple version of the back-end of a pipelined processor. We manually define an ILA model Pipe_I as the specification of the design. This specification can be verified against a detailed RTL implementation, using a given refinement relation. We exported the Verilog model (including Pipe_I and property assertions) and utilized Yosys and z3 for hardware verification. The equivalence is checked modularly per instruction, and took 22 s in total for all four instructions, as shown in Table 1.

4 Other ILAng Applications

Firmware/Hardware Co-verification. The ILA models program-visible hardware behavior while abstracting out lower-level implementation details. This enables scalable firmware/hardware co-verification, as demonstrated in our previous work on security verification of firmware in industrial SoCs using ILAs [3].

Reasoning about Concurrency and Memory Consistency. The ILA model is an operational model that captures program-visible state updates. When integrated with axiomatic memory consistency models that specify orderings between memory operations, the transition relation defined in ILAs (Sect. 2.3) can be used to reason about concurrent interactions between heterogeneous components [8].

Data Race Checking of GPU Programs. Besides general-purpose processors and accelerators, an ILA model has been synthesized for the nVidia GPU PTX instruction set using the synthesis engine (Sect. 2.2) [7]. This model has then been used for data race checking for GPU programs using the BMC utility (Sect. 2.3).

References

1. Burch, J.R., Dill, D.L.: Automatic verification of pipelined microprocessor control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 68–80. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_44
2. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
3. Huang, B.Y., Ray, S., Gupta, A., Fung, J.M., Malik, S.: Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In: DAC, pp. 1–6 (2018)
4. Huang, B.Y., Zhang, H., Subramanyan, P., Vizel, Y., Gupta, A., Malik, S.: Instruction-Level Abstraction (ILA): a uniform specification for System-on-Chip (SoC) verification. ACM TODAES **24**(1), 10 (2018)
5. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: ICSE, pp. 215–224 (2010)
6. Subramanyan, P., Huang, B.Y., Vizel, Y., Gupta, A., Malik, S.: Template-based parameterized synthesis of uniform instruction-level abstractions for SoC verification. IEEE TCAD **37**(8), 1692–1705 (2018)

7. Xing, Y., Huang, B.Y., Gupta, A., Malik, S.: A formal instruction-level GPU model for scalable verification. In: ICCAD, pp. 130–135 (2018)
8. Zhang, H., Trippel, C., Manerkar, Y.A., Gupta, A., Martonosi, M., Malik, S.: ILA-MCM: integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. In: FMCAD (2018)
9. Huang, B.-Y., Zhang, H., Gupta, A., Malik, S.: ILAng: A Modeling and Verification Platform for SoCs using Instruction-Level Abstractions (artifact). Figshare (2019). https://doi.org/10.6084/m9.figshare.7808960.v1

MetAcsl: Specification and Verification of High-Level Properties

Virgile Robles[1(✉)], Nikolai Kosmatov[1], Virgile Prevosto[1], Louis Rilling[2], and Pascale Le Gall[3]

[1] Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France
{virgile.robles,nikolai.kosmatov, virgile.prevosto}@cea.fr
[2] DGA, Bruz, France
louis.rilling@irisa.fr
[3] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France
pascale.legall@centralesupelec.fr

Abstract. Modular deductive verification is a powerful technique capable to show that each function in a program satisfies its contract. However, function contracts do not provide a global view of which high-level (e.g. security-related) properties of a whole software module are actually established, making it very difficult to assess them. To address this issue, this paper proposes a new specification mechanism, called meta-properties. A meta-property can be seen as an enhanced global invariant specified for a set of functions, and capable to express predicates on values of variables, as well as memory related conditions (such as separation) and read or write access constraints. We also propose an automatic transformation technique translating meta-properties into usual contracts and assertions, that can be proved by traditional deductive verification tools. This technique has been implemented as a Frama-C plugin called MetAcsl and successfully applied to specify and prove safety- and security-related meta-properties in two illustrative case studies.

1 Introduction

Modular deductive verification is a well-known technique for formally proving that a program respects some user-defined properties. It consists in providing for each function of the program a *contract*, which basically contains a *precondition* describing what the function expects from its callers, and a *postcondition* indicating what it guarantees when it successfully returns. Logical formulas, known as *verification conditions* or *proof obligations* (POs), can then be generated and given to automated theorem provers. If all POs are validated, the body of the function fulfills its contract. Many deductive verification frameworks exist for various programming and formal specification languages. We focus here on

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 358–364, 2019.
https://doi.org/10.1007/978-3-030-17462-0_22

Frama-C [1] and its deductive verification plugin Wp, which allows proving a C program correct with respect to a formal specification expressed in ACSL [1].

However, encoding *high-level* properties spanning across the entire program in a set of Pre/Post-based contracts is not always immediate. In the end, such high-level properties get split among many different clauses in several contracts, without an explicit link between them. Therefore, even if each individual clause is formally proved, it might be very difficult for a verification engineer, a code reviewer or a certification authority to convince themselves that the provided contracts indeed ensure the expected high-level properties. Moreover, a software product frequently evolves during its lifetime, leading to numerous modifications in the code and specifications. Maintaining a high-level (e.g. security-related) property is extremely complex without a suitable mechanism to formally specify and automatically verify it after each update.

The purpose of the present work is to propose such a specification mechanism for high-level properties, which we call *meta-properties*, and to allow their automatic verification on C code in Frama-C thanks to a new plugin called MetAcsl.

Motivation. This work was motivated by several previous projects. During the verification of a hypervisor, we observed the need for a mechanism of specification and automatic verification of high-level properties, in particular, for global properties related to isolation and memory separation. Isolation properties are known as key properties in many verification projects, in particular, for hypervisors and micro-kernels.

A similar need for specific high-level properties recently arose from a case study on a confidentiality-oriented page management system submitted by an industrial partner. In this example, each page and each user (process) are given a confidentiality level, and we wish to specify and verify that in particular:

- (P_{read}) a user cannot read data from a page with a confidentiality level higher than its own;
- (P_{write}) a user cannot write data to a page with a confidentiality level lower than its own.

This case study will be used as a running example in this paper. As a second case study (also verified, but not detailed in this paper), we consider a simple smart house manager with several interesting properties such as: "a door can only be unlocked after a proper authentication or in case of alarm" or "whenever the alarm is ringing, all doors must be unlocked". Again, these examples involve properties that are hard to express with function contracts since they apply to the entire program rather than a specific function.[1]

Contributions. The contributions of this paper[2] include:

- a new form of high-level properties, which we call *meta-properties*, and an extension of the ACSL language able to express them (Sect. 2),

[1] These examples are publicly available at https://huit.re/metacas.
[2] A longer version is available at https://arxiv.org/abs/1811.10509.

– a set of code transformations to translate meta-properties into native ACSL annotations that can be proved via the usual methods (Sect. 3),
– a Frama-C plugin MetAcsl able to parse C code annotated with meta-properties and to perform the aforementioned code transformations (Sect. 4),
– a case study: a confidentiality-oriented page system, where important security guarantees were expressed using meta-properties and automatically verified thanks to the code transformation with MetAcsl (Sect. 4).

```
1  struct Page { //Page handler structure
2    char* data; //First address of the page
3    enum allocation status; //ALLOCATED or FREE (ensured by M1, lines 10-12)
4    enum confidentiality level; /*Page level, CONFIDENTIAL or PUBLIC*/ }
5  enum confidentiality user_level; //Current user process level
6  struct Page metadata[PAGE_NB]; //All pages
7  struct Page* page_alloc(void); //Allocates a page
8  void page_read(struct Page*, char* buffer); //Reads a page
9  void page_encrypt(struct Page*); //Encrypts a page in place, makes it PUBLIC
10 /*@ meta M1: ∀function f; \strong_invariant(f),
11     ∀ int page; 0 ≤ page < PAGE_NB ⇒
12     metadata[page].status == FREE ∨ metadata[page].status == ALLOCATED;
13   meta M2: ∀function f; //Only page_encrypt can change levels of allocated
         pages
14     ! \subset(f, {page_encrypt}) ⇒ \writing(f),
15     ∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
16     ⇒ \separated(\written, &metadata[page].level);
17   meta M3: ∀function f; \reading(f), //Ensures P_read
18     ∀ int page; 0 ≤ page < PAGE_NB ∧ metadata[page].status == ALLOCATED
19     ∧ user_level == PUBLIC ∧ metadata[page].level == CONFIDENTIAL
20     ⇒ \separated(\read, metadata[page].data + (0 .. PAGE_LENGTH - 1));
21 */ //Meta-property ensuring  P_write is defined similarly to M3
```

Fig. 1. Partial meta-specification of a confidentiality case study

2 Specification of Meta-properties

A meta-property is a property meant to express high-level requirements. As such, it is not attached to any particular function but instead to a set of functions. It is thus defined in the global scope and can only refer to global objects.

To define a meta-property, the user must provide (i) the set of functions it will be applied to, (ii) a property (expressed in ACSL) and (iii) the *context*, *i.e.* a characterization of the situations in which they want the property to hold in each of these functions (everywhere in the function, only at the beginning and the end, upon writing in a variable, etc.). Furthermore, depending on the context, the property can refer to some special variables which we call *meta-variables*. Figure 1 features a few examples of meta-properties further explained below.

Let $\mathcal{F}$ denote the set of functions defined in the current program, and $\mathcal{P}$ the set of native ACSL properties. Formally, we can define a meta-property as a triple (c, F, P), where c is a context (see Sect. 2.2), $F \subseteq \mathcal{F}$ and $P \in \mathcal{P}$. Intuitively, we can interpret this triple as "$\forall f \in F$, P holds for f in the context c". For the meta-property to be well-formed, P must be a property over a subset

of $\mathcal{G} \cup \mathcal{M}(c)$, where $\mathcal{G}$ is the set of variables available in the global scope of the program and $\mathcal{M}(c)$ is the set of meta-variables provided by the context c.

The actual MetAcsl syntax for defining a meta-property (c, F, P) is `meta [specification of F] c, P;` An example is given by property M_1 (cf. lines 10–12 in Fig. 1), where $F = \mathcal{F}$, $c =$ `strong_invariant` and P is the predicate stating that the status of any page should be either `FREE` or `ALLOCATED`.

2.1 Target Functions and Quantification

Meta-properties are applied to a given *target set* of functions F defined as $F = F_+ \setminus F_-$ by providing explicit lists of considered and excluded functions $F_+, F_- \subseteq \mathcal{F}$. If not provided, F_+ and F_- are respectively equal to $\mathcal{F}$ and $\emptyset$ by default, *i.e.* the meta-property should hold for all functions of the program. F_- is useful when the user wants to target every function except a few, since they do not have to explicitly provide every resulting target function.

The MetAcsl syntax for the specification of F uses the built-in ACSL construction `\forall`, possibly followed by `\subset` with or without logic negation `!` (to express $f \in F_+$ and $f \notin F_-$). It can be observed in property M_2 (lines 13–16), where $F_+ = \mathcal{F}$ and $F_- = \{$`page_encrypt`$\}$ excludes only one function.

2.2 Notion of Context

The *context* c of a meta-property defines the states in which property P must hold, and may introduce *meta-variables* that can be used in the definition of P.

Beginning/Ending Context (Weak Invariant). A *weak invariant* indicates that P must hold at the beginning and at the end of each target function $f \in F$.

Everywhere Context (Strong invariant). A *strong invariant* is similar to a weak invariant, except that it ensures that P holds at *every point*[3] of each target function. For example, property M_1 specifies that at every point of the program, the status of any page must be either `FREE` or `ALLOCATED`.

Writing Context. This ensures that P holds upon any modification of the memory (both stack and heap). It provides a meta-variable `\written` that refers to the variable (and, more generally, the memory location) being written to.

A simple usage of this context can be to forbid any direct modification of some global variable, as in property M_2. This property states that for any function that is not `page_encrypt`, the left-hand side of any assignment must be *separated* from (that is, disjoint with) the global variable `metadata[page].level` for any `page` with the `ALLOCATED` status. In other words, only the `page_encrypt` function is allowed to modify the confidentiality level of an allocated page.

An important benefit of this setting is a *non-transitive restriction of modifications* that cannot be specified using the ACSL clause `assigns`, since the latter is transitive over function calls and necessarily permits to modify a variable when

[3] More precisely, every *sequence point* as defined by the C standard.

at least one callee has the right to modify it. Here, since we only focus on *direct* modifications, a call to `page_encrypt` (setting to public the level of the page it has encrypted) from another function does not violate meta-property M_2.

Furthermore, modification can be forbidden *under some condition* (i.e. that the page is allocated), while `assigns` has no such mechanism readily available.

Reading Context. Similar to the writing context, this ensures that the property holds whenever some memory location is read, and provides a meta-variable `\read` referring to the read location. It is used in property M_3 (lines 17–20), which expresses the guarantee P_{read} of the case study (see Motivation in Sect. 1) by imposing a separation of a read location and the contents of allocated confidential pages when the user does not have sufficient access rights. As another example, an isolation of a page can be specified as separation of all reads and writes from it.

These few simple contexts, combined with the native features of ACSL, turn out to be powerful enough to express quite interesting properties, including memory isolation and all properties used in our two motivating case studies.

3 Verification of Meta-properties

Figure 2 shows an (incorrect) toy implementation of two functions of Fig. 1 that we will use to illustrate the verification of meta-properties M_1–M_3.

The key idea of the verification is the translation of meta-properties into native ACSL annotations, which are then verified using existing Frama-C analyzers. To that end, the property P of a meta-property (c, F, P) must be inserted as an assertion in relevant locations (as specified by context c) in each target function $f \in F$, and the meta-variables (if any) must be instantiated.

We define a specific translation for each context. For weak invariants, property P is simply added as both a precondition and a postcondition in the contract of f. This is also done for the strong invariant, for which P is additionally inserted after each instruction potentially modifying the values of the free variables in P[4] For example, Fig. 3a shows the translation of M_1 on `page_alloc`. Our property (defined on lines 11–12 in Fig. 1, denoted P_{M_1} here) is inserted after the modification of a `status` field

```
1  struct Page* page_alloc() {
2    //try to find a free page
3    struct Page* fp = find_free_page();
4    //if a free page is found,
5    //allocate it with current user
         level
6    if(fp ≠ NULL) {
7      fp->status = ALLOCATED;
8      fp->level = user_level;
9    }
10   return fp;
11 }
12 void page_read(struct Page* from,
       char* buffer) {
13   for(i ∈ ℕ = 0 ; i < PAGE_LENGTH ;
         ++i)
14     buffer[i] = from->data[i];
15 }
```

Fig. 2. Incorrect code w.r.t. M_2 and M_3

[4] The AST is normalized so that every memory modification happens through an assignment. Then we conservatively determine if the object being assigned is one of the free variables of P: in presence of pointers, we assume the worst case.

(line 6) since the property involves these objects, but not after the modification of a `level` field (line 8).

For *Writing* (resp. *Reading*) contexts, P is inserted before any instruction potentially making a write (resp. read) access to the memory, with the exception of function calls. In addition, each meta-variable is replaced by its actual value. For example, in the translation of M_2 on `page_alloc` (Fig. 3b), the property is inserted before the two modifications of `fp`, and `\written` is replaced respectively by `fp->status` and `fp->level`. In this case M_2 does not hold. While its first instantiation (lines 4–6) is easily proved, it is not the case for the second one (lines 8–10). Indeed, there exists a `page` (the one being modified) that has a status set to ALLOCATED because of the previous instruction (line 7) and for which the `\separated` clause is obviously false. Hence, the assertion fails, meaning that the whole meta-property M_2 cannot be proved. The fix consists in swapping lines 6 and 7 in Fig. 2. After that, all assertions generated from M_2 are proved.

A similar transformation for M_3 on `page_read` shows that the proof fails since the implementation allows an agent to read from any page without any check. Adding proper guards allows the meta-property to be proved. Conversely, if a meta-property is broken by an erroneous code update, a proof failure after *automatically* re-running MetAcsl helps to easily detect it.

```
1 /*@ requires P_M1;
2   ensures P_M1; */
3 struct Page* page_alloc() {
4   struct Page* fp =
        find_free_page();
5   if(fp ≠ NULL) {
6     fp->status = ALLOCATED;
7     /*@ assert P_M1;*/
8     fp->level = user_level;
9     //Line 8 cannot break P_M1
10  }
11 }
```

(a) Transformation for M_1

```
1 struct Page* page_alloc() {
2   struct Page* fp = find_free_page();
3   if(fp ≠ NULL) {
4     /*@ assert ∀ int page; 0 ≤ page < PAGE_NB
5       ⇒ metadata[page].status == ALLOCATED
6       ⇒ \separated(fp->status,
            &metadata[page].level);*/
7     fp->status = PAGE_ALLOCATED;
8     /*@ assert ∀ int page; 0 ≤ page < PAGE_NB
9       ⇒ metadata[page].status == ALLOCATED
10      ⇒ \separated(fp->level,
            &metadata[page].level);*/
11    fp->level = user_level;
12  }
13 }
```

(b) Transformation for M_2

Fig. 3. Examples of code transformations for functions of Fig. 2

4 Results on Case Studies and Conclusion

Experiments. The support of meta-properties and the proposed methodology for their verification were fully implemented in OCaml as a Frama-C plugin called MetAcsl. We realized a simple implementation of the two case studies mentioned in Sect. 1 and were able to fully specify and automatically verify all aforementioned properties (in particular P_{read} and P_{write}) using MetAcsl. The transformation step is performed in less than a second while the automatic proof takes generally less than a minute.

Conclusion. We proposed a new specification mechanism for high-level properties in Frama-C, as well as an automatic transformation-based technique to verify these properties by a usual deductive verification approach. The main idea

of this technique is similar to some previous efforts e.g. [2]. Meta-properties provide a useful extension to function contracts offering the possibility to express a variety of high-level safety- and security-related properties. They also provide a verification engineer with an explicit global view of high-level properties being really proved, avoiding the risk to miss some part of an implicit property which is not formally linked to relevant parts of several function contracts, thus facilitating code review and certification. Another benefit of the new mechanism is the possibility to easily re-execute a proof after a code update. Initial experiments confirm the interest of the proposed solution.

Future Work. We plan to establish a formal soundness proof for our transformation technique, thereby allowing MetAcsl to be reliably used for critical code verification. Other future work directions include further experiments to evaluate the proposed approach on real-life software and for more complex properties.

Acknowledgment. This work was partially supported by the project VESSEDIA, which has received funding from the EU Horizon 2020 research and innovation programme under grant agreement No 731453. The work of the first author was partially funded by a Ph.D. grant of the French Ministry of Defense. Many thanks to the anonymous referees for their helpful comments.

References

1. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. FAOC **27**, 573–609 (2015)
2. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.L.: Enforcing high-level security properties for applets. In: Quisquater, J.J., Paradinas, P., Deswarte, Y., El Kalam, A.A. (eds.) Smart Card Research and Advanced Applications VI. IFIP International Federation for Information Processing, vol. 153. Springer, Boston (2004). https://doi.org/10.1007/1-4020-8147-2_1

ROLL 1.0: ω-Regular Language Learning Library

Yong Li[1,2], Xuechao Sun[1,2], Andrea Turrini[1,3(✉)], Yu-Fang Chen[4], and Junnan Xu[1,2]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
turrini@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing, China
[3] Institute of Intelligent Software, Guangzhou, China
[4] Institute of Information Science, Academia Sinica, Taipei, Taiwan

Abstract. We present ROLL 1.0, an ω-regular language learning library with command line tools to learn and complement Büchi automata. This open source Java library implements all existing learning algorithms for the complete class of ω-regular languages. It also provides a learning-based Büchi automata complementation procedure that can be used as a baseline for automata complementation research. The tool supports both the Hanoi Omega Automata format and the BA format used by the tool RABIT. Moreover, it features an interactive Jupyter notebook environment that can be used for educational purpose.

1 Introduction

In her seminal work [3], Angluin introduced the well-known algorithm L^* to learn regular languages by means of deterministic finite automata (DFAs). In the learning setting presented in [3], there is a *teacher*, who knows the target language L, and a *learner*, whose task is to learn the target language, represented by an automaton. The learner interacts with the teacher by means of two kinds of queries: *membership queries* and *equivalence queries*. A membership query $\mathrm{MQ}(w)$ asks whether a string w belongs to L while an equivalence query $\mathrm{EQ}(A)$ asks whether the conjectured DFA A recognizes L. The teacher replies with a witness if the conjecture is incorrect otherwise the learner completes its job. This learning setting now is widely known as *active automata learning*. In recent years, active automata learning algorithms have attracted increasing attention in the computer aided verification community: it has been applied in *black-box* model checking [24], *compositional verification* [12], *program verification* [10], *error localization* [8], and *model learning* [26].

Due to the increasing importance of automata learning algorithms, many efforts have been put into the development of automata learning libraries such as libalf [6] and LearnLib [18]. However, their focus is only on automata accepting

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 365–371, 2019.
https://doi.org/10.1007/978-3-030-17462-0_23

finite words, which correspond to safety properties. The ω-regular languages are the standard formalism to describe liveness properties. The problem of learning the complete class of ω-regular languages was considered open until recently, when it has been solved by Farzan *et al.* [15] and improved by Angluin *et al.* [4].

However, the research on applying ω-regular language learning algorithms for verification problems is still in its infancy. Learning algorithms for ω-regular languages are admittedly much more complicated than their finite regular language counterparts. This becomes a barrier for the researchers doing further investigations and experiments on such topics. We present ROLL 1.0, an open-source library implementing all existing learning algorithms for the complete class of ω-regular languages known in literature, which we believe can be an enabling tool for this direction of research. To the best of our knowledge, ROLL 1.0 is the only publicly available tool focusing on ω-regular language learning.

ROLL, a preliminary version of ROLL 1.0, was developed in [22] to compare the performance of different learning algorithms for Büchi automata (BAs). The main improvements made in ROLL 1.0 compared to its previous version are as follows. ROLL 1.0 rewrites the algorithms in the core part of ROLL and obtains high modularity to allow for supporting the learning algorithms for more types of ω-automata than just BAs, algorithms to be developed in future. In addition to the BA format [1,2,11], ROLL 1.0 now also supports the Hanoi Omega Automata (HOA) format [5]. Besides the learning algorithms, ROLL 1.0 also contains complementation [23] and a new language inclusion algorithm. Both of them are built on top of the BAs learning algorithms. Experiments [23] have shown that the resulting automata produced by the learning-based complementation can be much smaller than those built by structure-based algorithms [7,9,19,21,25]. Therefore, the learning-based complementation is suitable to serve as a baseline for Büchi automata complementation researches. The language inclusion checking algorithm implemented in ROLL 1.0 is based on learning and a Monte Carlo word sampling algorithm [17]. ROLL 1.0 features an interactive mode which is used in the ROLL Jupyter notebook environment. This is particularly helpful for teaching and learning how ω-regular language learning algorithms work.

2 ROLL 1.0 Architecture and Usage

ROLL 1.0 is written entirely in Java and its architecture, shown in Fig. 1, comprises two main components: the **Learning Library**, which provides all known existing learning algorithms for Büchi automata, and the **Control Center**, which uses the learning library to complete the input tasks required by the user.

Learning Library. The learning library implements all known BA learning algorithms for the full class of ω-regular languages: the $L^{\$}$ learner [15], based on DFA learning [3], and the L^{ω} learner [22], based on three canonical *family of DFAs* (FDFAs) learning algorithms [4,22]. ROLL 1.0 supports both *observation tables* [3] and *classification trees* [20] to store membership query answers. All learning algorithms provided in ROLL 1.0 implement the `Learner` interface; their

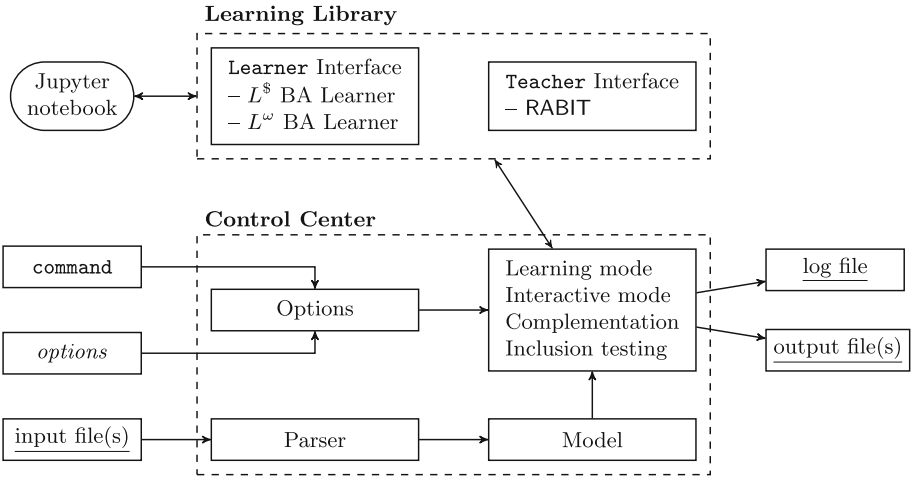


Fig. 1. Architecture of ROLL 1.0

corresponding teachers implement the `Teacher` interface. Any Java object that implements `Teacher` and can decide the equivalence of two Büchi automata is a valid teacher for the BA learning algorithms. Similarly, any Java object implementing `Learner` can be used as a learner, making ROLL 1.0 easy to extend with new learning algorithms and functionalities. The BA teacher implemented in ROLL 1.0 uses RABIT [1,2,11] to answer the equivalence queries posed by the learners since the counterexamples RABIT provides tend to be short and hence are easier to analyze; membership queries are instead answered by implementing the ASCC algorithm from [16].

Control Center. The control center is responsible for calling the appropriate learning algorithm according to the user's command and options given at command line, which is used to set the `Options`. The file formats supported by ROLL 1.0 for the input automata are the RABIT BA format [1,2,11] and the standard Hanoi Omega Automata (HOA) format [5], identified by the file extensions .ba and .hoa, respectively. Besides managing the different execution modes, which are presented below, the control center allows for saving the learned automaton into a given file (option *-out*), for further processing, and to save execution details in a log file (option *-log*). The output automaton is generated in the same format of the input. The standard way to call ROLL 1.0 from command line is

`java -jar ROLL.jar command` input file(s) [*options*]

Learning mode (command `learn`) makes ROLL 1.0 learn a Büchi automaton equivalent to the given Büchi automaton; this can be used, for instance, to get a possibly smaller BA. The default option for storing answers to membership

queries is *-table*, which selects the observation tables; classification trees can be chosen instead by means of the *-tree* option.

```
java -jar ROLL.jar learn aut.hoa
```

for instance runs ROLL 1.0 in learning mode against the input BA aut.hoa; it learns aut.hoa by means of the L^ω learner using observation tables. The three canonical FDFA learning algorithms given in [4] can be chosen by means of the options *-syntactic* (default), *-recurrent*, and *-periodic*. Options *-under* (default) and *-over* control which approximation is used in the L^ω learner [22] to transform an FDFA to a BA. By giving the option *-ldollar*, ROLL 1.0 switches to use the $L^\$$ learner instead of the default L^ω learner.

Interactive mode (command `play`) allows users to play as the teacher guiding ROLL 1.0 in learning the language they have in mind. To show how the learning procedure works, ROLL 1.0 outputs each intermediate result in the Graphviz dot layout format[1]; users can use Graphviz's tools to get a graphical view of the output BA so to decide whether it is the right conjecture.

Complementation (command `complement`) of the BA $\mathcal{B}$ in ROLL 1.0 is based on the algorithm from [23] which learns the complement automaton $\mathcal{B}^{\mathsf{c}}$ from a teacher who knows the language $\Sigma^\omega \setminus \mathcal{L}(\mathcal{B})$. This allows ROLL 1.0 to disentangle $\mathcal{B}^{\mathsf{c}}$ from the structure of $\mathcal{B}$, avoiding the $\Omega((0.76n)^n)$ blowup [27] of the structure-based complementation algorithms (see., e.g., [7,19,21,25]).

Inclusion testing (command `include`) between two BAs $\mathcal{A}$ and $\mathcal{B}$ is implemented in ROLL 1.0 as follows: (1) first, sample several ω-words $w \in \mathcal{L}(\mathcal{A})$ and check whether $w \notin \mathcal{L}(\mathcal{B})$ to prove $\mathcal{L}(\mathcal{A}) \not\subseteq \mathcal{L}(\mathcal{B})$; (2) then, try simulation techniques [11,13,14] to prove inclusion; (3) finally, use the learning based complementation algorithm to check inclusion. The ROLL 1.0's ω-word sampling algorithm is an extension of the one proposed in [17]. The latter only samples paths visiting any state at most twice while ROLL 1.0's variant allows for sampling paths visiting any state at most K times, where K is usually set to the number of states in $\mathcal{A}$. In this way, ROLL 1.0 can get a larger set of ω-words accepted by $\mathcal{A}$ than the set from the original algorithm.

Online availability of ROLL 1.0. ROLL 1.0 is an open-source library freely available online at https://iscasmc.ios.ac.cn/roll/, where more details are provided about its commands and options, its use as a Java library, and its GitHub repository[2]. Moreover, from the roll page, it is possible to access an online Jupyter notebook[3] allowing to interact with ROLL 1.0 without having to download and compile it. Each client gets a new instance of the notebook, provided by JupyterHub[4], so to avoid unexpected interactions between different users. Figure 2 shows few screenshots of the notebook for learning in interactive mode the language $\Sigma^* \cdot b^\omega$ over the alphabet $\Sigma = \{a, b\}$. As we can see, the membership query

[1] https://www.graphviz.org/.
[2] https://github.com/ISCAS-PMC/roll-library.
[3] https://iscasmc.ios.ac.cn/roll/jupyter.
[4] https://jupyterhub.readthedocs.io/.

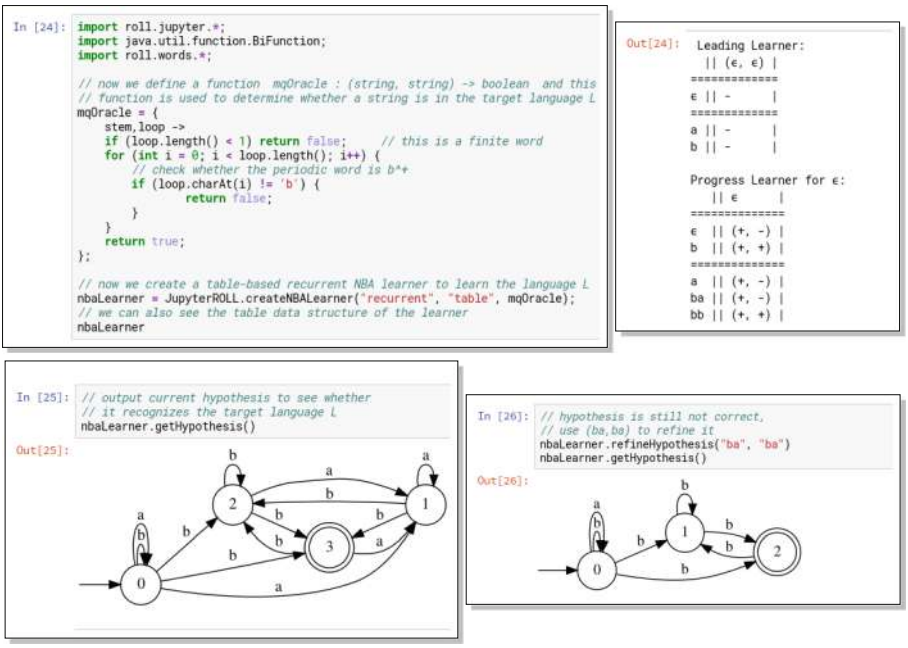


Fig. 2. ROLL 1.0 running in the Jupyter notebook for interactively learning $\Sigma^* \cdot b^\omega$

MQ(w) is answered by means of the `mqOracle` function: it gets as input two finite words, the `stem` and the `loop` of the ultimately periodic word w, and it checks whether `loop` contains only b. Then one can create a BA learner with the oracle `mqOracle`, say the BA learner `nbaLearner`, based on observation tables and the recurrent FDFAs, as shown in the top-left screenshot. One can check the internal table structures of `nbaLearner` by printing out the learner, as in the top-right screenshot. The answer to an equivalence query is split in two parts: first, the call to `getHypothesis()` shows the currently conjectured BA; then, the call to `refineHypothesis("ba", "ba")` simulates a negative answer with counterexample $ba \cdot (ba)^\omega$. After the refinement by `nbaLearner`, the new conjectured BA is already the right conjecture.

Acknowledgement. This work has been supported by the National Natural Science Foundation of China (Grants Nos. 61532019, 61761136011) and by the CAP project GZ1023.

References

1. Abdulla, P.A., et al.: Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 132–147. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_14
2. Abdulla, P.A., et al.: Advanced Ramsey-based Büchi automata inclusion testing. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 187–202. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_13
3. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
4. Angluin, D., Fisman, D.: Learning regular omega languages. Theoret. Comput. Sci. **650**, 57–72 (2016)
5. Babiak, T., et al.: The Hanoi Omega-Automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
6. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: the automata learning framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_32
7. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: International Congress on Logic, Methodology and Philosophy of Science, pp. 1–11 (1962)
8. Chapman, M., Chockler, H., Kesseli, P., Kroening, D., Strichman, O., Tautschnig, M.: Learning the language of error. In: Finkbeiner, B., Pu, G., Zhang, L. (eds.) ATVA 2015. LNCS, vol. 9364, pp. 114–130. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24953-7_9
9. Chen, Y.-F., et al.: Advanced automata-based algorithms for program termination checking. In: PLDI, pp. 135–150 (2018)
10. Chen, Y.-F., et al.: PAC learning-based verification and model synthesis. In: ICSE, pp. 714–724 (2016)
11. Clemente, L., Mayr, R.: Advanced automata minimization. In: POPL, pp. 63–74 (2013)
12. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_24
13. Dill, D.L., Hu, A.J., Wong-Toi, H.: Checking for language inclusion using simulation preorders. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 255–265. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55179-4_25
14. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 694–707. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_57
15. Farzan, A., Chen, Y.-F., Clarke, E.M., Tsay, Y.-K., Wang, B.-Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_2
16. Gaiser, A., Schwoon, A.: Comparison of algorithms for checking emptiness of Büchi automata. In: MEMICS (2009)

17. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_18
18. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
19. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70575-8_59
20. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
21. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Logic **2**(3), 408–429 (2001)
22. Li, Y., Chen, Y.-F., Zhang, L., Liu, D.: A novel learning algorithm for Büchi automata based on family of DFAs and classification trees. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 208–226. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_12
23. Li, Y., Turrini, A., Zhang, L., Schewe, S.: Learning to complement Büchi automata. In: Dillig, I., Palsberg, J. (eds.) VMCAI 2018. LNCS, vol. 10747, pp. 313–335. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73721-8_15
24. Peled, D., Vardi, M.Y., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2001)
25. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: LICS, pp. 255–264 (2006)
26. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)
27. Yan, Q.: Lower bounds for complementation of omega-automata via the full automata technique. Logical Methods Comput. Sci. **4**(1) (2008)

Symbolic Regex Matcher

Olli Saarikivi[1], Margus Veanes[1(✉)], Tiki Wan[2], and Eric Xu[2]

[1] Microsoft Research, Redmond, USA
margus@microsoft.com
[2] Microsoft Azure, Redmond, USA

Abstract. Symbolic regex matcher is a new open source .NET regular expression matching tool and match generator in the Microsoft Automata framework. It is based on the .NET regex parser in combination with a set based representation of character classes. The main feature of the tool is that the core matching algorithms are based on symbolic derivatives that support extended regular expression operations such as intersection and complement and also support a large set of commonly used features such as bounded loop quantifiers. The particularly useful features of the tool are that it supports full UTF16 encoded strings, the match generation is backtracking free, thread safe, and parallelizes with low overhead in multithreaded applications. We discuss the main design decisions behind the tool, explain the core algorithmic ideas and how the tool works, discuss some practical usage scenarios, and compare it to existing state of the art.

1 Motivation

We present a new tool called *Symbolic Regex Matcher* or *SRM* for fast match generation from extended regular expressions. The development of SRM has been motivated by some concrete industrial use cases and should meet the following *expectations*. Regarding *performance*, the overall algorithm complexity of match generation should be *linear* in the length of the input string. Regarding *expressivity*, it should handle common types of .NET regexes, including support for *bounded quantifiers* and *Unicode categories*; while nonregular features of regexes, such as back-references, are not required. Regarding *semantics*, the tool should be .NET compliant regarding strings and regexes, and the main type of match generation is: *earliest eager nonoverlapping* matches in the input string. Moreover, the tool should be safe to use in distributed and *multi threaded* development environments. Compilation time should be reasonable but it is not a critical factor because the intent is that the regexes are used frequently but updated infrequently. A concrete application of SRM is in an internal tool at Microsoft that scans for credentials and other sensitive content in cloud service software, where the search patterns are stated in form of individual regexes or in certain scenarios as intersections of regexes.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 372–378, 2019.
https://doi.org/10.1007/978-3-030-17462-0_24

The built-in .NET regex engine uses a backtracking based match search algorithm and does not meet the above expectations; in particular, some patterns may cause *exponential* search time. While SRM uses the *same parser* as the .NET regex engine, its back-end is a new engine that is built on the notion of *derivatives* [1], is developed as a tool in the open source Microsoft Automata framework [5], the framework was originally introduced in [8]. SRM meets *all* of the above expectations. Derivatives of regular expressions have been studied before in the context of matching of regular expressions, but only in the functional programming world [2,6] and in related domains [7]. Compared to earlier derivative based matching engines, the new contribution of SRM is that it supports *match generation* not only match detection, it supports extended features, such as *bounded quantifiers*, *Unicode categories*, and *case insensitivity*, it is .NET compliant, and is implemented in an imperative language. As far as we are aware of, SRM is the first tool that supports derivative based match generation for extended regular expressions. In our evaluation SRM shows significant performance improvements over .NET, with more predictable performance than RE2 [3], a state of the art automata based regex matcher.

In order to use SRM in a .NET application instead of the built-in match generator, `Microsoft.Automata.dll` can be built from [5] on a .NET platform version 4.0 or higher. The library extends the built-in `Regex` class with methods that expose SRM, in particular through the `Compile` method.

2 Matching with Derivatives

Here we work with derivatives of *symbolic extended regular expressions* or *regexes* for short. *Symbolic* means that the basic building blocks of single character regexes are *predicates* as opposed to singleton characters. In the case of standard .NET regexes, these are called *character classes*, such as the class of digits or `\d`. In general, such predicates are drawn from a given effective Boolean algebra and are here denoted generally by α and β; $\bot$ denotes the *false* predicate and `.` the *true* predicate. For example, in .NET $\bot$ can be represented by the empty character class `[0-[0]]`.[1] *Extended* here means that we allow *intersection*, *complement*, and *bounded quantifiers*.

The abstract syntax of regexes assumed here is the following, assuming the usual semantics where () denotes the empty sequence ϵ and $\langle\alpha\rangle$ denotes any singleton sequence of character that belongs to the set $[\![\alpha]\!] \subseteq \Sigma$, where Σ is the alphabet, and n and m are nonnegative integers such that $n \leq m$:

$$() \quad \langle\alpha\rangle \quad R_1R_2 \quad R\{n,m\} \quad R\{n,*\} \quad R_1|R_2 \quad R_1\&R_2 \quad \neg R$$

where $[\![R\{n,m\}]\!] \stackrel{\text{def}}{=} \{v \in [\![R]\!]^i \mid n \leq i \leq m\}$, $[\![R\{n,*\}]\!] \stackrel{\text{def}}{=} \{v \in [\![R]\!]^i \mid n \leq i\}$. The expression $R*$ is a shorthand for $R\{0,*\}$. We write $\bot$ also for $\langle\bot\rangle$. We assume that $[\![R_1|R_2]\!] = [\![R_1]\!] \cup [\![R_2]\!]$, $[\![R_1\&R_2]\!] = [\![R_1]\!] \cap [\![R_2]\!]$, and $[\![\neg R]\!] = \Sigma^* \setminus [\![R]\!]$.

[1] The more intuitive syntax `[]` is unfortunately not allowed.

A less known feature of the .NET regex grammar is that it also supports *if-then-else* expressions over regexes, so, when combined appropriately with $\bot$ and $\mathbf{.}$, it also supports intersection and complement. R is *nullable* if $\epsilon \in [\![R]\!]$. Nullability is defined recursively, e.g., $R\{n,m\}$ is nullable iff R is nullable or $n = 0$.

Given a concrete character x in the underlying alphabet Σ, and a regex R, the *x-derivative of* R, denoted by $\partial_x R$, is defined on the right. Given a language $L \subseteq \Sigma^*$, the x-derivative of L, $\boldsymbol{\partial}_x L \stackrel{\text{def}}{=} \{v \mid xv \in L\}$. It is well-known that $[\![\partial_x R]\!] = \boldsymbol{\partial}_x [\![R]\!]$. The abstract derivation rules provide a way to decide if an input u matches a regex R as follows. If $u = \epsilon$ then u matches R iff R is nullable; else, if $u = xv$ for some $x \in \Sigma, v \in \Sigma^*$ then u matches R iff v matches $\partial_x R$. In other words, the derivation rules can be unfolded lazily to create the transitions of the underlying DFA. In this setting we are considering Brzozowski derivatives [1].

$$
\begin{aligned}
\partial_x() &\stackrel{\text{def}}{=} \bot \\
\partial_x\langle\alpha\rangle &\stackrel{\text{def}}{=} \begin{cases} (), \text{ if } x \in [\![\alpha]\!]; \\ \bot, \text{ otherwise.} \end{cases} \\
\partial_x(R_1 R_2) &\stackrel{\text{def}}{=} \begin{cases} ((\partial_x R_1)R_2)|\partial_x R_2, \text{ if } R_1 \text{ is nullable;} \\ (\partial_x R_1)R_2, \text{ otherwise.} \end{cases} \\
\partial_x R\{n,m\} &\stackrel{\text{def}}{=} \begin{cases} (\partial_x R)R\{n{-}1, m{-}1\}, \text{ if } n{>}0; \\ (\partial_x R)R\{0, m{-}1\}, \text{ if } n{=}0 \text{ and } m{>}0; \\ \bot, \text{ otherwise (since } R\{0,0\} \stackrel{\text{def}}{=} ()). \end{cases} \\
\partial_x R\{n,*\} &\stackrel{\text{def}}{=} \begin{cases} (\partial_x R)R\{n-1,*\}, \text{ if } n > 0; \\ (\partial_x R)R\{0,*\}, \text{ otherwise.} \end{cases} \\
\partial_x(R_1|R_2) &\stackrel{\text{def}}{=} (\partial_x R_1)|(\partial_x R_2) \\
\partial_x(R_1 \& R_2) &\stackrel{\text{def}}{=} (\partial_x R_1)\&(\partial_x R_2) \\
\partial_x \neg R &\stackrel{\text{def}}{=} \neg \partial_x R
\end{aligned}
$$

Match Generation. The main purpose of the tool is to *generate* matches. While match generation is a topic that has been studied extensively for classical regular expressions, we are not aware of efforts that have considered the use of derivatives and extended regular expressions in this context, while staying *backtracking free* in order to guarantee *linear complexity* in terms of the length of the input. Our matcher implements by default *nonoverlapping earliest eager* match semantics. An important property in the matcher is that the above set of regular expressions is closed under *reversal*. The reversal of regex R is denoted $R^{\mathbf{r}}$. Observe that:

$$(R_1 R_2)^{\mathbf{r}} \stackrel{\text{def}}{=} (R_2^{\mathbf{r}} R_1^{\mathbf{r}}) \quad R\{n,m\}^{\mathbf{r}} \stackrel{\text{def}}{=} R^{\mathbf{r}}\{n,m\} \quad R\{n,*\}^{\mathbf{r}} \stackrel{\text{def}}{=} R^{\mathbf{r}}\{n,*\}$$

It follows that $[\![R^{\mathbf{r}}]\!] = [\![R]\!]^{\mathbf{r}}$ where $L^{\mathbf{r}}$ denotes the reversal of $L \subseteq \Sigma^*$. The match generation algorithm can now be described at a high level as follows. Given a regex R, find all the (nonoverlapping earliest eager) matches in a given input string u. This procedure uses the three regexes: R, $R^{\mathbf{r}}$ and $\mathbf{.}{*}R$:

1. Initially $i = 0$ is the start position of the first symbol u_0 of u.
2. Let $i_{\text{orig}} = i$. Find the *earliest* match starting from i and $q = \mathbf{.}{*}R$: Compute $q := \partial_{u_i} q$ and $i := i + 1$ until q is nullable. **Terminate** if no such q exists.
3. Find the *start position* for the above match closest to i_{orig}: Let $p = R^{\mathbf{r}}$. While $i > i_{\text{orig}}$ let $p := \partial_{u_i} p$ and $i := i - 1$, if p is nullable let $i_{\text{start}} := i$.

4. Find the *end position* for the match: Let $q = R$ and $i = i_{\text{start}}$. Compute $q := \partial_{u_i} q$ and $i := i + 1$ and let $i_{\text{end}} := i$ if q is nullable; repeat until $q = \bot$.
5. **Return** the match from i_{start} to i_{end}.
6. Repeat step 2 from $i := i_{\text{end}} + 1$ for the next *nonoverlapping* start position.

Observe that step 4 guarantees *longest* match in R from the position i_{start} found in step 3 for the earliest match found in step 2. In order for the above procedure to be practical there are several optimizations that are required. We discuss some of the implementation aspects next.

3 Implementation

SRM is implemented in `C#`. The input to the tool is a .NET regex (or an array of regexes) that is compiled into a serializable object R that implements the main matching interface `IMatcher`. Initially, this process uses a Binary Decision Diagram (*BDD*) based representation of predicates in order to efficiently canonicalize various conditions such as *case insensitivity* and *Unicode categories*. The use of BDDs as character predicates is explained in [4]. Then all the BDDs that occur in R are collected and their *minterms* (satisfiable Boolean combinations) are calculated, called the *atoms* $(\alpha_1, \ldots, \alpha_k)$ *of* R, where $\{[\![\alpha_i]\!]_{\texttt{BDD}}\}_{i=1}^{k}$ forms a partition of Σ. Each `BDD`-predicate α in R is now translated into a k-bit bit-vector (or `BV`) value β whose i'th bit is 1 iff $\alpha \wedge_{\texttt{BDD}} \alpha_i$ is nonempty. Typically k is small (often $k \leq 64$) and allows `BV` to be implemented very efficiently (often by `ulong`), where $\wedge_{\texttt{BV}}$ is bit-wise-and. All subsequent Boolean operations are performed on this more efficient and *thread safe* data type. The additional step required during input processing is that each concrete input character c (`char` value) is now first mapped into an atom id i that determines the bit position in the `BV` predicate. In other words, $c \in [\![\beta]\!]_{\texttt{BV}}$ is implemented by finding the index i such that $c \in [\![\alpha_i]\!]_{\texttt{BDD}}$ and testing if the i'th bit of β is 1, where the former search is hardcoded into a precomputed lookup table or decision tree.

For example let R be constructed for the regex `\w\d*`. Then R has three atoms: $[\![\alpha_1]\!] = \Sigma \setminus [\![\texttt{\textbackslash w}]\!]$, $[\![\alpha_2]\!] = [\![\texttt{\textbackslash d}]\!]$, and $[\![\alpha_3]\!] = [\![\texttt{\textbackslash w}]\!] \setminus [\![\texttt{\textbackslash d}]\!]$, since $[\![\texttt{\textbackslash d}]\!] \subset [\![\texttt{\textbackslash w}]\!]$. For example `BV` 110_2 represents `\w` and 010_2 represents `\d`.

The symbolic regex AST type is treated as a value type and is handled similarly to the case of derivative based matching in the context of functional languages [2,6]. A key difference though, is that *weak equivalence* [6] checking is not enough to avoid state-space explosion when *bounded quantifiers* are allowed. A common situation during derivation is appearance of subexpressions of the form $(A\{0,k\}B)|(A\{0,k-1\}B)$ that, when kept unchecked, keep reintroducing disjuncts of the same subexpression but with smaller value of the upper bound, potentially causing a substantial blowup. However, we know that $A\{0,n\}B$ is *subsumed* by $A\{0,m\}B$ when $n \leq m$, thus $(A\{0,m\}B)|(A\{0,n\}B)$ can be simplified to $A\{0,m\}B$. To this end, a disjunct $A\{0,k\}B$, where $k > 0$, is represented internally as a *multiset element* $\langle A, B\rangle \mapsto k$ and the expression $(\langle A, B\rangle \mapsto m)|(\langle A, B\rangle \mapsto n)$ reduces to $(\langle A, B\rangle \mapsto \max(m, n))$. This is a form of *weak subsumption checking* that provides a crucial optimization step during

derivation. Similarly, when A and B are both singletons, say $\langle\alpha\rangle$ and $\langle\beta\rangle$, then $\langle\alpha\rangle|\langle\beta\rangle$ reduces to $\langle\alpha \vee_{\mathtt{BV}} \beta\rangle$ and $\langle\alpha\rangle\&\langle\beta\rangle$ reduces to $\langle\alpha \wedge_{\mathtt{BV}} \beta\rangle$. Here thread safety of the Boolean operations is important in a multi threaded application.

Finally, two more key optimizations are worth mentioning. First, during the main match generation loop, symbolic regex nodes are internalized into integer state ids and a DFA is maintained in form of an integer array δ indexed by $[i, q]$ where $1 \leq i \leq k$ is an atom index, and q is a state integer id, such that old state ids are immediately looked up as $\delta[i, q]$ and not rederived. Second, during step 2, initial search for the *relevant initial prefix*, when applicable, is performed using `string.IndexOf` to completely avoid the trivial initial state transition corresponding to the loop $\partial_c .*R = .*R$ in the case when $\partial_c R = \bot$.

4 Evaluation

We have evaluated the performance of SRM on two benchmarks:

Twain: 15 regexes matched against a 16 MB file containing the collected works of Mark Twain.

Assorted: 735 regexes matched against a synthetic input that includes some matches for each regex concatenated with random strings to produce an input file of 32 MB. The regexes are from the Automata library's samples and were originally collected from an online database of regular expressions.

We compare the performance of our matcher against the built-in .NET regex engine and Google's RE2 [3], a state of the art backtracking free regex match generation engine. RE2 is written in `C++` and internally based on automata. It eliminates bounded quantifiers in a preprocessing step by unwinding them, which may cause the regex to be rejected if the unwinding exceeds a certain limit. RE2 does not support extended operations over regexes such as intersection or complement. We use RE2 through a `C#` wrapper library.

The input to the built-in .NET regex engine and SRM is in UTF16, which is the encoding for NET's built-in strings, while RE2 is called with UTF8 encoded input. This implies for example that a regex such as `[\uD800-\uDFFF]` that tries to locate a single UTF16 surrogate is not meaningful in the context UTF8. All experiments were run on a machine with dual Intel Xeon E5-2620v3 CPUs running Windows 10 with .NET Framework 4.7.1. The reported running times for **Twain** are averages of 10 samples, while the statistics for **Assorted** are based on a single sample for each regex.

Figure 1 presents running times for each regex in **Twain**, while Fig. 2 presents a selection of metrics for the **Assorted** benchmark.

Both SRM and RE2 are faster than .NET on most regexes. This highlights the advantages of automata based regular expression matching when the richer features of a backtracking matcher are not required.

Compilation of regular expressions into matcher objects takes more time in SRM than RE2 or .NET. The largest contributor to this is finding the minterms of all predicates in the regex. For use cases where initialization time is critical

Regex	.NET	RE2	SRM
`Twain`	0.20s	**0.02s**	0.05s
`(?i)Twain`	0.66s	**0.26s**	0.39s
`[a-z]shing`	4.11s	**0.21s**	0.78s
`Huck[a-zA-Z]+|Saw[a-zA-Z]+`	1.47s	0.21s	**0.11s**
`[a-q][^u-z]{13}x`	10.20s	16.64s	**3.07s**
`Tom|Sawyer|Huckleberry|Finn`	1.53s	0.24s	**0.12s**
`(?i)Tom|Sawyer|Huckleberry|Finn`	6.73s	**0.22s**	0.51s
`.{0,2}(Tom|Sawyer|Huckleberry|Finn)`	15.84s	**0.22s**	0.64s
`.{2,4}(Tom|Sawyer|Huckleberry|Finn)`	16.15s	**0.22s**	0.62s
`Tom.{10,25}river|river.{10,25}Tom`	1.83s	**0.21s**	**0.21s**
`[a-zA-Z]+ing`	9.62s	**0.73s**	0.92s
`\s[a-zA-Z]{0,12}ing\s`	5.56s	**0.30s**	0.82s
`([A-Za-z]awyer|[A-Za-z]inn)\s`	6.26s	**0.21s**	0.87s
`["’][^"’]{0,30}[?!\.][\"’]`	2.00s	0.25s	**0.19s**
`\p{Sm}`	1.71s	0.21s	**0.10s**

Fig. 1. Time to generate all matches for each regex in **Twain**.

Metric	.NET	RE2	SRM
# of regexes with best time	0	305	**430**
Total compilation time for all regexes	0.8s	**0.1s**	13.9s
Average matching time	7.93s	0.36s	**0.25s**
80th percentile matching time	2.24s	**0.08s**	0.13s

Fig. 2. Metrics for the **Assorted** benchmark.

and inputs are known in advance, SRM provides support for pre-compilation and fast deserialization of matchers.

Comparing SRM to RE2 we can see that both matchers have regexes they do better on. While SRM achieves a lower average matching time on **Assorted**, this is due to the more severe outliers in RE2's performance profile, as shown by the lower 80th percentile matching time. Overall SRM offers performance that is comparable to RE2 while being implemented in `C#` without any unsafe code.

Application to Security Leak Scanning. SRM has been adopted in an internal tool at Microsoft that scans for credentials and other sensitive content in cloud service software. With the built-in .NET regex engine the tool was susceptible to catastrophic backtracking on files with long lines, such as minified JavaScript and SQL server seeding files. SRM's linear matching complexity has helped address these issues, while maintaining compatibility for the large set of .NET regexes used in the application.

References

1. Brzozowski, J.A.: Derivatives of regular expressions. JACM **11**, 481–494 (1964)
2. Fischer, S., Huch, F., Wilke, T.: A play on regular expressions: functional pearl. SIGPLAN Not. **45**(9), 357–368 (2010)
3. Google: RE2. https://github.com/google/re2

4. Hooimeijer, P., Veanes, M.: An evaluation of automata algorithms for string analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_18
5. Microsoft: Automata. https://github.com/AutomataDotNet/
6. Owens, S., Reppy, J., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009)
7. Traytel, D., Nipkow, T.: Verified decision procedures for MSO on words based on derivatives of regular expressions. SIGPLAN Not. **48**(9), 3–12 (2013)
8. Veanes, M., Bjørner, N.: Symbolic automata: the toolkit. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 472–477. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_33

COMPASS 3.0

Marco Bozzano[1(✉)], Harold Bruintjes[2], Alessandro Cimatti[1], Joost-Pieter Katoen[2], Thomas Noll[2], and Stefano Tonetta[1]

[1] Embedded Systems Unit, Fondazione Bruno Kessler, Trento, Italy
bozzano@fbk.eu

[2] Software Modeling and Verification Group, RWTH Aachen University, Aachen, Germany

Abstract. COMPASS (COrrectness, Modeling and Performance of AeroSpace Systems) is an international research effort aiming to ensure system-level correctness, safety, dependability and performability of on-board computer-based aerospace systems. In this paper we present COMPASS 3.0, which brings together the results of various development projects since the original inception of COMPASS. Improvements have been made both to the frontend, supporting an updated modeling language and user interface, as well as to the backend, by adding new functionalities and improving the existing ones. New features include Timed Failure Propagation Graphs, contract-based analysis, hierarchical fault tree generation, probabilistic analysis of non-deterministic models and statistical model checking.

1 Introduction

The COMPASS toolset provides an integrated model-based approach for System-Software Co-Engineering in the aerospace domain. It uses formal verification techniques, notably model checking, and originates from an ESA initiative dating back to 2008 [6]. Over the past eight years, various projects followed which extended the toolset. In a recent effort funded by ESA, the results of this work have been thoroughly consolidated into a single release, which is now available.

COMPASS 3.0 includes features originally included in distinct tool releases that diverged from the original development trunk[1]. The AUTOGEF and FAME projects focused on Fault Detection, Identification, and Recovery (FDIR) requirements modeling and development, and on fault propagation analysis; HASDEL extended formal analysis techniques to deal with the specific needs of launcher systems, with a strong focus on timed aspects of the model; and finally CATSY had the goal of improving the requirements specification process.

This paper presents an overview of the toolset, as well as a description of the enhancements made since its last official release in 2013 (v2.3). For a more detailed description of the (pre-)existing features and capabilities, we refer to [5,6].

[1] See www.compass-toolset.org.

This work has been funded by the European Space Agency (ESA-ESTEC) under contract 4000115870/15/NL/FE/as.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 379–385, 2019.
https://doi.org/10.1007/978-3-030-17462-0_25

2 Toolset Overview

The COMPASS toolset can be divided into the user facing side (the frontend), and the verification engines used (the backend). The frontend provides a GUI that offers access to all the analysis functions of the toolset, as well as command-line scripts. The backend tools are chosen and invoked by the toolset automatically.

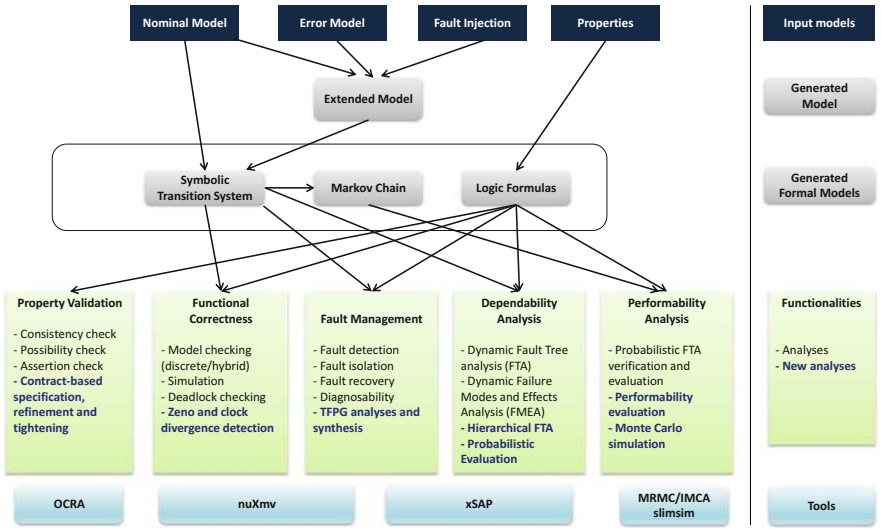


Fig. 1. Overview of the COMPASS toolset

The functionalities of COMPASS are summarized in Fig. 1, where arrows represent I/O relationships for model transformations, and link models with the corresponding analyses. User inputs are models written in a dialect of AADL [18] and properties. COMPASS is based on the concept of *model extension*, i.e., the possibility to automatically inject faults into a nominal model, by specifying error models and a set of fault injections. The extended model is internally converted into a symbolic model amenable to formal verification and to a Markov chain (for probabilistic analyses). Properties are automatically translated into temporal logic formulas. Given the models and the properties, COMPASS provides a full set of functionalities, including property validation, functional correctness, fault management, dependability and performability analyses.

The analyses are supported by the following verification engines: nuXmv [9] for correctness checking; OCRA [10] for contract-based analysis; IMCA [19] and MRMC [20] for performance analysis by probabilistic model checking; slimsim [8] for statistical model checking and xSAP [1] for safety analysis.

COMPASS is licensed under a variant of the GPL license, restricted to ESA member states. It is distributed on a pre-configured and ready-to-use virtual machine, or as two different code bundles.

In a typical workflow, one would start with both a nominal and an error specification of the system and then use simulation to interactively explore its dynamic behavior. In a next step, verification of functional correctness based on user-defined properties can be performed, followed by more specialized analyses as indicated in Fig. 1. For a complete overview of the COMPASS toolset, we refer to the COMPASS tutorial [15] and user manual [16].

3 Input Language

Input models for COMPASS use a variant of the AADL language [18], named SLIM. AADL is standardized and used in e.g., the aerospace and automotive industries. SLIM provides extensions for behavior and error specification. A model is described in terms of components, which may specify subcomponents, forming a component hierarchy. Components interact with each other by means of ports, which send either discrete events or data values. Components may furthermore specify modes, which may render subcomponents active or inactive, thus enabling system reconfiguration. Error behavior is specified in terms of error components, which define error states, (probabilistic) events and propagations which may trigger a synchronization between error components. The impact of faults occurring in the error model onto the nominal model is described by means of *fault injections*, which specify the fault effect by update operations on data components. This has been extended by supporting specifications that can inhibit certain events from occurring, or specifying a set of modes that is available in a certain failure state. The language's semantics and syntax are described in [14].

Language Updates. COMPASS 3.0 supports the property system used by AADL. This makes it possible to annotate various elements in the model by using SLIM specific attributes, and makes the language more compatible with the core AADL specification, improving interoperability. New features also include timed error models (that is, error components may contain clocks), non-blocking ports and separation of configuration and behavior. The latter entails that composite components can only specify modes and cannot change data values or generate events by means of transitions, whereas atomic components may specify states.

Property Specification. Properties can be specified in three different ways. The first two options are simpler to use, since they hide the details of the underlying temporal logic, but less expressive. *Design attributes* [3] represent a specific property of a model's element, such as the delay of an event or the invariant of a mode. They are directly associated to model elements. Formal properties are automatically derived based on the Catalogue of System and Software Properties (CSSP) [3]. The *pattern-based* system uses pre-defined patterns with placeholders to define formal properties. Time bounds and probabilities can optionally be specified. As a last option, the user can encode properties directly using *logical expressions*. This enables the modeler to exploit the full power of the underlying temporal logics, and offers the highest expressivity.

4 New Functionalities in COMPASS 3.0

In this section, we discuss the main functionalities of the COMPASS 3.0 toolset.

Correctness Checking. COMPASS supports checking for correctness of the model by providing properties. The toolset indicates for each property whether it holds or not, and gives a counter example in the latter case. Verification relies on edge technologies based on BDD- and SAT-based model checking, including *K-liveness verification* [12]. In order to assist the user in the specification of timed models, COMPASS 3.0 offers functionality to check the timed correctness of the model w.r.t. *Zenoness* and *clock divergence*. The former is caused by cycles in the system's state space that do not require progressing of time. The latter is caused by clocks that can attain an arbitrarily large value. The toolset can automatically check Zenoness for all modes in the system, and divergence for all clocks.

Contract-Based Analysis. COMPASS 3.0 offers the possibility to perform *contract-based analysis* [11]. Contracts must be specified in the model and attached to components. Each contract consists of an *assumption* (a property of the environment of the component) and a *guarantee* (a property of the implementation of the component, which must hold as long as the assumption holds). In order to perform compositional analysis, a contract refinement must be further specified, which links a contract to a set of contracts of the subcomponents. COMPASS 3.0 supports the following analyses. *Validation* is performed on assumptions and guarantees. The user can choose a subset of these properties and check consistency or entailment. *Refinement checking* verifies whether the contract refinements are correct. Namely, that whenever the implementations of the subcomponents satisfy their contracts and the environment satisfies its assumption, then the guarantee of the supercomponent and the assumptions of its subcomponents are satisfied. Finally, *tightening* looks for a weakening and/or strengthening of the assumptions/guarantees, respectively, such that the refinement still holds.

Fault Trees. COMPASS 3.0 can generate fault trees associated with particular error states in the model. Standard fault trees are flat in nature (being two- or three-leveled), hiding some of the nested dependencies. Contract-based analysis can be used to generate a *hierarchical fault tree*, which captures the hierarchy of the model. This approach makes use of the specified contracts, and checks which events may cause them to be invalidated. COMPASS 3.0 offers further alternatives to analyze fault trees. Static probabilities can be calculated for the entire tree by specifying the probabilities of basic events. Fault Tree Evaluation calculates the probability of failure for a given time span. Finally, Fault Tree Verification checks a probabilistic property specified for the fault tree.

Performability. COMPASS 3.0 offers two model checking approaches to probabilistic analysis (which, using a probabilistic property, determine the

probability of failure within a time period): using numerical analysis or using Monte-Carlo simulation. The former originally only supported Continuous Time Markov Chains (CTMCs) using the MRMC [20] tool. This has now been extended to Interactive Markov Chains (IMCs) using IMCA [19], which makes it possible to analyze continuous-time stochastic models which exhibit *non-determinism*. However, neither approach supports hybrid models containing clocks. For the analysis of these models, statistical model checking techniques [7,8] are employed, which use Monte-Carlo simulation to determine, within a certain margin of likelihood and error, the probability of quantitative properties.

Timed Failure Propagation Graphs. Timed Failure Propagation Graphs (TFPGs) [2] support various aspects of diagnosis and prognosis, such as modeling the temporal dependency between the occurrence of events and their dependence on system modes. A TFPG is a labeled directed graph where nodes represent either fault modes or discrepancies, which are off-nominal conditions that are effects of fault modes. COMPASS 3.0 supports three kinds of analyses based on TFPGs: *synthesis*, where a TFPG is automatically derived from the model, *behavioral validation*, which checks whether a given TFPG is complete (i.e., a faithful abstraction) w.r.t. the model; and *effectiveness validation*, which checks whether the TFPG is sufficiently accurate for allowing diagnosability of failures.

5 Related Work and Conclusion

Closely related to COMPASS is the TASTE toolset, dedicated to the development of embedded, real-time systems. It focuses on the integration of heterogeneous technologies for modeling and verification (including AADL), code generation and integration (e.g., written in C and ADA) and deployment. Another ESA initiative is COrDeT-2, part of which defines OSRA (On-board Software Reference Architecture), which aims to improve software reuse by defining a standardized architecture. Security extensions in COMPASS 3.0 have been added as part of the D-MILS project [21], enabling reasoning on data security.

Various case studies have been performed using COMPASS 3.0. The first one was targeting at the Preliminary Design Review stage of a satellite's design [17]. The study lasted for six months and encompassed a model of about 90 components. A second case study followed during the Critical Design Review stage, focusing on modeling practices and diagnosability [4], with a scale twice the size of [17]. A smaller scale case study was later performed as part of the HASDEL project [8]. Recently, the CubETH nano-satellite was represented as a model with 82 components and analyzed using COMPASS 3.0 [7].

The case studies demonstrate that the key benefit of the COMPASS approach is the culmination of a single comprehensive system model that covers all aspects (discrete, real-time, hybrid, probabilistic). This ensures consistency of the analyses, which is a major benefit upon current practices where various (tailored) models are constructed each covering different aspects. For further directions,

we refer to the COMPASS roadmap [13], which thoroughly discusses goals for the toolset as well as the development process, research directions, community outreach and further integration with other ESA initiatives.

References

1. Bittner, B., et al.: The xSAP safety analysis platform. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 533–539. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_31
2. Bittner, B., Bozzano, M., Cimatti, A.: Automated synthesis of timed failure propagation graphs. In: Proceedings of IJCAI, pp. 972–978 (2016)
3. Bos, V., Bruintjes, H., Tonetta, S.: Catalogue of system and software properties. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) SAFECOMP 2016. LNCS, vol. 9922, pp. 88–101. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45477-1_8
4. Bozzano, M., et al.: Spacecraft early design validation using formal methods. Reliab. Eng. Syst. Saf. **132**, 20–35 (2014)
5. Bozzano, M., Cimatti, A., Katoen, J.P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability and performance analysis of extended AADL models. Comput. J. **54**(5), 754–775 (2011)
6. Bozzano, M., et al.: A model checker for AADL. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 562–565. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_48
7. Bruintjes, H.: Model-based reliability analysis of aerospace systems. Ph.D. thesis, RWTH Aachen University (2018)
8. Bruintjes, H., Katoen, J.P., Lesens, D.: A statistical approach for timed reachability in AADL models. In: Proceedings of DSN, pp. 81–88. IEEE (2015)
9. Cavada, R., et al.: The nuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
10. Cimatti, A., Dorigatti, M., Tonetta, S.: OCRA: a tool for checking the refinement of temporal contracts. In: Proceedings of ASE, pp. 702–705 (2013)
11. Cimatti, A., Tonetta, S.: Contracts-refinement proof system for component-based embedded systems. Sci. Comput. Program. **97**, 333–348 (2015)
12. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Proceedings of FMCAD, pp. 52–59 (2012)
13. COMPASS Consortium: COMPASS roadmap. Technical report (2016). http://www.compass-toolset.org/docs/compass-roadmap.pdf
14. COMPASS Consortium: SLIM 3.0 - syntax and semantics. Technical report (2016). http://www.compass-toolset.org/docs/slim-specification.pdf
15. COMPASS Consortium: COMPASS tutorial - version 3.0.1. Technical report (2018). http://www.compass-toolset.org/docs/compass-tutorial.pdf
16. COMPASS Consortium: COMPASS user manual - version 3.0.1. Technical report (2018). http://www.compass-toolset.org/docs/compass-manual.pdf
17. Esteve, M.A., Katoen, J.P., Nguyen, V.Y., Postma, B., Yushtein, Y.: Formal correctness, safety, dependability and performance analysis of a satellite. In: Proceeding of ICSE, pp. 1022–1031. ACM and IEEE (2012)
18. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley, Upper Saddle River (2012)

19. Guck, D., Han, T., Katoen, J.-P., Neuhäußer, M.R.: Quantitative timed analysis of interactive Markov Chains. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 8–23. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28891-3_4
20. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. Perform. Eval. **68**(2), 90–104 (2011)
21. van der Pol, K., Noll, T.: Security type checking for MILS-AADL specifications. In: International MILS Workshop. Zenodo (2015). http://mils-workshop-2015.euromils.eu/

Debugging of Behavioural Models with CLEAR

Gianluca Barbon[1], Vincent Leroy[2], and Gwen Salaün[1(✉)]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
gwen.salaun@inria.fr
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. This paper presents a tool for debugging behavioural models being analysed using model checking techniques. It consists of three parts: (i) one for annotating a behavioural model given a temporal formula, (ii) one for visualizing the erroneous part of the model with a specific focus on decision points that make the model to be correct or incorrect, and (iii) one for abstracting counterexamples thus providing an explanation of the source of the bug.

1 Introduction

Model checking [2] is an established technique for automatically verifying that a behavioural model satisfies a given temporal property, which specifies some expected requirement of the system. In this work, we use Labelled Transition Systems (LTS) as behavioural models of concurrent programs. An LTS consists of states and labelled transitions connecting these states. An LTS can be produced from a higher-level specification of the system described with a process algebra for instance. Temporal properties are usually divided into two main families: safety and liveness properties [2]. Both are supported in this work. If the LTS does not satisfy the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied.

Understanding this counterexample for debugging the specification is a complicated task for several reasons: (i) the counterexample may consist of many actions; (ii) the debugging task is mostly achieved manually (satisfactory automatic debugging techniques do not yet exist); (iii) the counterexample does not explicitly point out the source of the bug that is hidden in the model; (iv) the most relevant actions are not highlighted in the counterexample; (v) the counterexample does not give a global view of the problem.

The CLEAR tools (Fig. 1) aims at simplifying the debugging of concurrent systems whose specification compiles into a behavioural model. To do so, we propose a novel approach for improving the comprehension of counterexamples by highlighting some of the states in the counterexample that are of prime importance because from those states the specification can reach a correct part of the model or an incorrect one. These states correspond to decisions or choices that are particularly interesting because they usually provide an explanation of the

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 386–392, 2019.
https://doi.org/10.1007/978-3-030-17462-0_26

source of the bug. The first component of the CLEAR toolset computes these specific states from a given LTS (AUT format) and a temporal property (MCL logic [5]). Second, visualization techniques are provided in order to graphically observe the whole model and see how those states are distributed over that model. Third, explanations of the bug are built by abstracting away irrelevant parts of the counterexample, which results in a simplified counterexample.

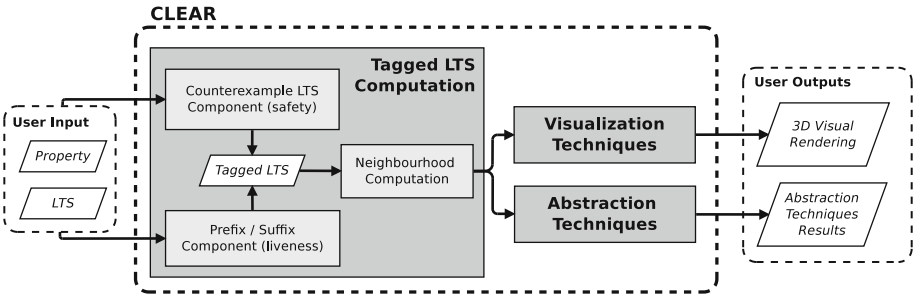


Fig. 1. Overview of the CLEAR toolset.

The CLEAR toolset has been developed mainly in Java and consists of more than 10K lines of code. All source files and several case studies are available online [1]. CLEAR has been applied to many examples and the results turn out to be quite positive as presented in an empirical evaluation which is also available online.

The rest of this paper is organised as follows. Section 2 overviews the LTS and property manipulations in order to compute annotated or tagged LTSs. Sections 3 and 4 present successively our techniques for visualizing tagged models and for abstracting counterexamples with the final objective in both cases to simplify the debugging steps. Section 5 describes experiments we carried out for validating our approach on case studies. Section 6 concludes the paper.

2 Tagged LTSs

The first step of our approach is to identify in the LTS parts of it corresponding to correct or incorrect behaviours. This is achieved using several algorithms that we define and that are presented in [3,4]. We use different techniques depending on the property family. As far as safety properties are concerned, we compute an LTS consisting of all counterexamples and compare it with the full LTS. As for liveness properties, for each state, we compute the set of prefixes and suffixes. Then, we use this information for tagging transitions as correct, incorrect or neutral in the full LTS. A correct transition leads to a behaviour that always satisfies the property, while an incorrect one leads to a behaviour that always violates the property. A neutral transition is common to correct and incorrect behaviours.

Once we have this information about transitions, we can identify specific states in the LTS where there is a choice in the LTS that directly affects the compliance with the property. We call these states and the transitions incoming to/outgoing from those states *neighbourhoods*.

There are four kinds of neighbourhoods, which differ by looking at their outgoing transitions (Fig. 2 from left to right): (1) with at least one correct transition (and no incorrect transition), (2) with at least one incorrect transition (and no correct transition), (3) with at least one correct transition and one incorrect transition, but no neutral transition, (4) with at least one correct transition, one incorrect transition and one neutral transition. The transitions contained in neighbourhood of type (1) highlight a choice that can lead to behaviours that always satisfy the property. Note that neighbourhoods with only correct outgoing transitions are not possible, since they would not correspond to a problematic choice. Consequently, this type of neighbourhood always presents at least one outgoing neutral transition. The transitions contained in neighbourhood of type (2), (3) or (4) highlight a choice that can lead to behaviours that always violate the property.

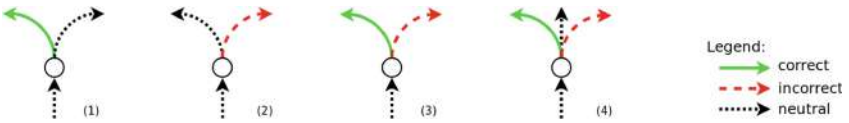


Fig. 2. The four types of neighbourhoods. (Color figure online)

It is worth noting that both visualization and counterexample abstraction techniques share the computation of the tagged LTS (correct/incorrect/neutral transitions) and of the neighbourhoods.

3 Visualization Techniques

The CLEAR visualizer provides support for visualizing the erroneous part of the LTS and emphasizes all the states (a.k.a. neighbourhoods) where a choice makes the specification either head to correct or incorrect behaviour. This visualization is very useful from a debugging perspective to have a global point of view and not only to focus on a specific erroneous trace (that is, a counterexample).

More precisely, the CLEAR visualizer supports the visualization of tagged LTSs enriched with neighbourhoods. These techniques have been developed using Javascript, the AngularJS framework, the bootstrap CSS framework, and the 3D force graph library. These 3D visualization techniques make use of different colors to distinguish correct (green), incorrect (red) and neutral (black) transitions on the one hand, and all kinds of neighbourhoods (represented with different shades of yellow) on the other hand. The tool also provides several functionalities in order to explore tagged LTSs for debugging purposes, the main one

being the step-by-step animation starting from the initial state or from any chosen state in the LTS. This animation keeps track of the already traversed states/transitions and it is possible to move backward in that trace. Beyond visualizing the whole erroneous LTS, another functionality allows one to focus on one specific counterexample and rely on the animation features introduced beforehand for exploring the details of that counterexample (correct/incorrect transitions and neighbourhoods).

Figure 3 gives a screenshot of the CLEAR visualizer. The legend on the left hand side of this figure depicts the different elements and colors used in the LTS visualization. All functionalities appear in the bottom part. When the LTS is loaded, one can also load a counterexample. On the right hand side, there is the name of the file and the list of states/transitions of the current animation. Note that transitions labels are not shown, they are only displayed through mouseover. This choice allows the tool to provide a clearer view of the LTS.

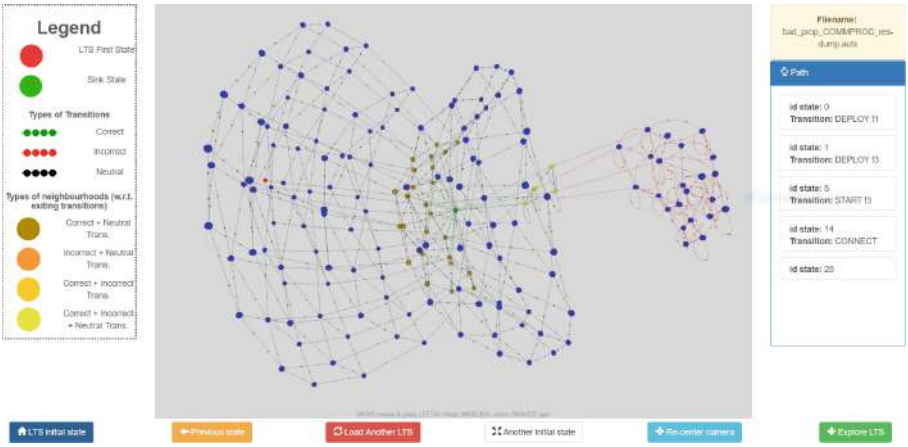


Fig. 3. Screenshot of the CLEAR visualizer. (Color figure online)

From a methodological point of view, it is adviced to use first the CLEAR visualizer during the debugging process for taking a global look at the erroneous part of the LTS and possibly notice interesting structures in that LTS that may guide the developer to specific kinds of bug. Step-by-step animation is also helpful for focusing on specific traces and for looking more carefully at some transitions and neighbourhoods on those traces. If the developer does not identify the bug using these visualization techniques, (s)he can make use of the CLEAR abstraction techniques presented in the next section.

4 Abstraction Techniques

In this section, once the LTS has been tagged using algorithms overviewed in Sect. 2, the developer can use abstraction techniques that aim at simplifying a

counterexample produced from the LTS and a given property. To do so we make a joint analysis of the counterexample and of the LTS enriched with neighbourhoods computed previously. This analysis can be used for obtaining different kinds of simplifications, such as: (i) an abstracted counterexample, that allows one to remove from a counterexample actions that do not belong to neighbourhoods (and thus represent noise); (ii) a shortest path to a neighbourhood, which retrieves the shortest sequence of actions that leads to a neighbourhood; (iii) improved versions of (i) and (ii), where the developer provides a pattern representing a sequence of non-contiguous actions, in order to allow the developer to focus on a specific part of the model; (iv) techniques focusing on a notion of distance to the bug in terms of neighbourhoods. For the sake of space, we focus on the abstracted counterexample in this paper.

Abstracted Counterexample. This technique takes as input an LTS where neighbourhoods have been identified and a counterexample. Then, it removes all the actions in the counterexample that do not represent incoming or outgoing transitions of neighbourhoods. Figure 4 shows an example of a counterexample where two neighbourhoods, highlighted on the right side, have been detected and allow us to identify actions that are preserved in the abstracted counterexample.

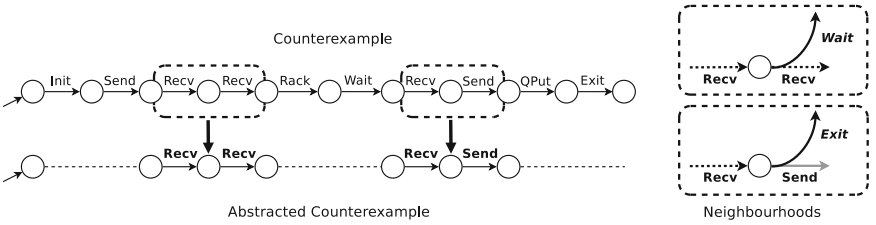


Fig. 4. Abstracted counterexample.

5 Experiments

We carried out experiments on about 100 examples. For each one, we use as input a process algebraic specification that was compiled into an LTS model, and a temporal property. As far as computation time is concerned, the time is quite low for small examples (a few seconds), while it tends to increase w.r.t. the size of the LTS when we deal with examples with hundreds of thousands of transitions and states (a few minutes). In this case, it is mainly due to the computation of tagged LTSs, which is quite costly because it is based on several graph traversals. Visualization techniques allowed us to identify several examples of typical bugs with their corresponding visual models. This showed that the visualizations exhibit specific structures that characterize the bug and are helpful for supporting the developer during his/her debugging tasks. As for abstraction techniques, we observed some clear gain in length (up to 90%) between the original counterexample and the abstracted one, which keeps only relevant actions using our approach and thus facilitates the debugging task for the developer.

We also carried out an empirical study to validate our approach. We asked 17 developers, with different degrees of expertise, to find bugs on two test cases by taking advantage of the abstracted counterexample techniques. The developers were divided in two groups, in order to evaluate both test cases with and without the abstracted counterexample. The developers were asked to discover the bug and measure the total time spent in debugging each test case. We measured the results in terms of time, comparing for both test cases the time spent with and without the abstracted counterexample. We observed a gain of about 25% of the total average time spent in finding the bug for the group using our approach. We finally asked developers' opinion about the benefit given by our method in detecting the bug. Most of them agreed considering our approach helpful.

The CLEAR toolset is available online [1] jointly with several case studies and the detailed results of the empirical study.

6 Concluding Remarks

In this paper, we have presented the CLEAR toolset for simplifying the comprehension of erroneous behavioural specifications under validation using model checking techniques. To do so, we are able to detect the choices in the model (neighbourhood) that may lead to a correct or incorrect behaviour, and generate a tagged LTS as result. The CLEAR visualizer takes as input a tagged LTS and provides visualization techniques of the whole erroneous part of the model as well as animation techniques that help the developer to navigate in the model for better understanding what is going on and hopefully detect the source of the bug. The counterexample abstraction techniques are finally helpful for building abstractions from counterexamples by keeping only relevant actions from a debugging perspective. The experiments we carried out show that our approach is useful in practice to help the designer in finding the source of the bug(s).

References

1. CLEAR Debugging Tool. https://github.com/gbarbon/clear/
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Barbon, G., Leroy, V., Salaün, G.: Debugging of concurrent systems using counterexample analysis. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 20–34. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_2
4. Barbon, G., Leroy, V., Salaün, G.: Counterexample simplification for liveness property violation. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 173–188. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_11
5. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_12

Machine Learning

Omega-Regular Objectives in Model-Free Reinforcement Learning

Ernst Moritz Hahn[1,2], Mateo Perez[3], Sven Schewe[4], Fabio Somenzi[3], Ashutosh Trivedi[5](✉), and Dominik Wojtczak[4]

[1] School of EEECS, Queen's University Belfast, Belfast, UK
[2] State Key Laboratory of Computer Science, Institute of Software, CAS, Beijing, People's Republic of China
[3] Department of ECEE, University of Colorado Boulder, Boulder, USA
[4] Department of Computer Science, University of Liverpool, Liverpool, UK
[5] Department of Computer Science, University of Colorado Boulder, Boulder, USA
ashutosh.trivedi@colorado.edu

Abstract. We provide the first solution for model-free reinforcement learning of ω-regular objectives for Markov decision processes (MDPs). We present a constructive reduction from the almost-sure satisfaction of ω-regular objectives to an almost-sure reachability problem, and extend this technique to learning how to control an unknown model so that the chance of satisfying the objective is maximized. We compile ω-regular properties into limit-deterministic Büchi automata instead of the traditional Rabin automata; this choice sidesteps difficulties that have marred previous proposals. Our approach allows us to apply model-free, off-the-shelf reinforcement learning algorithms to compute optimal strategies from the observations of the MDP. We present an experimental evaluation of our technique on benchmark learning problems.

1 Introduction

Reinforcement learning (RL) [3,37,40] is an approach to sequential decision making in which agents rely on reward signals to choose actions aimed at achieving prescribed objectives. Model-free RL refers to a class of techniques that are asymptotically space-efficient [36] because they do not construct a full model of the environment. These techniques include classic algorithms like Q-learning [37] as well as their extensions to deep neural networks [14,31]. Some objectives, like running a maze, are naturally expressed in terms of scalar rewards; in other cases the translation is less obvious. We solve the problem of *ω-regular rewards*, that

This work is supported in part by the Marie Skłodowska Curie Fellowship *Parametrised Verification and Control*, NSFC grants 61761136011 and 61532019, an ASIRT grant by the College of Engineering and Applied Sciences of CU Boulder, and EPSRC grants EP/M027287/1 and EP/P020909/1.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 395–412, 2019.
https://doi.org/10.1007/978-3-030-17462-0_27

is, the problem of defining scalar rewards for the transitions of a Markov decision process (MDP) so that strategies that maximize the probability to satisfy an ω-regular objective may be computed by off-the-shelf, *model-free* RL algorithms.

Omega-regular languages [28,38] provide a rich formalism to unambiguously express qualitative safety and progress requirements of MDPs [2]. A common way to describe an ω-regular language is via a formula in Linear Time Logic (LTL); other specification mechanisms include extensions of LTL, various types of automata, and monadic second-order logic. A typical requirement that is naturally expressed as an ω-regular objective prescribes that the agent should eventually control the MDP to stay within a given set of states, while at all times avoiding another set of states. In LTL this would be written $(\mathsf{F}\,\mathsf{G}\,\texttt{goal}) \wedge (\mathsf{G}\,\neg\texttt{trap})$, where `goal` and `trap` are labels attached to the appropriate states, F stands for "finally," and G stands for "globally."

For verification or synthesis, an ω-regular objective is usually translated into an automaton that monitors the traces of the MDP [10]. Successful executions cause the automaton to take certain (accepting) transitions infinitely often, and ultimately avoid certain (rejecting) transitions. That is, ω-regular objectives are about the long-term behavior of an MDP; the frequency of reward collected is not what matters. A policy that guarantees no rejecting transitions and an accepting transition every ten steps, is better than a policy that promises an accepting transition at each step, but with probability 0.5 does not accept at all.

The problem of ω-regular rewards in the context of model-free RL was first tackled in 2014 by translating the objective into a deterministic Rabin automaton and deriving positive and negative rewards directly from the acceptance condition of the automaton [32]. In Sect. 3 we show that their algorithm, and the extension of [18] may fail to find optimal strategies, and may underestimate the probability of satisfaction of the objective. In [16,17] the use of limit-deterministic Büchi automata avoids the problems connected with the use of a Rabin acceptance condition. However, as shown in Sect. 3, that approach may still produce incorrect results.

We avoid the problems inherent in the use of deterministic Rabin automata for model-free RL by resorting to *limit-deterministic* Büchi automata, which, under mild restrictions, were shown by [8,15,33] to be suitable for both qualitative and quantitative analysis of MDPs under all ω-regular objectives. The Büchi acceptance condition, which, unlike the Rabin condition, does not use rejecting transitions, allows us to constructively reduce the almost-sure satisfaction of ω-regular objectives to an almost-sure reachability problem. It is also suitable for quantitative analysis: the value of a state converges to the maximum probability of satisfaction of the objective from that state as a parameter approaches 1.

We concentrate on model-free approaches and infinitary behaviors for finite MDPs. Related problems include model-based RL [13], RL for finite-horizon objectives [22,23], and learning for efficient verification [4].

This paper is organized as follows. Section 2 recalls definitions and notations. Section 3 shows the problems that arise when the reward of the RL algorithm is derived from the acceptance condition of a deterministic Rabin automaton. In Sect. 4 we prove the main results. Finally, Sect. 5 discusses our experiments.

2 Preliminaries

2.1 Markov Decision Processes

Let $\mathcal{D}(S)$ be the set of distributions over S. A *Markov decision process* $\mathcal{M}$ is a tuple (S, A, T, AP, L) where S is a finite set of states, A is a finite set of *actions*, $T\colon S \times A \nrightarrow \mathcal{D}(S)$ is the probabilistic transition (partial) function, AP is the set of *atomic propositions*, and $L\colon S \to 2^{AP}$ is the *proposition labeling function*.

For any state $s \in S$, we let $A(s)$ denote the set of actions that can be selected in state s. For states $s, s' \in S$ and $a \in A(s)$, $T(s,a)(s')$ equals $p(s'|s,a)$. A *run* of $\mathcal{M}$ is an ω-word $\langle s_0, a_1, s_1, \ldots\rangle \in S \times (A \times S)^\omega$ such that $p(s_{i+1}|s_i, a_{i+1})>0$ for all $i \geq 0$. A finite run is a finite such sequence. For a *run* $r = \langle s_0, a_1, s_1, \ldots\rangle$ we define the corresponding labeled run as $L(r) = \langle L(s_0), L(s_1), \ldots\rangle \in (2^{AP})^\omega$. We write $Runs^{\mathcal{M}}(FRuns^{\mathcal{M}})$ for the set of runs (finite runs) of the MDP $\mathcal{M}$ and $Runs^{\mathcal{M}}(s)(FRuns^{\mathcal{M}}(s))$ for the set of runs (finite runs) of the MDP $\mathcal{M}$ starting from state s. We write $last(r)$ for the last state of a finite run r.

A strategy in $\mathcal{M}$ is a function $\sigma\colon FRuns \to \mathcal{D}(A)$ such that $supp(\sigma(r)) \subseteq A(last(r))$, where $supp(d)$ denotes the support of the distribution d. Let $Runs_\sigma^{\mathcal{M}}(s)$ denote the subset of runs $Runs^{\mathcal{M}}(s)$ that correspond to strategy σ with initial state s. Let $\Sigma_{\mathcal{M}}$ be the set of all strategies. A strategy σ is *pure* if $\sigma(r)$ is a point distribution for all runs $r \in FRuns^{\mathcal{M}}$ and we say that σ is *stationary* if $last(r) = last(r')$ implies $\sigma(r) = \sigma(r')$ for all runs $r, r' \in FRuns^{\mathcal{M}}$. A strategy that is not pure is *mixed*. A strategy is *positional* if it is both pure and stationary.

The behavior of an MDP $\mathcal{M}$ under a strategy σ is defined on a probability space $(Runs_\sigma^{\mathcal{M}}(s), \mathcal{F}_{Runs_\sigma^{\mathcal{M}}(s)}, \Pr_\sigma^{\mathcal{M}}(s))$ over the set of infinite runs of σ with starting state s. Given a real-valued random variable over the set of infinite runs $f\colon Runs^{\mathcal{M}} \to \mathbb{R}$, we denote by $\mathbb{E}_\sigma^{\mathcal{M}}(s)\{f\}$ the expectation of f over the runs of $\mathcal{M}$ originating at s that follow strategy σ.

Given an MDP $\mathcal{M} = (S, A, T, AP, L)$, we define its directed underlying graph $\mathcal{G}_{\mathcal{M}} = (V, E)$ where $V = S$ and $E \subseteq S \times S$ is such that $(s, s') \in E$ if $T(s,a)(s') > 0$ for some $a \in A(s)$. A sub-MDP of $\mathcal{M}$ is an MDP $\mathcal{M}' = (S', A', T', AP, L')$, where $S' \subset S$, $A' \subseteq A$ is such that $A'(s) \subseteq A(s)$ for every $s \in S'$, and T' and L' are analogous to T and L when restricted to S' and A'. In particular, $\mathcal{M}'$ is closed under probabilistic transitions, i.e. for all $s \in S'$ and $a \in A'$ we have that $T(s,a)(s') > 0$ implies that $s' \in S'$. An *end-component* [10] of an MDP $\mathcal{M}$ is a sub-MDP $\mathcal{M}'$ of $\mathcal{M}$ such that $\mathcal{G}_{\mathcal{M}'}$ is strongly connected.

Theorem 1 (End-Component Properties [10]). *Once an end-component C of an MDP is entered, there is a strategy that visits every state-action combination in C with probability 1 and stays in C forever. Moreover, for every strategy the union of the end-components is visited with probability 1.*

A Markov chain is an MDP whose set of actions is singleton. A *bottom strongly connected component* (BSCC) of a Markov chain is any of its end-components. A BSCC is accepting if it contains an accepting transition (see below) and otherwise it is rejecting. For any MDP $\mathcal{M}$ and positional strategy σ, let $\mathcal{M}_\sigma$ be the Markov chain resulting from resolving the nondeterminism in $\mathcal{M}$ using σ.

A *rewardful* MDP is a pair $(\mathcal{M}, \rho)$, where $\mathcal{M}$ is an MDP and $\rho\colon S \times A \to \mathbb{R}$ is a reward function assigning utility to state-action pairs. A rewardful MDP $(\mathcal{M}, \rho)$ under a strategy σ determines a sequence of random rewards $\rho(X_{i-1}, Y_i)_{i\geq 1}$, where X_i and Y_i are the random variables denoting the i-th state and action, respectively. Depending upon the problem of interest, different performance objectives may be of interest. The *reachability probability* objective $Reach(T)^{\mathcal{M}}_{\sigma}(s)$ (with $T \subseteq S$) is defined as $\Pr^{\mathcal{M}}_{\sigma}(s)\left\{\langle s, a_1, s_1, \ldots\rangle \in Runs^{\mathcal{M}}_{\sigma}(s)\colon \exists i\,.\, s_i \in T\right\}$. For a given discount factor $\lambda \in [0,1[$, the *discounted reward* objective $Disct(\lambda)^{\mathcal{M}}_{\sigma}(s)$ is defined as $\lim_{N\to\infty} \mathbb{E}^{\mathcal{M}}_{\sigma}(s)\left\{\sum_{1\leq i\leq N} \lambda^{i-1}\rho(X_{i-1}, Y_i)\right\}$, while the *average reward* $Avg^{\mathcal{M}}_{\sigma}(s)$ is defined as $\limsup_{N\to\infty}(1/N)\mathbb{E}^{\mathcal{M}}_{\sigma}(s)\left\{\sum_{1\leq i\leq N} \rho(X_{i-1}, Y_i)\right\}$. For an objective $Reward^{\mathcal{M}} \in \{Reach(T)^{\mathcal{M}}, Disct(\lambda)^{\mathcal{M}}, Avg^{\mathcal{M}}\}$ and an initial state s, we define the optimal reward $Reward^{\mathcal{M}}_{*}(s)$ as $\sup_{\sigma\in\Sigma_{\mathcal{M}}} Reward^{\mathcal{M}}_{\sigma}(s)$. A strategy $\sigma \in \Sigma_{\mathcal{M}}$ is optimal for $Reward^{\mathcal{M}}$ if $Reward^{\mathcal{M}}_{\sigma}(s) = Reward^{\mathcal{M}}_{*}(s)$ for all $s \in S$.

2.2 ω-Regular Performance Objectives

A *nondeterministic ω-automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \mathsf{Acc})$, where Σ is a finite *alphabet*, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta\colon Q \times \Sigma \to 2^Q$ is the *transition function*, and Acc is the *acceptance condition*. A *run* r of $\mathcal{A}$ on $w \in \Sigma^\omega$ is an ω-word $r_0, w_0, r_1, w_1, \ldots$ in $(Q \cup \Sigma)^\omega$ such that $r_0 = q_0$ and, for $i > 0$, $r_i \in \delta(r_{i-1}, w_{i-1})$. Each triple (r_{i-1}, w_{i-1}, r_i) is a *transition* of $\mathcal{A}$.

We consider Büchi and Rabin acceptance conditions, which depend on the transitions that occur infinitely often in a run of an automaton. We write $\mathsf{inf}(r)$ for the set of transitions that appear infinitely often in the run r. The *Büchi* acceptance condition defined by $F \subseteq Q \times \Sigma \times Q$ is the set of runs $\{r \in (Q \cup \Sigma)^\omega\colon \mathsf{inf}(r) \cap F \neq \emptyset\}$. A *Rabin* acceptance condition is defined in terms of k pairs of subsets of $Q \times \Sigma \times Q$, $(B_0, G_0), \ldots, (B_{k-1}, G_{k-1})$, as the set $\{r \in (Q \cup \Sigma)^\omega\colon \exists i < k\,.\, \mathsf{inf}(r) \cap B_i = \emptyset \wedge \mathsf{inf}(r) \cap G_i \neq \emptyset\}$. The *index* of a Rabin condition is its number of pairs.

A run r of $\mathcal{A}$ is *accepting* if $r \in \mathsf{Acc}$. The *language*, $L_{\mathcal{A}}$, of $\mathcal{A}$ (or, *accepted* by $\mathcal{A}$) is the subset of words in Σ^ω that have accepting runs in $\mathcal{A}$. A language is *ω-regular* if it is accepted by an ω-automaton.

Given an MDP $\mathcal{M}$ and an ω-regular objective φ given as an ω-automaton $\mathcal{A}_\varphi = (\Sigma, Q, q_0, \delta, \mathsf{Acc})$, we are interested in computing an optimal strategy satisfying the objective. We define the satisfaction probability of a strategy σ from initial state s as: $\Pr^{\mathcal{M}}_{\sigma}(s \models \varphi) = \Pr^{\mathcal{M}}_{\sigma}(s)\left\{r \in Runs^{\mathcal{M}}_{\sigma}(s)\colon L(r) \in L_{\mathcal{A}}\right\}$. The optimal satisfaction probability $\Pr^{\mathcal{M}}_{*}(s \models \varphi)$ is defined as $\sup_{\sigma\in\Sigma_{\mathcal{M}}} \Pr^{\mathcal{M}}_{\sigma}(s \models \varphi)$ and we say that $\sigma \in \Sigma_{\mathcal{M}}$ is an optimal strategy for φ if $\Pr^{\mathcal{M}}_{*}(s \models \varphi) = \Pr^{\mathcal{M}}_{\sigma}(s \models \varphi)$.

An automaton $\mathcal{A} = (\Sigma, Q, q_0, \delta, \mathsf{Acc})$ is *deterministic* if $|\delta(q, \sigma)| \leq 1$ for all $q \in Q$ and all $\sigma \in \Sigma$. $\mathcal{A}$ is *complete* if $|\delta(q, \sigma)| \geq 1$. A word in Σ^ω has exactly one run in a deterministic, complete automaton. We use common three-letter abbreviations to distinguish types of automata. The first (D or N) tells whether

the automaton is deterministic; the second denotes the acceptance condition (B for Büchi and R for Rabin). The third letter (W) says that the automaton reads ω-words. For example, an NBW is a nondeterministic Büchi automaton, and a DRW is a deterministic Rabin automaton.

Every ω-regular language is accepted by some DRW and by some NBW. In contrast, there are ω-regular languages that are not accepted by any DBW. The *Rabin index* of a Rabin automaton [6,20] is the index of its acceptance condition. The Rabin index of an ω-regular language $\mathcal{L}$ is the minimum index among those of the DRWs that accept $\mathcal{L}$. For each $n \in \mathbb{N}$ there exist ω-regular languages of Rabin index n. The languages accepted by DBWs, however, form a proper subset of the languages of index 1.

2.3 The Product MDP

Given an MDP $\mathcal{M} = (S, A, T, AP, L)$ with a designated initial state $s_0 \in S$, and a deterministic ω-automaton $\mathcal{A} = (2^{AP}, Q, q_0, \delta, \mathsf{Acc})$, the *product* $\mathcal{M} \times \mathcal{A}$ is the tuple $(S \times Q, (s_0, q_0), A, T^\times, \mathsf{Acc}^\times)$. The probabilistic transition function $T^\times \colon (S \times Q) \times A \rightharpoonup \mathcal{D}(S \times Q)$ is such that $T^\times((s,q),a)((\hat{s},\hat{q})) = T(s,a)(\hat{s})$ if $\{\hat{q}\} = \delta(q, L(s))$ and is 0 otherwise. If $\mathcal{A}$ is a DBW, Acc is defined by $F \subseteq Q \times 2^{AP} \times Q$; then $F^\times \subseteq (S \times Q) \times A \times (S \times Q)$ defines $\mathsf{Acc}^\times$ as follows: $((s,q),a,(s',q')) \in F^\times$ if and only if $(q, L(s), q') \in F$ and $T(s,a)(s') \neq 0$. If $\mathcal{A}$ is a DRW of index k, $\mathsf{Acc}^\times = \{(B_0^\times, G_0^\times), \ldots, (B_{k-1}^\times, G_{k-1}^\times)\}$. To set B_i of Acc, there corresponds $B_i^\times$ of $\mathsf{Acc}^\times$ such that $((s,q),a,(s',q')) \in B_i^\times$ if and only if $(q, L(s), q') \in B_i$ and $T(s,a)(s') \neq 0$. Likewise for $G_i^\times$.

If $\mathcal{A}$ is a nondeterministic automaton, the actions in the product are enriched to identify both the actions of the original MDP and the choice of the successor state of the nondeterministic automaton.

End-components and runs are defined for products just like for MDPs. A run of $\mathcal{M} \times \mathcal{A}$ is accepting if it satisfies the product's acceptance condition. An *accepting end-component* of $\mathcal{M} \times \mathcal{A}$ is an end-component such that every run of the product MDP that eventually dwells in it is accepting.

In view of Theorem 1, satisfaction of an ω-regular objective φ by an MDP $\mathcal{M}$ can be formulated in terms of the accepting end-components of the product $\mathcal{M} \times \mathcal{A}_\varphi$, where $\mathcal{A}_\varphi$ is an automaton accepting φ. The maximum probability of satisfaction of φ by $\mathcal{M}$ is the maximum probability, over all strategies, that a run of the product $\mathcal{M} \times \mathcal{A}_\varphi$ eventually dwells in one of its accepting end-components.

It is customary to use DRWs instead of DBWs in the construction of the product, because the latter cannot express all ω-regular objectives. On the other hand, general NBWs are not used since causal strategies cannot optimally resolve nondeterministic choices because that requires access to future events [39].

2.4 Limit-Deterministic Büchi Automata

In spite of the large gap between DRWs and DBWs in terms of indices, even a very restricted form of nondeterminism is sufficient to make DBWs as expressive as DRWs. Broadly speaking, an LDBW behaves deterministically once it has seen an accepting transition. Formally, a *limit-deterministic* Büchi automaton

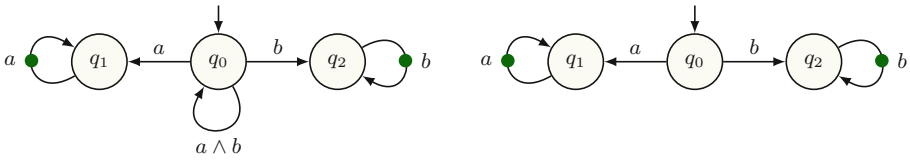


Fig. 1. Suitable (left) and unsuitable (right) LDBWs for the LTL formula $(\mathsf{G}\, a) \vee (\mathsf{G}\, b)$.

(LDBW) is an NBW $\mathcal{A} = (\Sigma, Q_i \cup Q_f, q_0, \delta, F)$ such that

- $Q_i \cap Q_f = \emptyset$, $F \subseteq Q_f \times \Sigma \times Q_f$;
- $|\delta(q, \sigma) \cap Q_i| \leq 1$ for all $q \in Q_i$ and $\sigma \in \Sigma$;
- $|\delta(q, \sigma)| \leq 1$ for all $q \in Q_f$ and $\sigma \in \Sigma$;
- $\delta(q, \sigma) \subseteq Q_f$ for all $q \in Q_f$ and $\sigma \in \Sigma$.

LDBWs are as expressive as general NBWs. Moreover, NBWs can be translated into LDBWs that can be used for the qualitative and quantitative analysis of MDPs [8,15,33,39]. We use the translation from [15], which uses LDBWs that consist of two parts: an initial deterministic automaton (without accepting transitions) obtained by a subset construction; and a final part produced by a breakpoint construction. They are connected by a single "guess", where the automaton guesses a singleton subset of the reachable states to start the breakpoint construction. Like in other constructions (e.g. [33]), one can compose the resulting automata with an MDP, such that the optimal control of the product defines a control on the MDP that maximizes the probability of obtaining a word from the language of the automaton. We refer to LDBWs with this property as *suitable* limit-deterministic automata (SLDBWs).

Definition 1 (Suitable LDBW). *An SLDBW $\mathcal{A}$ for property φ is an LDBW that recognizes φ and such that, for every finite MDP $\mathcal{M}$, there exists a positional strategy $\sigma \in \Sigma_{\mathcal{M}\times\mathcal{A}}$ such that the probability of satisfying the Büchi condition in the Markov chain $(\mathcal{M} \times \mathcal{A})_\sigma$ is $\Pr_*^{\mathcal{M}}(s \models \varphi)$.*

Although the construction of a suitable LDBW reaches back to the 80s [39], not all LDBWs are suitable. Broadly speaking, the nondeterministic decisions taken in the initial part may not depend on the future—though it may depend on the state of an MDP. The example LDBW from Fig. 1 (left) satisfies the property: it can try to delay to progress to one of the accepting states to when an end-component in an MDP is reached that always produces a's or b's, respectively. In contrast, the LDBW from Fig. 1 (right)—which recognizes the same language—will have to make the decision of seeing only a's or only b's immediately, without the option to wait for reaching an end-component. This makes it unsuitable for the use in MDPs.

Theorem 2 [8,15,33,39]**.** *Suitable limit-deterministic Büchi automata exist for all ω-regular languages.*

SLDBWs—and their properties described in Definition 1—are used in the qualitative and quantitative model checking algorithms in [8,15,33,39]. The accepting end-components of the product MDPs are all using only states from the final part of the SLDBW. Büchi acceptance then allows for using memoryless almost sure winning strategies in the accepting end-components, while outside of accepting end-components a memoryless strategy that maximizes the chance of reaching such an end-component can be used. The distinguishing property is the guarantee that they provide the correct probability, while using a product with a general NBW would only provide a value that cannot exceed it.

2.5 Linear Time Logic Objectives

LTL (Linear Time Logic) is a temporal logic whose formulae describe a subset of the ω-regular languages, which is often used to specify objectives in human-readable form. Translations exist from LTL to various forms of automata, including NBW, DRW, and SLDBW. Given a set of atomic propositions AP, a is an LTL formula for each $a \in AP$. Moreover, if φ and ψ are LTL formulae, so are $\neg\varphi, \varphi \vee \psi, \mathsf{X}\,\varphi, \psi\,\mathsf{U}\,\varphi$. Additional operators are defined as abbreviations: $\top \stackrel{\text{def}}{=} a \vee \neg a$; $\bot \stackrel{\text{def}}{=} \neg\top$; $\varphi \wedge \psi \stackrel{\text{def}}{=} \neg(\neg\varphi \vee \neg\psi)$; $\varphi \rightarrow \psi \stackrel{\text{def}}{=} \neg\varphi \vee \psi$; $\mathsf{F}\,\varphi \stackrel{\text{def}}{=} \top\,\mathsf{U}\,\varphi$; and $\mathsf{G}\,\varphi \stackrel{\text{def}}{=} \neg\,\mathsf{F}\,\neg\varphi$. We write $w \models \varphi$ if ω-word w over 2^{AP} satisfies LTL formula φ. The satisfaction relation is defined inductively [2,24].

2.6 Reinforcement Learning

For an MDP $\mathcal{M}$ and an objectives $Reward^{\mathcal{M}} \in \{Reach(T)^{\mathcal{M}}, Disct(\lambda)^{\mathcal{M}}, Avg^{\mathcal{M}}\}$, the optimal reward and an optimal strategy can be computed using value iteration, policy iteration, or, in polynomial time, using linear programming [12,30]. On the other hand, for ω-regular objectives (given as DRW, SLDBW, or LTL formulae) optimal satisfaction probabilities and strategies can be computed using graph-theoretic techniques (computing accepting end-component and then maximizing the probability to reach states in such components) over the product structure. However, when the MDP transition/reward structure is unknown, such techniques are not applicable.

For MDPs with unknown transition/reward structure, *reinforcement learning* [37] provides a framework to compute optimal strategies from repeated interactions with the environment. Of the two main approaches to reinforcement learning in MDPs, *model-free* approaches and *model-based* approaches the former, which is asymptotically space-efficient [36], has been demonstrated to scale well [14,25,35]. In a model-free approach such as Q-learning [31,37], the learner computes optimal strategies without explicitly estimating the transition probabilities and rewards. We focus on making it possible for model-free RL to learn a strategy that maximizes the probability of satisfying a given ω-regular objective.

3 Problem Statement and Motivation

Given MDP $\mathcal{M}$ with unknown transition structure and ω-regular objective φ, we seek a strategy that maximizes the probability that $\mathcal{M}$ satisfies φ.

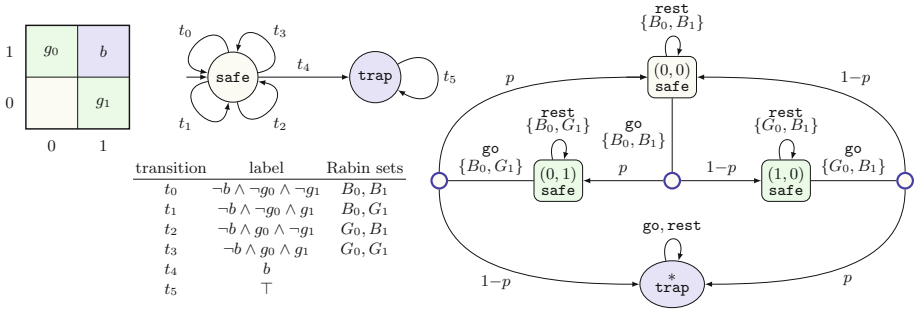


Fig. 2. A grid-world example (left), a Rabin automaton for $[(\mathsf{F}\,\mathsf{G}\,g_0) \vee (\mathsf{F}\,\mathsf{G}\,g_1)] \wedge \mathsf{G}\,\neg b$ (center), and product MDP (right).

To apply model-free RL algorithms to this task, one needs to define rewards that depend on the observations of the MDP and reflect the satisfaction of the objective. It is natural to use the product of the MDP and an automaton monitoring the satisfaction of the objective to assign suitable rewards to various actions chosen by the learning algorithm.

Sadigh *et al.* [32] were the first to apply model-free RL to a qualitative-version of this problem, i.e., to learn a strategy that satisfies the property with probability 1. For an MDP $\mathcal{M}$ and a DRW $\mathcal{A}_\varphi$ of index k, they formed the product MDP $\mathcal{M} \times \mathcal{A}_\varphi$ with k different "Rabin" reward functions $\rho_1, \ldots, \rho_k$. The function ρ_i corresponds to the Rabin pair $(B_i^\times, G_i^\times)$: it assigns a fixed negative reward $-R_- < 0$ to all edges in $B_i^\times$ and a fixed positive reward $R_+ > 0$ to all edges in $G_i^\times$. [32] claimed that if there exists a strategy satisfying an ω-regular objective φ with probability 1, then there exists a Rabin pair i, discount factor $\lambda_* \in [0, 1[$, and suitably high ratio R_*, such that for all $\lambda \in [\lambda_*, 1[$ and $R_-/R_+ \geq R_*$, any strategy maximizing λ-discounted reward for the MDP $(\mathcal{M} \times \mathcal{A}_\varphi, \rho_i)$ also satisfies the ω-regular objective φ with probability 1. Using Blackwell-optimality theorem [19], a paraphrase of this claim is that if there exists a strategy satisfying an ω-regular objective φ with probability 1, then there exists a Rabin pair i and suitably high ratio R_*, such that for all $R_-/R_+ \geq R_*$, any strategy maximizing expected average reward for the MDP $(\mathcal{M} \times \mathcal{A}_\varphi, \rho_i)$ also satisfies the ω-regular objective φ with probability 1. This approach has two faults, the second of which also affects approaches that replace DRWs with LDBWs [16,17].

1. We provide in Example 1 an MDP and an ω-regular objective φ with Rabin index 2, such that, although there is a strategy that satisfies the property with probability 1, optimal average strategies from any Rabin reward do not satisfy the objective with probability 1.
2. Even for an ω-regular objective with one Rabin pair (B, G) and $B{=}\emptyset$—i.e., one that can be specified by a DBW—we demonstrate in Example 2 that the problem of finding a strategy that satisfies the property with probability 1 may not be reduced to finding optimal average strategies.

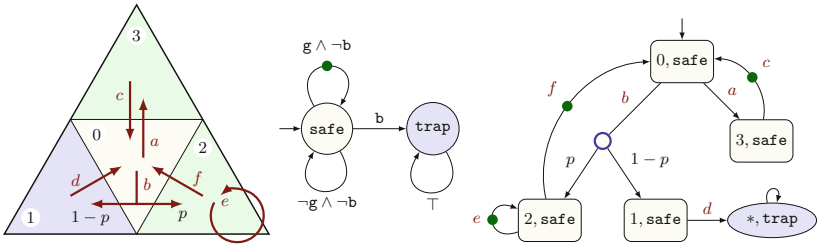


Fig. 3. A grid-world example. The arrows represent actions (left). When action b is performed, Cell 2 is reached with probability p and Cell 1 is reached with probability $1-p$, for $0 < p < 1$. Deterministic Büchi automaton for $\varphi = (\mathsf{G}\,\neg\mathtt{b}) \wedge (\mathsf{G}\,\mathsf{F}\,\mathtt{g})$ (center). The dotted transition is the only accepting transition. Product MDP (right).

Example 1 (Two Rabin Pairs). Consider the MDP given as a simple grid-world example shown in Fig. 2. Each cell (state) of the MDP is labeled with the atomic propositions that are true there. In each cell, there is a choice between two actions `rest` and `go`. With action `rest` the state of the MDP does not change. However, with action `go` the MDP moves to the other cell in the same row with probability p, or to the other cell in the same column with probability $1-p$. The initial cell is $(0, 0)$.

The specification is given by LTL formula $\varphi = [(\mathsf{F}\,\mathsf{G}\,g_0) \vee (\mathsf{F}\,\mathsf{G}\,g_1)] \wedge \mathsf{G}\,\neg b$. A DRW that accepts φ is shown in Fig. 2. The DRW has two accepting pairs: (B_0, G_0) and (B_1, G_1). The table besides the automaton gives, for each transition, its label and the B and G sets to which it belongs.

The optimal strategy that satisfies the objective φ with probability 1 chooses `go` in Cell $(0, 0)$ and chooses `rest` subsequently. However, for both Rabin pairs, the optimal strategy for expected average reward is to maximize the probability of reaching one of the $(0, 1)$, `safe` or $(1, 0)$, `safe` states of the product and stay there forever. For the first accepting pair the maximum probability of satisfaction is $\frac{1}{2-p}$, while for the second pair it is $\frac{1}{1+p}$.

Example 2 (DBW to Expected Average Reward Reduction). This counterexample demonstrates that even for deterministic Büchi objectives, the problem of finding an optimal strategy satisfying an objective may not be reduced to the problem of finding an optimal average strategy. Consider the simple grid-world example of Fig. 3 with the specification $\varphi = (\mathsf{G}\,\neg\mathtt{b}) \wedge (\mathsf{G}\,\mathsf{F}\,\mathtt{g})$, where atomic proposition `b` (blue) labels Cell 1 and atomic proposition `g` (green) labels Cells 2 and 3. Actions enabled in various cells and their probabilities are depicted in the figure.

The strategy from Cell 0 that chooses Action a guarantees satisfaction of φ with probability 1. An automaton with accepting transitions for φ is shown in Fig. 3; it is a DBW (or equivalently a DRW with one pair (B, G) and $B = \emptyset$).

The product MDP is shown at the bottom of Fig. 3. All states whose second component is `trap` have been merged. Notice that there is no negative reward since the set B is empty. If reward is positive and equal for all accepting transitions, and 0 for all other transitions, then when $p > 1/2$, the strategy that

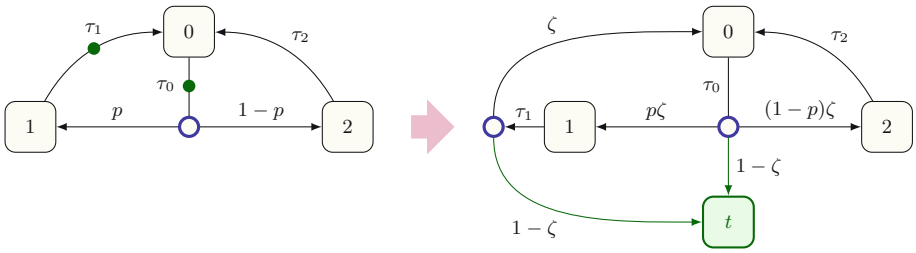


Fig. 4. Adding transitions to the target in the augmented product MDP.

maximizes expected average reward chooses Action b in the initial state and Action e from State $(2, \texttt{safe})$. Note that, for large values of λ, the optimal expected average reward strategies are also optimal strategies for the λ-discounted reward objective. However, these strategies are not optimal for ω-regular objectives.

Example 1 shows that one cannot select a pair from a Rabin acceptance condition ahead of time. This problem can be avoided by the use of Büchi acceptance conditions. While DBWs are not sufficiently expressive, SLDBWs express all ω-regular properties and are suitable for probabilistic model checking. In the next section, we show that they are also "the ticket" for model-free reinforcement learning, because they allow us to maximize the probability of satisfying an ω-regular specification by solving a reachability probability problem that can be solved efficiently by off-the-shelf RL algorithms.

4 Model-Free RL from Omega-Regular Rewards

We now reduce the model checking problem for a given MDP and SLDBW to a reachability problem by slightly changing the structure of the product: We add a target state t that can be reached with a given probability $1-\zeta$ whenever visiting an accepting transition of the original product MDP.

Our reduction avoids the identification of winning end-components and thus allows a natural integration to a wide range of model-free RL approaches. Thus, while the proofs do lean on standard model checking properties that are based on identifying winning end-components, they serve as a justification not to consider them when running the learning algorithm. In the rest of this section, we fix an MDP $\mathcal{M}$ and an SLDBW $\mathcal{A}$ for the ω-regular property φ.

Definition 2 (Augmented Product). *For any $\zeta \in]0,1[$, the* augmented MDP $\mathcal{M}^\zeta$ *is an MDP obtained from $\mathcal{M} \times \mathcal{A}$ by adding a new state t with a self-loop to the set of states of $\mathcal{M} \times \mathcal{A}$, and by making t a destination of each accepting transition τ of $\mathcal{M} \times \mathcal{A}$ with probability $1-\zeta$. The original probabilities of all other destinations of an accepting transition τ are multiplied by ζ.*

An example of an augmented MDP is shown in Fig. 4. With a slight abuse of notation, if σ is a strategy on the augmented MDP $\mathcal{M}^\zeta$, we denote by σ also the strategy on $\mathcal{M} \times \mathcal{A}$ obtained by removing t from the domain of σ.

We let $p_s^\sigma(\zeta)$ denote the probability of reaching t in $\mathcal{M}_\sigma^\zeta$ when starting at state s. Notice that we can encode this value as the expected average reward in the following rewardful MDP $(\mathcal{M}^\zeta, \rho)$, where we set the reward function $\rho(t,a) = 1$ for all $a \in A$ and $\rho(s,a) = 0$ otherwise. For any strategy σ, the probability $p_s^\sigma(\zeta)$ and the reward of σ from s in $(\mathcal{M}^\zeta, \rho)$ are the same. We also let a_s^σ be the probability that a run that starts from s in $(\mathcal{M} \times \mathcal{A})_\sigma$ is accepting.

Lemma 1. *If σ is a positional strategy on $\mathcal{M}^\zeta$, then, for every state s of the Markov chain $(\mathcal{M}\times\mathcal{A})_\sigma$, the following holds:*

1. *if the state s is in a rejecting BSCC of $(\mathcal{M}\times\mathcal{A})_\sigma$, then $p_s^\sigma(\zeta) = 0$;*
2. *if the state s is in an accepting BSCC of $(\mathcal{M}\times\mathcal{A})_\sigma$, then $p_s^\sigma(\zeta) = 1$;*
3. *the probability $p_s^\sigma(\zeta)$ of reaching t is greater than a_s^σ; and*
4. *if $p_s^\sigma(\zeta)$=1 then no rejecting BSCC is reachable from s in $(\mathcal{M}\times\mathcal{A})_\sigma$ and $a_s^\sigma = 1$.*

Proof. (1) holds as there are no accepting transition in a rejecting BSCC of $(\mathcal{M}\times\mathcal{A})_\sigma$, and so t cannot be reached when starting at s in $\mathcal{M}_\sigma^\zeta$. (2) holds because t (with its self-loop) is the only BSCC reachable from s in $\mathcal{M}^\zeta$. In other words, t (with its self-loop) and the rejecting BSCCs of $(\mathcal{M}\times\mathcal{A})_\sigma$ are the only BSCCs in $\mathcal{M}_\sigma^\zeta$. (3) then follows, because the same paths lead to a rejecting BSCCs in $(\mathcal{M}\times\mathcal{A})_\sigma$ and $\mathcal{M}_\sigma^\zeta$, while the probability of each such a path is no larger—and strictly smaller iff it contains an accepting transition—than in $\mathcal{M}_\sigma^\zeta$. (4) holds because, if $p_s^\sigma(\zeta) = 1$, then t (with its self-loop) is the only BSCC reachable from s in $\mathcal{M}_\sigma^\zeta$. Thus, there is no path to a rejecting BSCC in $\mathcal{M}_\sigma^\zeta$, and therefore no path to a rejecting BSCC in $(\mathcal{M}\times\mathcal{A})_\sigma$. □

Lemma 2. *Let σ be a positional strategy on $\mathcal{M}^\zeta$. For every state s of $\mathcal{M}^\zeta$, we have that $\lim_{\zeta\uparrow 1} p_s^\sigma(\zeta) = a_s^\sigma$.*

Proof. As shown in Lemma 1(3) for all ζ, we have $p_s^\sigma(\zeta) \geq a_s^\sigma$. For a coarse approximation of their difference, we recall that $(\mathcal{M}\times\mathcal{A})_\sigma$ is a finite Markov chain. The expected number of transitions taken before reaching a BSCC from s in $(\mathcal{M}\times\mathcal{A})_\sigma$ is therefore a finite number. Let us refer to the—no larger—expected number of accepting transitions taken before reaching a BSCC when starting at s in $(\mathcal{M}\times\mathcal{A})_\sigma$ as f_s^σ. We claim that $a_s^\sigma \geq p_s^\sigma(\zeta) - (1-\zeta)\cdot f_s^\sigma$. This is because the probability of reaching a rejecting BSCC in $(\mathcal{M}\times\mathcal{A})_\sigma$ is at most the probability of reaching a rejecting BSCC in $\mathcal{M}_\sigma^\zeta$, which is at most $1 - p_s^\sigma(\zeta)$, plus the probability of moving on to t from a state that is not in any BSCC in $(\mathcal{M}\times\mathcal{A})_\sigma$, which we are going to show next is at most $f_s^\sigma \cdot (1-\zeta)$.

First, a proof by induction shows that $(1-\zeta^k) \leq k(1-\zeta)$ for all $k \geq 0$. Let $P_s^\sigma(\zeta, k)$ be the probability of generating a path from s with k accepting transitions before t or a node in some BSCC of $(\mathcal{M}\times\mathcal{A})_\sigma$ is reached in $\mathcal{M}_\sigma^\zeta$. The probability of seeing k accepting transitions and not reaching t is at least ζ^k. Therefore, probability of moving to t from a state not in any BSCC is at most

$$\sum_k P_s^\sigma(\zeta,k)(1-\zeta^k) \leq \sum_k P_s^\sigma(\zeta,k)k\cdot(1-\zeta) \leq f_s^\sigma \cdot (1-\zeta).$$

The proof is now complete. □

This provides us with our main theorem.

Theorem 3. *There exists a threshold $\zeta' \in]0,1[$ such that, for all $\zeta > \zeta'$ and every state s, any strategy σ that maximizes the probability $p_s^\sigma(\zeta)$ of reaching the sink in $\mathcal{M}^\zeta$ is (1) an optimal strategy in $\mathcal{M} \times \mathcal{A}$ from s and (2) induces an optimal strategy for the original MDP $\mathcal{M}$ from s with objective φ.*

Proof. We use the fact that it suffices to study positional strategies, and there are only finitely many of them. Let σ_1 be an optimal strategy of $\mathcal{M} \times \mathcal{A}$, and let σ_2 be a strategy that has the highest likelihood of creating an accepting run among all non-optimal memoryless ones. (If σ_2 does not exist, then all strategies are equally good, and it does not matter which one is chosen.) Let $\delta = a_s^{\sigma_1} - a_s^{\sigma_2}$.

Let $f_{\max} = \max_\sigma \max_s f_s^\sigma$, where σ ranges over positional strategies only, and f_s^σ is defined as in Lemma 2. We claim that it suffices to pick $\zeta' \in]0,1[$ such that $(1-\zeta') \cdot f_{\max} < \delta$. Suppose that σ is a positional strategy that is optimal in $\mathcal{M}^\zeta$ for $\zeta > \zeta'$, but is not optimal in $\mathcal{M} \times \mathcal{A}$. We then have

$$a_s^\sigma \leq p_s^\sigma(\zeta) \leq a_s^\sigma + (1-\zeta) f_s^\sigma < a_s^\sigma + \delta \leq a_s^{\sigma_1} \leq p_s^{\sigma_1}(\zeta),$$

where these inequalities follow, respectively, from: Lemma 1(3), the proof of Lemma 2, the definition of ζ', the assumption that σ is not optimal and the definition of δ, and the last one from Lemma 1(3). This shows that $p_s^\sigma(\zeta) < p_s^{\sigma_1}(\zeta)$, i.e., σ is not optimal in $\mathcal{M}^\zeta$; a contradiction. Therefore, any positional strategy that is optimal in $\mathcal{M}^\zeta$ for $\zeta > \zeta'$ is also optimal in $\mathcal{M} \times \mathcal{A}$.

Now, suppose that σ is a positional strategy that is optimal in $\mathcal{M} \times \mathcal{A}$. Then the probability of satisfying φ in $\mathcal{M}$ when starting at s is at least[1] a_s^σ. At the same time, if there was a strategy for which the probability of satisfying φ in $\mathcal{M}$ is $> a_s^\sigma$, then the property of $\mathcal{A}$ to be an SLDBW (Definition 1) would guarantee the existence of strategy σ' for which $a_s^{\sigma'} > a_s^\sigma$; a contradiction with the assumption that σ is optimal. Therefore any positional strategy that is optimal in $\mathcal{M} \times \mathcal{A}$ induces an optimal strategy in $\mathcal{M}$ with objective φ. □

Corollary 1. *Due to Lemma 1(4), $\mathcal{M}$ satisfies φ almost surely if and only if the sink is almost surely reachable in $\mathcal{M}^\zeta$ for all $0 < \zeta < 1$.*

Theorem 3 leads to a very simple model-free RL algorithm. The augmented product is not built by the RL algorithm, which does not know the transition structure of the environment MDP. Instead, the observations of the MDP are used by an *interpreter* process to compute a run of the objective automaton. The interpreter also extracts the set of actions for the learner to choose from. If the automaton is not deterministic and it has not taken the one nondeterministic transition it needs to take yet, the set of actions the interpreter provides to the learner includes the choice of special "jump" actions that instruct the automaton to move to a chosen accepting component.

[1] This holds for all nondeterministic automata that recognize the models of φ: an accepting run establishes that the path was a model of φ.

When the automaton reports an accepting transition, the interpreter gives the learner a positive reward with probability $1 - \zeta$. When the learner actually receives a reward, the training episode terminates. Any RL algorithm that maximizes this probabilistic reward is guaranteed to converge to a policy that maximizes the probability of satisfaction of the ω-regular objective.

5 Experimental Results

We implemented the construction described in the previous sections in a tool named MUNGOJERRIE [11], which reads MDPs described in the PRISM language [21], and ω-regular automata written in the HOA format [1,9]. MUNGOJERRIE builds the augmented product $\mathcal{M}^{\zeta}$, provides an interface for RL algorithms akin to that of [5] and supports probabilistic model checking. Our algorithm computes, for each pair (s, a) of state and action, the maximum probability of satisfying the given objective after choosing action a from state s by using off-the-shelf, temporal difference algorithms. Not all actions with maximum probability are part of positional optimal strategies—consider a product MDP with one state and two actions, a and b, such that a enables an accepting self-loop, and b enables a non-accepting one: both state/action pairs are assigned probability 1. In b's case, because choosing b once—or a finite number of times—does not preclude acceptance. Since the probability values alone do not identify a pure optimal strategy, MUNGOJERRIE computes an optimal mixed strategy, uniformly choosing all maximum probability actions from a state.

The MDPs on which we tested our algorithms [26] are listed in Table 1. For each model, the numbers of decision states in the MDP, the automaton, and the product MDP are given. Next comes the probability of satisfaction of the objective for the strategy chosen by the RL algorithm as computed by the model checker (which has full access to the MDP). This is followed by the estimate of the probability of satisfaction of the objective computed by the RL algorithm and the time taken by learning. The last six columns report values of the hyperparameters when they deviate from the default values: ζ controls the probability of reward, ϵ is the exploration rate, α is the learning rate, and tol is the tolerance for probabilities to be considered different. Finally, ep-l controls the episode length (it is the maximum allowed length of a path in the MDP that does contain an accepting edge) and ep-n is the number of episodes. All performance data are the averages of three trials with Q-learning. Rewards are undiscounted, so that the value of a state-action pair computed by Q-learning is a direct estimate of the probability of satisfaction of the objective from that state when taking that action.

Models `twoPairs` and `riskReward` are from Examples 1 and 2, respectively. Model `deferred` is discussed later. Models `grid5x5` and `trafficNtk` are from [32]. The three "windy" MDPs are taken from [37]. The "frozen" examples are from [27]. Some ω-regular objectives are simple reachability requirements (e.g., `frozenSmall` and `frozenLarge`). The objective for the `othergrid` models is to collect three types of coupons, while incurring at most one of two types of penalties. In `doublegrid` two agents simultaneously move across the grid.

Table 1. Q-learning results. The default values of the learner hyperparameters are: $\zeta = 0.99$, $\epsilon = 0.1$, $\alpha = 0.1$, tol= 0.01, ep-l= 30, and ep-n= 20000. Times are in seconds.

Name	states	aut.	prod.	prob.	est.	time	ζ	ε	α	tol	ep-l	ep-n
`twoPairs`	4	4	16	1	1	0.26						
`riskReward`	4	2	8	1	1	1.47						
`deferred`	41	1	41	1	1	1.01						
`grid5x5`	25	3	75	1	1	10.82		0.01	0.2		400	30k
`trafficNtk`	122	13	462	1	1	2.89						
`windy`	123	2	240	1	1	12.35	0.95	0.001	0.05	0	900	200k
`windyKing`	130	2	256	1	1	14.34	0.95	0.02	0.2	0	300	120k
`windyStoch`	130	2	260	1	1	47.70	0.95	0.02	0.2	0	300	200k
`frozenSmall`	16	3	48	0.823	0.83	0.51			0.05	0	200	
`frozenLarge`	64	3	192	1	1	1.81			0.05	0	700	
`othergrid6`	36	25	352	1	1	10.80				0	300	75k
`othergrid20`	400	25	3601	1	1	78.00	0.9999	0.07	0.2	0	5k	
`othergrid40`	1600	25	14401	1	0.99	87.90	0.9999	0.05	0.2	0	14k	25k
`doublegrid8`	4096	3	12287	1	1	45.50				0	3k	100k
`doublegrid12`	20736	3	62207	1	1	717.6				0	20k	300k
`slalom`	36	5	84	1	1	0.98						
`rps1`	121	2	130	0.768	0.76	5.21		0.12	0.006	0		500k
`dpenny`	52	2	65	0.5	0.5	1.99		0.001	0.2	0	50	120k
`devious`	11	1	11	1	1	0.81						
`arbiter2`	32	3	72	1	1	5.16			0.5	0.02	200	
`knuthYao`	13	3	39	1	1	0.31					100	
`threeWayDuel`	10	3	13	0.397	0.42	0.08						
`mutual4-14`	27600	128	384386	1	1	2.74						
`mutual4-15`	27600	527	780504	1	1	3.61						

The objective for `slalom` is given by the LTL formula $\mathsf{G}(p \rightarrow \mathsf{X}\,\mathsf{G}\,\neg q) \wedge \mathsf{G}(q \rightarrow \mathsf{X}\,\mathsf{G}\,\neg p)$. For model `rps1` the strategy found by RL is (slightly) suboptimal. The difference in probability of 0.01 is explained by the existence of many strategies of nearly identical values. Model `mutual` [7,15,34] describes the mutual exclusion protocol of Pnueli and Zuck [29]. Though large, this model is easy for learning.

Figure 5 illustrates how increasing the parameter ζ makes the RL algorithm less sensitive to the presence of transient (not in an end-component) accepting transitions. Model `deferred` consists of two chains of states: one, which the agent choses with action a, has accepting transitions throughout, but leads to an end-component that is not accepting. The other chain, selected with action b, leads to an accepting end-component, but has no other accepting transitions. There are no other decisions in the model; hence only two strategies are possible, which we denote by a and b, depending on the action chosen.

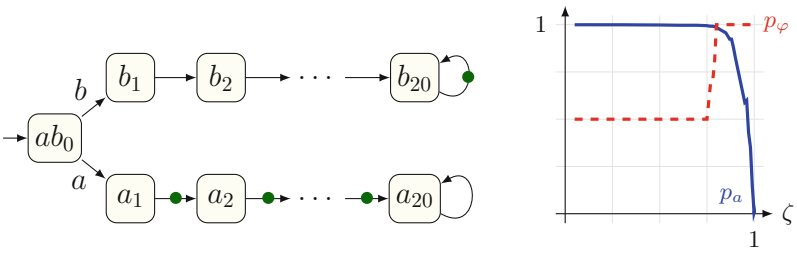


Fig. 5. Model `deferred` and effect of ζ on it.

The curve labeled p_a in Fig. 5 gives the probability of satisfaction under strategy a of the MDP's objective as a function of ζ as computed by Q-learning. The number of episodes is kept fixed at 20,000 and each episode has length 80. Each data point is the average of five experiments for the same value of ζ.

For values of ζ close to 0, the chance is high that the sink is reached directly from a transient state. Consequently, Q-learning considers strategies a and b equally good. For this reason, the probability of satisfaction of the objective, p_φ, according to the strategy that mixes a and b, is computed by Mungojerrie's model checker as 0.5. As ζ approaches 1, the importance of transient accepting transitions decreases, until the probability computed for strategy a is no longer considered to be approximately the same as the probability of strategy b. When that happens, p_φ abruptly goes from 0.5 to its true value of 1, because the pure strategy b is selected. The value of p_a continues to decline for larger values of ζ until it reaches its true value of 0 for $\zeta = 0.9999$. Probability p_b, not shown in the graph, is 1 throughout.

The change in value of p_φ does not contradict Theorem 3, which says that $p_b = 1 > p_a$ for all values of ζ. In practice a high value of ζ may be needed to reliably distinguish between transient and recurrent accepting transitions in numerical computation. Besides, Theorem 3 suggests that even in the almost-sure case there is a meaningful path to the target strategy where the likelihood of satisfying φ can be expected to grow. This is important, as it comes with the promise of a generally increasing quality of intermediate strategies.

6 Conclusion

We have reduced the problem of maximizing the satisfaction of an ω-regular objective in a MDP to reachability on a product graph augmented with a sink state. This change is so simple and elegant that it may surprise that it has not been used before. But the reason for this is equally simple: it does not help in a model checking context, as it does not remove any analysis step there. In a reinforcement learning context, however, it simplifies our task significantly. In previous attempts to use suitable LDBW [4], the complex part of the model checking problem—identifying the accepting end-components—is still present. Only after this step, which is expensive and requires knowledge of the structure

of the underlying MDP, can these methods reduce the search for optimal satisfaction to the problem of maximizing the chance to reach those components. Our reduction avoids the identification of accepting end-components entirely and thus allows a natural integration with a wide range of model-free RL approaches.

References

1. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-Dynamic Programming. Athena Scientific, Belmont (1996)
4. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
5. Brockman, G., et al.: OpenAI Gym. CoRR, abs/1606.01540 (2016)
6. Carton, O., Maceiras, R.: Computing the Rabin index of a parity automaton. Theoret. Inf. Appl. **33**, 495–505 (1999)
7. Chatterjee, K., Gaiser, A., Křetínský, J.: Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 559–575. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_37
8. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM **42**(4), 857–907 (1995)
9. cpphoafparser (2016). https://automata.tools/hoa/cpphoafparser. Accessesd 05 Sept 2018
10. de Alfaro, L.: Formal Verification of Probabilistic Systems. Ph.D. thesis, Stanford University (1998)
11. Eliot, T.S.: Old Possum's Book of Practical Cats. Harcourt Brace Jovanovich, San Diego (1939)
12. Feinberg, E.A., Shwartz, A. (eds.): Handbook of Markov Decision Processes. Springer, New York (2002). https://doi.org/10.1007/978-1-4615-0805-2
13. Fu, J., Topcu, U.: Probably approximately correct MDP learning and control with temporal logic constraints. In: Robotics: Science and Systems, July 2014
14. Guez, A., et al.: An investigation of model-free planning. CoRR, abs/1901.03559 (2019)
15. Hahn, E.M., Li, G., Schewe, S., Turrini, A., Zhang, L.: Lazy probabilistic model checking without determinisation. In: Concurrency Theory (CONCUR), pp. 354–367 (2015)
16. Hasanbeig, M., Abate, A., Kroening, D.: Logically-correct reinforcement learning. CoRR, abs/1801.08099v1, January 2018
17. Hasanbeig, M., Abate, A., Kroening, D.: Certified reinforcement learning with logic guidance. arXiv e-prints, arXiv:1902.00778, February 2019
18. Hiromoto, M., Ushio, T.: Learning an optimal control policy for a Markov decision process under linear temporal logic specifications. In: Symposium Series on Computational Intelligence, pp. 548–555, December 2015

19. Hordijk, A., Yushkevich, A.A.: Blackwell optimality. In: Feinberg, E.A., Shwartz, A. (eds.) Handbook of Markov Decision Processes: Methods and Applications, pp. 231–267. Springer, Boston (2002). https://doi.org/10.1007/978-1-4615-0805-2_8
20. Krishnan, S.C., Puri, A., Brayton, R.K., Varaiya, P.P.: The Rabin index and chain automata, with applications to automata and games. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 253–266. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60045-0_55
21. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
22. Lahijanian, M., Andersson, S.B., Belta, C.: Temporal logic motion planning and control with probabilistic satisfaction guarantees. IEEE Trans. Robot. **28**(2), 396–409 (2012)
23. Li, X., Vasile, C.I., Belta, C.: Reinforcement learning with temporal logic rewards. In: International Conference on Intelligent Robots and Systesm (IROS), pp. 3834–3839 (2017)
24. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems *Specification*. Springer, New York (1991). https://doi.org/10.1007/978-1-4612-0931-7
25. Mnih, V., et al.: Human-level control through reinforcement learning. Nature **518**, 529–533 (2015)
26. Mungojerrie ω-regular reinforcement learning benchmarks (2019). https://plv.colorado.edu/omega-regular-rl-benchmarks-2019
27. OpenAI Gym (2018). https://gym.openai.com. Accessed 05 Sept 2018
28. Perrin, D., Pin, J.É.: Infinite Words: Automata, Semigroups, Logic and Games. Elsevier, Amsterdam (2004)
29. Pnueli, A., Zuck, L.: Verification of multiprocess probabilistic protocols. Distrib. Comput. **1**, 53–72 (1986)
30. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (1994)
31. Riedmiller, M.: Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In: Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) ECML 2005. LNCS (LNAI), vol. 3720, pp. 317–328. Springer, Heidelberg (2005). https://doi.org/10.1007/11564096_32
32. Sadigh, D., Kim, E., Coogan, S., Sastry, S.S., Seshia, S.A.: A learning based approach to control synthesis of Markov decision processes for linear temporal logic specifications. In: IEEE Conference on Decision and Control (CDC), pp. 1091–1096, December 2014
33. Sickert, S., Esparza, J., Jaax, S., Křetínský, J.: Limit-deterministic Büchi automata for linear temporal logic. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 312–332. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_17
34. Sickert, S., Křetínský, J.: MoChiBA: probabilistic LTL model checking using limit-deterministic Büchi automata. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 130–137. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_9
35. Silver, D., et al.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (2016)

36. Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L.: PAC model-free reinforcement learning. In: International Conference on Machine Learning ICML, pp. 881–888 (2006)
37. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction, 2nd edn. MIT Press, Cambridge (2018)
38. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, pp. 133–191. The MIT Press/Elsevier, Cambridge (1990)
39. Vardi, M.Y.: Automatic verification of probabilistic concurrent finite state programs. In: Foundations of Computer Science, pp. 327–338 (1985)
40. Wiering, M., van Otterlo, M. (eds.): Reinforcement Learning: State of the Art. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27645-3

Verifiably Safe Off-Model Reinforcement Learning

Nathan Fulton(✉) and André Platzer

Computer Science Department, Carnegie Mellon University,
Pittsburgh, USA
{nathanfu,aplatzer}@cs.cmu.edu

Abstract. The desire to use reinforcement learning in safety-critical settings has inspired a recent interest in formal methods for learning algorithms. Existing formal methods for learning and optimization primarily consider the problem of constrained learning or constrained optimization. Given a single correct model and associated safety constraint, these approaches guarantee efficient learning while provably avoiding behaviors outside the safety constraint. Acting well given an accurate environmental model is an important pre-requisite for safe learning, but is ultimately insufficient for systems that operate in complex heterogeneous environments. This paper introduces verification-preserving model updates, the first approach toward obtaining formal safety guarantees for reinforcement learning in settings where multiple possible environmental models must be taken into account. Through a combination of inductive data and deductive proving with design-time model updates and runtime model falsification, we provide a first approach toward obtaining formal safety proofs for autonomous systems acting in heterogeneous environments.

1 Introduction

The desire to use reinforcement learning in safety-critical settings has inspired several recent approaches toward obtaining formal safety guarantees for learning algorithms. Formal methods are particularly desirable in settings such as self-driving cars, where testing alone cannot guarantee safety [22]. Recent examples of work on formal methods for reinforcement learning algorithms include justified speculative control [14], shielding [3], logically constrained learning [17], and constrained Bayesian optimization [16]. Each of these approaches provide formal safety guarantees for reinforcement learning and/or optimization algorithms by stating assumptions and specifications in a formal logic, generating monitoring conditions based upon specifications and environmental assumptions, and then

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) under grant number FA8750-18-C-0092. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 413–430, 2019.
https://doi.org/10.1007/978-3-030-17462-0_28

leveraging these monitoring conditions to constrain the learning/optimization process to a known-safe subset of the state space.

Existing formal methods for learning and optimization consider the problem of constrained learning or constrained optimization [3,14,16,17]. They address the question: assuming we have a single accurate environmental model with a given specification, how can we learn an efficient control policy respecting this specification?

Correctness proofs for control software in a single well-modeled environment are necessary but not sufficient for ensuring that reinforcement learning algorithms behave safely. Modern cyber-physical systems must perform a large number of subtasks in many different environments and must safely cope with situations that are not anticipated by system designers. These design goals motivate the use of reinforcement learning in safety-critical systems. Although some formal methods suggest ways in which formal constraints might be used to inform control even when modeling assumptions are violated [14], none of these approaches provide formal safety guarantees when environmental modeling assumptions are violated.

Holistic approaches toward safe reinforcement learning should provide formal guarantees even when a single, a priori model is not known at design time. We call this problem *verifiably safe off-model learning*. In this paper we introduce a first approach toward obtaining formal safety proofs for off-model learning. Our approach consists of two components: (1) a model synthesis phase that constructs a set of candidate models together with provably correct control software, and (2) a runtime model identification process that selects between available models at runtime in a way that preserves the safety guarantees of all candidate models.

Model update learning is initialized with a set of models. These models consist of a set of differential equations that model the environment, a control program for selecting actuator inputs, a safety property, and a formal proof that the control program constrains the overall system dynamics in a way that correctly ensures the safety property is never violated.

Instead of requiring the existence of a single accurate initial model, we introduce *model updates* as syntactic modifications of the differential equations and control logic of the model. We call a model update *verification-preserving* if there is a corresponding modification to the formal proof establishing that the modified control program continues to constrain the system of differential equations in a way that preserves the original model's safety properties.

Verification-preserving model updates are inspired by the fact that different parts of a model serve different roles. The continuous portion of a model is often an assumption about how the world behaves, and the discrete portion of a model is derived from these equations and the safety property. For this reason, many of our updates inductively synthesize ODEs (i.e., in response to data from previous executions of the system) and then deductively synthesize control logic from the resulting ODEs and the safety objective.

Our contributions enabling verifiably safe off-model learning include: **(*1*)** A set of verification preserving model updates (VPMUs) that systematically

update differential equations, control software, and safety proofs in a way that preserves verification guarantees while taking into account possible deviations between an initial model and future system behavior. **(2)** A reinforcement learning algorithm, called model update learning (μlearning), that explains how to transfer safety proofs for a set of feasible models to a learned policy. The learned policy will actively attempt to falsify models at runtime in order to reduce the safety constraints on actions. These contributions are evaluated on a set of hybrid systems control tasks. Our approach uses a combination of program repair, system identification, offline theorem proving, and model monitors to obtain formal safety guarantees for systems in which a single accurate model is not known at design time. This paper fully develops an approach based on an idea that was first presented in an invited vision paper on Safe AI for CPS by the authors [13].

The approach described in this paper is model-based but does not assume that a single correct model is known at design time. Model update learning allows for the possibility that all we can know at design time is that there are many feasible models, one of which might be accurate. Verification-preserving model updates then explain how a combination of data and theorem proving can be used at design time to enrich the set of feasible models.

We believe there is a rich space of approaches toward safe learning in-between model-free reinforcement learning (where formal safety guarantees are unavailable) and traditional model-based learning that assumes the existence of a single ideal model. This paper provides a first example of such an approach by leveraging inductive data and deductive proving at both design time and runtime.

The remainder of this paper is organized as follows. We first review the logical foundations underpinning our approach. We then introduce verification-preserving model updates and discuss how experimental data may be used to construct a set of explanatory models for the data. After discussing several model updates, we introduce the μlearning algorithm that selects between models at runtime. Finally, we discuss case studies that validate both aspects of our approach. We close with a discussion of related work.

2 Background

This section reviews existing approaches toward safe on-model learning and discusses the fitness of each approach for obtaining guarantees about off-model learning. We then introduce the specification language and logic used throughout the rest of this paper.

Alshiekh et al. and Hasanbeig et al. propose approaches toward safe reinforcement learning based on Linear Temporal Logic [3,17]. Alshiekh et al. synthesize monitoring conditions based upon a safety specification and an environmental abstraction. In this formalism, the goal of off-model learning is to systematically expand the environmental abstraction based upon both design-time insights about how the system's behavior might change over time and based upon observed data at runtime. Jansen et al. extend the approach of Alshiekh et al. by observing that constraints should adapt whenever runtime data suggests that a

safety constraint is too restrictive to allow progress toward an over-arching objective [20]. Herbert et al. address the problem of safe motion planning by using offline reachability analysis of pursuit-evasion games to pre-compute an overapproximate monitoring condition that then constrains online planners [9,19].

The above-mentioned approaches have an implicit or explicit environmental model. Even when these environmental models are accurate, reinforcement learning is still necessary because these models focus exclusively on safety and are often nondeterministic. Resolving this nondeterminism in a way that is not only safe but is also effective at achieving other high-level objectives is a task that is well-suited to reinforcement learning.

We are interested in how to provide formal safety guarantees even when there is not a single accurate model available at design time. Achieving this goal requires two novel contributions. We must first find a way to generate a robust set of feasible models given some combination of an initial model and data on previous runs of the system (because formal safety guarantees are stated with respect to a model). Given such a set of feasible models, we must then learn how to safely identify which model is most accurate so that the system is not over-constrained at runtime.

To achieve these goals, we build on the safe learning work for a single model by Fulton et al. [14]. We choose this approach as a basis for verifiably safe learning because we are interested in safety-critical systems that combine discrete and continuous dynamics, because we would like to produce explainable models of system dynamics (e.g., systems of differential equations as opposed to large state machines), and, most importantly, because our approach requires the ability to systematically modify a model together with that model's safety proof.

Following [14], we recall Differential Dynamic Logic [26,27], a logic for verifying properties about safety-critical hybrid systems control software, the ModelPlex synthesis algorithm in this logic [25], and the KeYmaera X theorem prover [12] that will allow us to systematically modify models and proofs together.

Hybrid (dynamical) systems [4,27] are mathematical models that incorporate both discrete and continuous dynamics. Hybrid systems are excellent models for safety-critical control tasks that combine the discrete dynamics of control software with the continuous motion of a physical system such as an aircraft, train, or automobile. Hybrid programs [26–28] are a programming language for hybrid systems. The syntax and informal semantics of hybrid programs is summarized in Table 1. The continuous evolution program is a continuous evolution along the differential equation system $x_i' = \theta_i$ for an arbitrary duration within the region described by formula F.

Hybrid Program Semantics. The semantics of the hybrid programs described by Table 1 are given in terms of transitions between states [27,28], where a state s assigns a real number $s(x)$ to each variable x. We use $s[\![t]\!]$ to refer to the value of a term t in a state s. The semantics of a program α, written $[\![\alpha]\!]$, is the set of pairs (s_1, s_2) for which state s_2 is reachable by running α from state s_1. For example, $[\![x := t_1 \cup x := t_2]\!]$ is:

$$\{(s_1, s_2) \,|\, s_1{=}s_2 \text{ except } s_2(x){=}s_1[\![t_1]\!]\} \cup \{(s_1, s_2) \,|\, s_1{=}s_2 \text{ except } s_2(x){=}s_1[\![t_2]\!]\}$$

Table 1. Hybrid programs

Program statement	Meaning
$\alpha;\beta$	Sequentially composes β after α.
$\alpha \cup \beta$	Executes either α or β nondeterministically.
α^*	Repeats α zero or more times nondeterministically.
$x := \theta$	Evaluates term θ and assigns result to variable x.
$x := *$	Nondeterministically assign arbitrary real value to x.
$\{x'_1 = \theta_1, ..., x'_n = \theta_n \& F\}$	Continuous evolution for any duration within domain F.
$?F$	Aborts if formula F is not true.

for a hybrid program α and state s where $[\![\alpha]\!](s)$ is set of all states t such that $(s,t) \in [\![\alpha]\!]$.

Differential Dynamic Logic. Differential dynamic logic ($\mathsf{d}\mathcal{L}$) [26–28] is the dynamic logic of hybrid programs. The logic associates with each hybrid program α modal operators $[\alpha]$ and $\langle\alpha\rangle$, which express state reachability properties of α. The formula $[\alpha]\phi$ states that the formula ϕ is true in *all* states reachable by the hybrid program α, and the formula $\langle\alpha\rangle\phi$ expresses that the formula ϕ is true after *some* execution of α. The $\mathsf{d}\mathcal{L}$ formulas are generated by the grammar

$$\phi ::= \theta_1 \backsim \theta_2 \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x\, \phi \mid \exists x\, \phi \mid [\alpha]\phi \mid \langle\alpha\rangle\phi$$

where θ_i are arithmetic expressions over the reals, ϕ and ψ are formulas, α ranges over hybrid programs, and $\backsim$ is a comparison operator $=, \neq, \geq, >, \leq, <$. The quantifiers quantify over the reals. We denote by $s \models \phi$ the fact that formula ϕ is true in state s; e.g., we denote by $s \models [\alpha]\phi$ the fact that $(s,t) \in [\![\alpha]\!]$ implies $t \models \phi$ for all states t. Similarly, $\vdash \phi$ denotes the fact that ϕ has a proof in $\mathsf{d}\mathcal{L}$. When ϕ is true in every state (i.e., valid) we simply write $\models \phi$.

Example 1 (Safety specification for straight-line car model).

$$\underbrace{v{\geq}0 \wedge A{>}0}_{\text{initial condition}} \rightarrow [\big(\underbrace{(a{:=}A \cup a{:=}0)}_{ctrl}; \underbrace{\{p'{=}v,\, v'{=}a\}}_{plant}\big)^*] \underbrace{v{\geq}0}_{\text{post cond.}}$$

This formula states that if a car begins with a non-negative velocity, then it will also always have a non-negative velocity after repeatedly choosing new acceleration (A or 0), or coasting and moving for a nondeterministic period of time.

Throughout this paper, we will refer to sets of actions. An **action** is simply the effect of a loop-free deterministic discrete program without tests. For example, the programs $a{:=}A$ and $a{:=}0$ are the actions available in the above program. Notice that **actions** can be equivalently thought of as mappings from variables to terms. We use the term action to refer to both the mappings themselves and the hybrid programs whose semantics correspond to these mappings. For an action u, we write $u(s)$ to mean the effect of taking action u in state s; i.e., the unique state t such that $(s,t) \in [\![u]\!]$.

ModelPlex. Safe off-model learning requires noticing when a system deviates from model assumptions. Therefore, our approach depends upon the ability to check, at runtime, whether the current state of the system can be explained by a hybrid program.

The KeYmaera X theorem prover implements the ModelPlex algorithm [25]. For a given dℒ specification ModelPlex constructs a correctness proof for monitoring conditions expressed as a formula of quantifier-free real arithmetic. The monitoring condition is then used to extract provably correct monitors that check whether observed transitions comport with modeling assumptions. ModelPlex can produce monitors that enforce models of control programs as well as monitors that check whether the model's ODEs comport with observed state transitions.

ModelPlex *controller monitors* are boolean functions that return false if the controller portion of a hybrid systems model has been violated. A *controller monitor* for a model $\{\texttt{ctrl};\texttt{plant}\}^*$ is a function $\texttt{cm} : \mathcal{S} \times \mathcal{A} \to \mathbb{B}$ from states $\mathcal{S}$ and actions $\mathcal{A}$ to booleans $\mathbb{B}$ such that if $\texttt{cm}(s, a)$ then $(s, a(s)) \in [\![ctrl]\!]$. We sometimes also abuse notation by using controller monitors as an implicit filter on $\mathcal{A}$; i.e., $\texttt{cm} : \mathcal{S} \to \mathcal{A}$ such that $a \in \texttt{cm}(s)$ iff $\texttt{cm}(s, a)$ is true.

ModelPlex also produces *model monitors*, which check whether the model is accurate. A *model monitor* for a safety specification $\phi \to [\alpha^*]\psi$ is a function $\texttt{mm} : \mathcal{S} \times \mathcal{S} \to \mathbb{B}$ such that $(s_0, s) \in [\![\alpha]\!]$ if $\texttt{mm}(s_0, s)$. For the sake of brevity, we also define $\texttt{mm} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{B}$ as the model monitor applied after taking an action $(a \in A)$ in a state and then following the plant in a model of form $\alpha \equiv \texttt{ctrl};\texttt{plant}$. Notice that if the model has this canonical form and if if $\texttt{mm}(s, a, a(s))$ for an action a, then $\texttt{cm}(s, a(s))$.

The KeYmaera X system is a theorem prover [12] that provides a language called Bellerophon for scripting proofs of dℒ formulas [11]. Bellerophon programs, called tactics, construct proofs of dℒ formulas. This paper proposes an approach toward updating models in a way that preserves safety proofs. Our approach simultaneously changes a system of differential equations, control software expressed as a discrete loop-free program, and the formal proof that the controller properly selects actuator values such that desired safety constraints are preserved throughout the flow of a system of differential equations.

3 Verification-Preserving Model Updates

A *verification-preserving model update* (VPMU) is a transformation of a hybrid program accompanied by a proof that the transformation preserves key safety properties [13]. VPMUs capture situations in which a model and/or a set of data can be updated in a way that captures possible runtime behaviors which are not captured by an existing model.

Definition 1 (VPMU). *A* verification-preserving model update *is a mapping which takes as input an initial* dℒ *formula* φ *with an associated Bellerophon tactic* `e` *of* φ*, and produces as output a new* dℒ *formula* ψ *and a new Bellerophon tactic* `f` *such that* `f` *is a proof of* ψ*.*

Before discussing our VPMU library, we consider how a set of feasible models computed using VPMUs can be used to provide verified safety guarantees for a family of reinforcement learning algorithms. The primary challenge is to maintain safety with respect to all feasible models while also avoiding overly conservative monitoring constraints. We address this challenge by falsifying some of these models at runtime.

4 Verifiably Safe RL with Multiple Models

VPMUs may be applied whenever system designers can characterize likely ways in which an existing model will deviate from reality. Although applying model updates at runtime is possible and sometimes makes sense, model updates are easiest to apply at design time because of the computational overhead of computing both model updates and corresponding proof updates. This section introduces model update learning, which explains how to take a set of models generated using VPMUs at design time to provide safety guarantees at runtime.

Model update learning is based on a simple idea: begin with a set of *feasible models* and act safely with respect to all feasible models. Whenever a model does not comport with observed dynamics, the model becomes infeasible and is therefore removed from the set of feasible models. We introduce two variations of μlearning: a basic algorithm that chooses actions without considering the underlying action space, and an algorithm that prioritizes actions that rule out feasible models (adding an *eliminate* choice to the classical explore/exploit tradeoff [32]).

All μlearning algorithms use monitored models; i.e., models equipped with ModelPlex controller monitors and model monitors.

Definition 2 (Monitored Model). *A* ***monitored model*** *is a tuple* (m, cm, mm) *such that* m *is a* dℒ *formula of the form*

$$\mathit{init} \rightarrow [\{\mathtt{ctrl};\mathtt{plant}\}^*]\mathit{safe}$$

where `ctrl` *is a loop-free program, the entire formula* m *contains exactly one modality, and the formulas* cm *and* mm *are the control monitor and model monitor corresponding to* m*, as defined in Sect. 2.*

Monitored models may have a continuous action space because of both tests and the nondeterministic assignment operator. We sometimes introduce additional assumptions on the structure of the monitored models. A monitored model over a finite action space is a monitored model where $\{t : (s,t) \in [\![\mathtt{ctrl}]\!]\}$ is finite for all $s \in S$. A time-aware monitored model is a monitored model whose differential equations contain a local clock which is reset at each control step.

Model update learning, or μlearning, leverages verification-preserving model updates to maintain safety while selecting an appropriate environmental model. We now state and prove key safety properties about the μlearning algorithm.

Definition 3 (μlearning Process). *A learning process P_M for a finite set of monitored models M is defined as a tuple of countable sequences $(\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{Mon})$ where $\boldsymbol{U}$ are actions in a finite set of actions $\mathcal{A}$ (i.e., mappings from variables to values), elements of the sequence $\boldsymbol{S}$ are states, and $\boldsymbol{Mon}$ are monitored models with $\boldsymbol{Mon}_0 = M$. Let $specOK_m(\boldsymbol{U}, \boldsymbol{S}, i) \equiv mm(\boldsymbol{S}_{i-1}, \boldsymbol{U}_{i-1}, \boldsymbol{S}_i) \rightarrow cm(\boldsymbol{S}_i, \boldsymbol{U}_i)$ where cm and mm are the monitors corresponding to the model m. Let $specOK$ always return true for $i = 0$.*

*A μ**learning process** is a learning process satisfying the following additional conditions:* ***(a)*** *action availability: in each state $\boldsymbol{S}_i$ there is at least one action u such that for all $m \in \boldsymbol{Mon}_i$, $u \in \mathtt{specOK}_m(\boldsymbol{U}, \boldsymbol{S}, i)$,* ***(b)*** *actions are safe for all feasible models: $\boldsymbol{U}_{i+1} \in \{u \in A \,|\, \forall (m, cm, mm) \in \boldsymbol{Mon}_i, cm(\boldsymbol{S}_i, u)\}$,* ***(c)*** *feasible models remain in the feasible set: if $(\varphi, cm, mm) \in \boldsymbol{Mon}_i$ and $mm(\boldsymbol{S}_i, \boldsymbol{U}_i, \boldsymbol{S}_{i+1})$ then $(\varphi, cm, mm) \in \boldsymbol{Mon}_{i+1}$.*

Note that μlearning processes are defined over an environment $E : \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{S}$ that determines the sequences $\mathbf{U}$ and $\mathbf{S}$[1], so that $\mathbf{S}_{i+1} = E(\mathbf{U}_i, \mathbf{S}_i)$. In our algorithms, the set $\mathbf{Mon}_i$ never retains elements that are inconsistent with the observed dynamics at the previous state. We refer to the set of models in $\mathbf{Mon}_i$ as the set of feasible models for the i^{th} state in a μlearning process.

Notice that the safe actions constraint is not effectively checkable without extra assumptions on the range of parameters. Two canonical choices are discretizing options for parameters or including an effective identification process for parameterized models.

Our safety theorem focuses on time-aware μlearning processes, i.e., those whose models are all time-aware; similarly, a *finite action space μlearning process* is a μlearning process in which all models $m \in M$ have a finite action space. The basic correctness property for a μlearning process is the safe reinforcement learning condition: the system never takes unsafe actions.

Definition 4 (μlearning process with an accurate model). *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a μlearning process. Assume there is some element $m^* \in \boldsymbol{Mon}_0$ with the following properties. First,*

$$m^* \equiv (\mathtt{init}_m \rightarrow [\{\mathtt{ctrl}_m; \mathtt{plant}_m\}^*]\mathtt{safe}).$$

Second, $\vdash m^*$. Third, $(s, u(s)) \in [\![\mathtt{ctrl}_m]\!]$ implies $(u(s), E(u, s)) \in [\![plant]\!]$ for a mapping $E : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ from states and actions to new states called environment. When only one element of $\boldsymbol{Mon}_0$ satisfies these properties we call that element m^* the distinguished *and/or* accurate *model and say that the process P_M is* accurately modeled with respect to *E.*

We will often elide the environment E for which the process P_M is accurate when it is obvious from context.

Theorem 1 (Safety). *If P_M is a μlearning process with an accurate model, then $\boldsymbol{S}_i \models \mathtt{safe}$ for all $0 < i < |\boldsymbol{S}|$.*

[1] Throughout the paper, we denote by $\mathbf{S}$ a specific sequence of states and by $\mathcal{S}$ the set of all states.

Listing 1.1 presents the μlearning algorithm. The inputs are: **(*a*)** A set `M` of models each with an associated function `m.models` : $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{B}$ that implements the evaluation of its model monitor in the given previous and next state and actions and a method `m.safe` : $\mathcal{S} \times \mathcal{A} \to \mathbb{B}$ which implements evaluation of its controller monitor, **(*b*)** an action space `A` and an initial state `init` $\in S$, **(*c*)** an environment function `env` : $\mathcal{S} \times \mathcal{A} \to \mathcal{S} \times \mathbb{R}$ that computes state updates and rewards in response to actions, and **(*d*)** a function `choose` : $\wp(\mathcal{A}) \to \mathcal{A}$ that selects an action from a set of available actions and `update` updates a table or approximation. Our approach is generic and works for any reinforcement learning algorithm; therefore, we leave these functions abstract. It augments an existing reinforcement learning algorithm, defined by `update` and `choose`, by restricting the action space at each step so that actions are only taken if they are safe with respect to *all* feasible models. The feasible model set is updated at each control set by removing models that are in conflict with observed data.

The μlearning algorithm rules out incorrect models from the set of possible models by taking actions and observing the results of those actions. Through these experiments, the set of relevant models is winnowed down to either the distinguished correct model m^*, or a set of models M^* containing m^* and other models that cannot be distinguished from m^*.

Listing 1.1. The basic μlearning algorithm

```
def μlearn(M,A,init,env,choose,update):
  s_pre = s_curr = init
  act   = None
  while(not done(s_curr)):
    if act is not None:
      M = {m ∈ M : m.models(s_pre,act,s_curr)}
    avail = {a ∈ A : ∀ m ∈ M, m.safe(s_curr, a)}
    act = choose(avail)
    s_pre = s_curr
    (s_curr, reward) = env(s_curr, act)
    update(s_pre, act, s_curr, reward)
```

4.1 Active Verified Model Update Learning

Removing models from the set of possible models relaxes the monitoring condition, allowing less conservative and more accurate control decisions. Therefore, this section introduces an active learning refinement of the μlearning algorithm that prioritizes taking actions that help rule out models $m \in M$ that are not m^*. Instead of choosing a random safe action, μlearning prioritizes actions that differentiate between available models. We begin by explaining what it means for an algorithm to perform good experiments.

Definition 5 (Active Experimentation). *A μlearning process with an accurate model m^* has* locally active experimentation *provided that: if* $\boldsymbol{Mon}_i > 1$

and there exists an action a that is safe for all feasible models (see Definition 3) in state s_i such that taking action a results in the removal of m from the model set[2]*, then $|\boldsymbol{Mon}_{i+1}| < |\boldsymbol{Mon}_i|$. Experimentation is* `er`-active *if the following conditions hold: there exists an action a that is safe for all feasible models (see Definition 3) in state s_i, and taking action a resulted in the removal of m from the model set, then $|\boldsymbol{Mon}_{i+1}| < |\boldsymbol{Mon}_i|$ with probability $0 < \texttt{er} < 1$.*

Definition 6 (Distinguishing Actions). *Consider a μlearning process $(\boldsymbol{U}, \boldsymbol{S}, \boldsymbol{Mon})$ with an accurate model m^* (see Definition 4). An action a* distinguishes *m from m^* if $a = \boldsymbol{U}_i$, $m \in \boldsymbol{Mon}_i$ and $m \notin \boldsymbol{Mon}_{i+1}$ for some $i > 0$.*

The *active μlearning algorithm* uses model monitors to select distinguishing actions, thereby performing active experiments which winnow down the set of feasible models. The inputs to `active-`μ`learn` are the same as those to Listing 1.1 with two additions: **(*1*)** models are augmented with an additional prediction method `p` that returns the model's prediction of the next state given the current state, a candidate action, and a time duration. **(*2*)** An elimination rate `er` is introduced, which plays a similar role as the classical explore-exploit rate except that we are now deciding whether to insist on choosing a good experiment. The `active-`μ`learn` algorithm is guaranteed to make some progress toward winnowing down the feasible model set whenever $0 < \texttt{er} < 1$.

Theorem 2. *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a finite action space μlearning process with an accurate model m^*. Then $m^* \in \boldsymbol{Mon}_i$ for all $0 \leq i \leq |\boldsymbol{Mon}|$.*

Theorem 3. *Let P_M be a finite action space* `er`*-active μlearning process under environment E and with an accurate model m^*. Consider any model $m \in \boldsymbol{Mon}_0$ such that $m \neq m^*$. If every state s has an action a_s that is safe for all models and distinguishes m from m^*, then $\lim_{i\to\infty} Pr(m \notin \boldsymbol{Mon}_i) = 1$.*

Corollary 1. *Let $P_M = (\boldsymbol{S}, \boldsymbol{U}, \boldsymbol{Mon})$ be a finite action space* `er`*-active μlearning process under environment E and with an accurate model m^*. If* each *model $m \in \boldsymbol{Mon}_0 \setminus \{m^*\}$ has in each state s an action a_s that is safe for all models and distinguishes m from m^*, then $\boldsymbol{Mon}$ converges to $\{m^*\}$ a.s.*

Although locally active experimentation is not strong enough to ensure that P_M eventually converges to a minimal set of models[3], our experimental validation demonstrates that this heuristic is none-the-less effective on some representative examples of model update learning problems.

5 A Model Update Library

So far, we have established how to obtain safety guarantees for reinforcement learning algorithms given a set of formally verified $\mathsf{d}\mathcal{L}$ models. We now turn

[2] We say that taking action a_i in state s_i results in the removal of a model m from the model set if $m \in \mathbf{Mon}_i$ but $m \notin \mathbf{Mon}_{i+1}$.

[3] $x \geq 0 \wedge t = 0 \rightarrow [\{\{?t = 0; x := 1 \cup x := 0\}; \{x' = F, t' = 1\}\}^*]\, x \geq 0$ with the parameters $F = 0, F = 5,$ and $F = x$ are a counter example [10, Section 8.4.4].

our attention to the problem of generating such a set of models by systematically modifying dℒ formulas and their corresponding Bellerophon tactical proof scripts. This section introduces five generic model updates that provide a representative sample of the kinds of computations that can be performed on models and proofs to predict and account for runtime model deviations[4].

The simplest example of a VPMU instantiates a parameter whose value is not known at design time but can be determined at runtime via system identification. Consider a program p modeling a car whose acceleration depends upon both a known control input *accel* and parametric values for maximum braking force $-B$ and maximum acceleration A. Its proof is

$$\texttt{implyR}(1);\texttt{loop}(\texttt{pos} - \texttt{obsPos} > \frac{\texttt{vel}^2}{2B}, 1);\texttt{onAll}(\texttt{master})$$

This model and proof can be updated with concrete experimentally determined values for each parameter by uniformly substituting the variables B and A with concrete values in both the model and the tactic.

The **Automatic Parameter Instantiation** update improves the basic parameter instantiation update by automatically detecting which variables are parameters and then constraining instantiation of parameters by identifying relevant initial conditions.

The **Replace Worst-Case Bounds with Approximations** update improves models designed for the purpose of safety verification. Often a variable occurring in the system is bounded above (or below) by its worst-case value. Worst-case analyses are sufficient for establishing safety but are often overly conservative. The approximation model update replaces worst-case bounds with approximate bounds obtained via series expansions. The proof update then introduces a tactic on each branch of the proof that establishes our approximations are upper/lower bounds by performing.

Models often assume perfect sensing and actuation. A common way of robustifying a model is to add a piecewise constant noise term to the system's dynamics. Doing so while maintaining safety invariants requires also updating the controller so that safety envelope computations incorporate this noise term. The **Add Disturbance Term** update introduces noise terms to differential equations, systematically updates controller guards, and modifies the proof accordingly.

Uncertainty in object classification is naturally modeled in terms of sets of feasible models. In the simplest case, a robot might need to avoid an obstacle that is either static, moves in a straight line, or moves sinusoidally. Our generic model update library contains an update that changes the model by making a static point (x, y) dynamic. For example, one such update introduces the equations $\{x' = -y, y' = -x\}$ to a system of differential equations in which the variables x, y do not have differential equations. The controller is updated so that any statements about separation between (a, b) and (x, y) require global separation of (a, b) from the circle on which (x, y) moves. The proof is also updated by

[4] Extended discussion of these model updates is available in [10, Chapters 8 and 9].

prepending to the first occurrence of a differential tactic on each branch with a sequence of differential cuts that characterize circular motion.

Model updates also provide a framework for characterizing algorithms that combine model identification and controller synthesis. One example is our synthesis algorithm for systems whose ODEs have solutions in a decidable fragment of real arithmetic (a subset of linear ODEs). Unlike other model updates, we do not assume that any initial model is provided; instead, we learn a model (and associated control policy) entirely from data. The **Learn Linear Dynamics** update takes as input: (1) data from previous executions of the system, and (2) a desired safety constraint. From these two inputs, the update computes a set of differential equations `odes` that comport with prior observations, a corresponding controller `ctrl` that enforces the desired safety constraint with corresponding initial conditions `init`, and a Bellerophon tactic `prf` which proves `init` $\rightarrow$ [{`ctrl`; `odes`}*]`safe` . Computing the model requires an exhaustive search of the space of possible ODEs followed by a computation of a safe control policy using solutions to the resulting ODEs. Once a correct controller is computed, the proof proceeds by symbolically decomposing the control program and solving the ODEs on each resulting control branch. The full mechanism is beyond the scope of this paper but explained in detail elsewhere [10, Chapter 9].

Significance of Selected Updates. The updates described in this section demonstrate several possible modes of use for VPMUs and μlearning. VPMUS can update existing models to account for systematic modeling errors (e.g., missing actuator noise or changes in the dynamical behavior of obstacles). VPMUs can automatically optimize control logic in a proof-preserving fashion. VPMUS can also be used to generate accurate models and corresponding controllers from experimental data made available at design time, without access to any prior model of the environment.

6 Experimental Validation

The μlearning algorithms introduced in this paper are designed to answer the following question: given a set of possible models that contains the one true model, how can we *safely* perform a set of experiments that allow us to efficiently discover a minimal safety constraint? In this section we present two experiments which demonstrate the use of μlearning in safety-critical settings. Overall, these experiments empirically validate our theorems by demonstrating that μlearning processes with accurate models do not violate safety constraints.

Our simulations use a conservative discretization of the hybrid systems models, and we translated monitoring conditions by hand into Python from ModelPlex's C output. Although we evaluate our approach in a research prototype implemented in Python for the sake of convenience, there is a verified compilation pipeline for models implemented in $\mathsf{d}\mathcal{L}$ that eliminates uncertainty introduced by discretization and hand-translations [7].

Adaptive Cruise Control. Adaptive Cruise Control (ACC) is a common feature in new cars. ACC systems change the speed of the car in response to the changes in the speed of traffic in front of the car; e.g., if the car in front of an ACC-enabled car begins slowing down, then the ACC system will decelerate to match the velocity of the leading car. Our first set of experiments consider a simple linear model of ACC in which the acceleration set-point is perturbed by an unknown parameter p; i.e., the relative position of the two vehicles is determined by the equations $\text{pos}'_{\text{rel}} = \text{vel}_{\text{rel}}, \text{vel}'_{\text{rel}} = \text{acc}_{\text{rel}}$.

In [14], the authors consider the collision avoidance problem when a noise term is added so that $\text{vel}'_{\text{rel}} = p\text{acc}_{\text{rel}}$. We are able to outperform the approach in [14] by combining the **Add Noise Term** and **Parameter Instantiation** updates; we outperform in terms of both avoiding unsafe states and in terms of cumulative reward. These two updates allow us to insert a multiplicative noise term p into these equations, synthesize a provably correct controller, and then choose the correct value for this noise term at runtime. Unlike [14], μlearning avoids all safety violations. The graph in Fig. 1 compares the Justified Speculative Control approach of [14] to our approach in terms of cumulative reward; in addition to substantially outperforming the JSC algorithm of [14], μlearning also avoids 204 more crashes throughout a 1,000 episode training process.

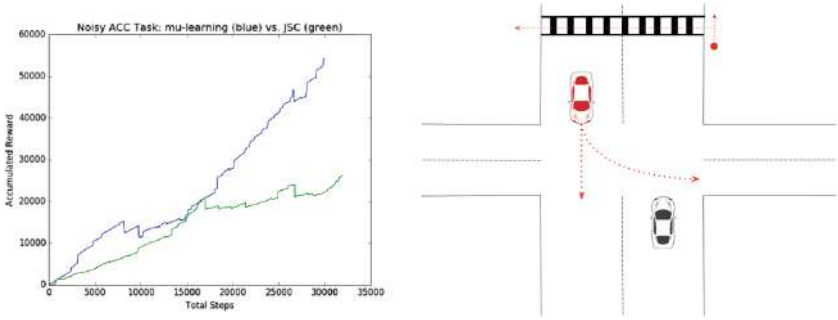


Fig. 1. Left: The cumulative reward obtained by Justified Speculative Control [14] (green) and μlearning (blue) during training over 1,000 episodes with each episode truncated at 100 steps. Each episode used a randomly selected error term that remains constant throughout each episode but may change between episodes. Right: a visualization of the hierarchical safety environment. (Color figure online)

A Hierarchical Problem. Model update learning can be extended to provide formal guarantees for hierarchical reinforcement learning algorithms [6]. If each feasible model m corresponds to a subtask, and if all states satisfying termination conditions for subtask m_i are also safe initial states for any subtask m_j reachable from m_i, then μlearning directly supports safe hierarchical reinforcement learning by re-initializing M to the initial (maximal) model set whenever reaching a termination condition for the current subtask.

We implemented a variant of μlearning that performs this re-initialization and validated this algorithm in an environment where a car must first navigate an intersection containing another car and then must avoid a pedestrian in a crosswalk (as illustrated in Fig. 1). In the crosswalk case, the pedestrian at (ped_x, ped_y) may either continue to walk along a sidewalk indefinitely or may enters the crosswalk at some point between $c_{min} \leq ped_y \leq c_{max}$ (the boundaries of the crosswalk). This case study demonstrates that safe hierarchical reinforcement learning is simply safe μlearning with safe model re-initialization.

7 Related Work

Related work falls into three broad categories: safe reinforcement learning, runtime falsification, and program synthesis.

Our approach toward safe reinforcement learning differs from existing approaches that do not include a formal verification component (e.g., as surveyed by García and Fernández [15] and the SMT-based constrained learning approach of Junges et al. [21]) because we focused on *verifiably* safe learning; i.e., instead of relying on oracles or conjectures, constraints are derived in a provably correct way from formally verified safety proofs. The difference between verifiably safe learning and safe learning is significant, and is equivalent to the difference between verified and unverified software. Unlike most existing approaches our safety guarantees apply to both the learning process and the final learned policy.

Section 2 discusses how our work relates to the few existing approaches toward *verifiably* safe reinforcement learning. Unlike those [3,14,17,20], as well as work on model checking and verification for MDPs [18], we introduce an approach toward verifiably safe off-model learning. Our approach is the first to combine model synthesis at design time with model falsification at runtime so that safety guarantees capture a wide range of possible futures instead of relying on a single accurate environmental model. Safe off-model learning is an important problem because autonomous systems must be able to cope with unanticipated scenarios. Ours is the first approach toward verifiably safe off-model learning.

Several recent papers focus on providing safety guarantees for model-free reinforcement learning. Trust Region Policy Optimization [31] defines safety as monotonic policy improvement, a much weaker notion of safety than the constraints guaranteed by our approach. Constrained Policy Optimization [1] extends TRPO with guarantees that an agent nearly satisfies safety constraints during learning. Brázdil et al. [8] give probabilistic guarantees by performing a heuristic-driven exploration of the model. Our approach is model-based instead of model-free, and instead of focusing on learning safely without a model we focus on identifying accurate models from data obtained both at design time and at runtime. Learning concise dynamical systems representations has one substantial advantage over model-free methods: safety guarantees are stated with respect to an explainable model that captures the safety-critical assumptions about the system's dynamics. Synthesizing explainable models is important because safety guarantees are always stated with respect to a model; therefore, engineers must

be able to understand inductively synthesized models in order to understand what safety properties their systems do (and do not) ensure.

Akazaki et al. propose an approach, based on deep reinforcement learning, for efficiently discovering defects in models of cyber-physical systems with specifications stated in signal temporal logic [2]. Model falsification is an important component of our approach; however, unlike Akazaki et al., we also propose an approach toward obtaining more robust models and explain how runtime falsification can be used to obtain safety guarantees for off-model learning.

Our approach includes a model synthesis phase that is closely related to program synthesis and program repair algorithms [23,24,29]. Relative to work on program synthesis and repair, VPMUs are unique in several ways. We are the first to explore *hybrid* program repair. Our approach combines program verification with mutation. We treat programs as *models* in which one part of the model is varied according to interactions with the environment and another part of the model is systematically derived (together with a correctness proof) from these changes. This separation of the dynamics into inductively synthesized models and deductively synthesized controllers enables our approach toward using programs as representations of dynamic safety constraints during reinforcement learning.

Although we are the first to explore hybrid program repair, several researchers have explored the problem of synthesizing hybrid systems from data [5,30]. This work is closely related to our **Learn Linear Dynamics** update. Sadraddini and Belta provide formal guarantees for data-driven model identification and controller synthesis [30]. Relative to this work, our **Learn Linear Dynamics** update is continuous-time, synthesizes a computer-checked correctness proof but does not consider the full class of linear ODEs. Unlike Asarin et al. [5], our full set of model updates is sometimes capable of synthesizing nonlinear dynamical systems from data (e.g., the static $\rightarrow$ circular update) and produces computer-checked correctness proofs for permissive controllers.

8 Conclusions

This paper introduces an approach toward verifiably safe off-model learning that uses a combination of design-time verification-preserving model updates and runtime model update learning to provide safety guarantees even when there is no single accurate model available at design time. We introduced a set of model updates that capture common ways in which models can deviate from reality, and introduced an update that is capable of synthesizing ODEs and provably correct controllers without access to an initial model. Finally, we proved safety and efficiency theorems for active μlearning and evaluated our approach on some representative examples of hybrid systems control tasks. Together, these contributions constitute a first approach toward verifiably safe off-model learning.

References

1. Achiam, J., Held, D., Tamar, A., Abbeel, P.: Constrained policy optimization. In: Precup, D., Teh, Y.W. (eds.) Proceedings of the 34th International Conference on Machine Learning (ICML 2017), Proceedings of Machine Learning Research, vol. 70, pp. 22–31. PMLR (2017)
2. Akazaki, T., Liu, S., Yamagata, Y., Duan, Y., Hao, J.: Falsification of cyber-physical systems using deep reinforcement learning. In: Havelund, K., Peleska, J., Roscoe, B., de Vink, E. (eds.) FM 2018. LNCS, vol. 10951, pp. 456–465. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-95582-7_27
3. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: McIlraith, S.A., Weinberger, K.Q (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018). AAAI Press (2018)
4. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H. (eds.) HS 1991–1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-57318-6_30
5. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proc. IEEE **88**(7), 1011–1025 (2000)
6. Barto, A.G., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discret. Event Dyn. Syst. **13**(1–2), 41–77 (2003)
7. Bohrer, B., Tan, Y.K., Mitsch, S., Myreen, M.O., Platzer, A.: VeriPhy: verified controller executables from verified cyber-physical system models. In: Grossman, D. (ed.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018), pp. 617–630. ACM (2018)
8. Brázdil, T., et al.: Verification of Markov decision processes using learning algorithms. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 98–114. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11936-6_8
9. Fridovich-Keil, D., Herbert, S.L., Fisac, J.F., Deglurkar, S., Tomlin, C.J.: Planning, fast and slow: a framework for adaptive real-time safe trajectory planning. In: IEEE International Conference on Robotics and Automation (ICRA), pp. 387–394 (2018)
10. Fulton, N.: Verifiably safe autonomy for cyber-physical systems. Ph.D. thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University (2018)
11. Fulton, N., Mitsch, S., Bohrer, B., Platzer, A.: Bellerophon: tactical theorem proving for hybrid systems. In: Ayala-Rincón, M., Muñoz, C.A. (eds.) ITP 2017. LNCS, vol. 10499, pp. 207–224. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66107-0_14
12. Fulton, N., Mitsch, S., Quesel, J.-D., Völp, M., Platzer, A.: KeYmaera X: an axiomatic tactical theorem prover for hybrid systems. In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 527–538. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_36
13. Fulton, N., Platzer, A.: Safe AI for CPS (invited paper). In: IEEE International Test Conference (ITC 2018) (2018)
14. Fulton, N., Platzer, A.: Safe reinforcement learning via formal methods: toward safe control through proof and learning. In: McIlraith, S., Weinberger, K. (eds.) Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018), pp. 6485–6492. AAAI Press (2018)

15. García, J., Fernández, F.: A comprehensive survey on safe reinforcement learning. J. Mach. Learn. Res. **16**, 1437–1480 (2015)
16. Ghosh, S., Berkenkamp, F., Ranade, G., Qadeer, S., Kapoor, A.: Verifying controllers against adversarial examples with Bayesian optimization. CoRR abs/1802.08678 (2018)
17. Hasanbeig, M., Abate, A., Kroening, D.: Logically-correct reinforcement learning. CoRR abs/1801.08099 (2018)
18. Henriques, D., Martins, J.G., Zuliani, P., Platzer, A., Clarke, E.M.: Statistical model checking for Markov decision processes. In: QEST, pp. 84–93. IEEE Computer Society (2012). https://doi.org/10.1109/QEST.2012.19
19. Herbert, S.L., Chen, M., Han, S., Bansal, S., Fisac, J.F., Tomlin, C.J.: FaSTrack: a modular framework for fast and guaranteed safe motion planning. In: IEEE Annual Conference on Decision and Control (CDC)
20. Jansen, N., Könighofer, B., Junges, S., Bloem, R.: Shielded decision-making in MDPs. CoRR abs/1807.06096 (2018)
21. Junges, S., Jansen, N., Dehnert, C., Topcu, U., Katoen, J.-P.: Safety-constrained reinforcement learning for MDPs. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 130–146. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_8
22. Kalra, N., Paddock, S.M.: Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?. RAND Corporation, Santa Monica (2016)
23. Kitzelmann, E.: Inductive programming: a survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 50–73. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11931-6_3
24. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2012)
25. Mitsch, S., Platzer, A.: ModelPlex: verified runtime validation of verified cyber-physical system models. Form. Methods Syst. Des. **49**(1), 33–74 (2016). Special issue of selected papers from RV'14
26. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reas. **41**(2), 143–189 (2008)
27. Platzer, A.: Logics of dynamical systems. In: LICS, pp. 13–24. IEEE (2012)
28. Platzer, A.: A complete uniform substitution calculus for differential dynamic logic. J. Autom. Reas. **59**(2), 219–266 (2017)
29. Rothenberg, B.-C., Grumberg, O.: Sound and complete mutation-based program repair. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 593–611. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_36
30. Sadraddini, S., Belta, C.: Formal guarantees in data-driven model identification and control synthesis. In: Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (HSCC 2018), pp. 147–156 (2018)
31. Schulman, J., Levine, S., Abbeel, P., Jordan, M.I., Moritz, P.: Trust region policy optimization. In: Bach, F.R., Blei, D.M. (eds.) Proceedings of the 32nd International Conference on Machine Learning (ICML 2015), JMLR Workshop and Conference Proceedings, vol. 37, pp. 1889–1897 (2015)
32. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)

Correction to: WAPS: Weighted and Projected Sampling

Rahul Gupta, Shubham Sharma, Subhajit Roy, and Kuldeep S. Meel

Correction to:
Chapter "WAPS: Weighted and Projected Sampling"
in: T. Vojnar and L. Zhang (Eds.): *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 11427,
https://doi.org/10.1007/978-3-030-17462-0_4

In the version of this paper that was originally published, there was an error in the acknowledgement at the bottom of the first page. "AI Singapore Grant [R-252-000-A16-490]" was mentioned instead of "National Research Foundation Singapore under its AI Singapore Programme [Award Number: AISG-RP-2018-005]". This has now been corrected.

The updated version of this chapter can be found at
https://doi.org/10.1007/978-3-030-17462-0_4

T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, p. C1, 2019.
https://doi.org/10.1007/978-3-030-17462-0_29

Author Index