

Efficient engineering of supervisory controllers

Citation for published version (APA):

Thuijsman, S. B. (2023). *Efficient engineering of supervisory controllers*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mechanical Engineering]. Eindhoven University of Technology.

Document status and date:

Published: 19/10/2023

Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy

If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Efficient engineering of supervisory controllers

Sander Thuijsman



Sander Thuijsman

Efficient engineering of
supervisory controllers

Department of Mechanical Engineering
EINDHOVEN UNIVERSITY OF TECHNOLOGY
Eindhoven, The Netherlands, 2023

©2023, Sander Thuijsman. All rights reserved.

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-5834-6

Printed by Print Service Ede.

Cover: ©2023, Eva van den Biggelaar.
<https://evavandenbiggelaar.com>

Research leading to these results has received funding from the EU ECSEL Joint Undertaking under grant agreement n° 826452 (project Arrowhead Tools) and from the partners' national programs/funding authorities.

Efficient engineering of supervisory controllers

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische
Universiteit Eindhoven, op gezag van de rector magnificus
prof.dr. S.K. Lenaerts, voor een commissie aangewezen door het
College voor Promoties, in het openbaar te verdedigen op
donderdag 19 oktober 2023 om 13:30 uur

door

Sander Benjamin Thuijsman

geboren te Eijsden

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter: prof.dr.ir. P.D. Anderson
1^e promotor: dr.ir. M.A. Reniers
2^e promotor: prof.dr. K. Cai (Osaka Metropolitan University)
leden: prof.dr. F. Basile (University of Salerno)
prof.dr. M. Fabian (Chalmers University of Technology)
prof.dr.ir. J.P.M. Voeten
dr.ir. T.A.C. Willemse
adviseur: Dr.-Ing. A.K. Schmuck (Max Planck Institute for Software Systems)

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Abstract

Cyber-physical systems consist of a mechanism interacting with the world (physical component), which is monitored and controlled by a computational (cyber) component. Many cyber-physical systems are safety-critical: failures or malfunctions may have serious consequences to people's lives or the environment. Therefore, correct functioning of cyber-physical systems is vital.

A cyber-physical system is steered by a supervisory controller (supervisor). The supervisor is responsible for the high-level control strategy, making sure all components cooperate and the system properly performs its tasks. This thesis contributes to synthesis-based engineering of supervisors. Synthesis-based engineering is centered around the use of mathematical models. These models describe the uncontrolled system behavior (what the system *can* do) and a model of the requirements (what the system *should* (not) do). By applying supervisor synthesis to these models a supervisor is algorithmically obtained. This supervisor is correct-by-construction: when the system is steered by this supervisor, the requirements are always adhered to, and some more desirable behavioral properties are satisfied. The modeling and synthesis framework is called supervisory control theory.

This thesis makes several contributions to synthesis-based engineering. These contributions aid the ease of use, applicability, and efficiency of supervisory control theory. The contributions are toward three separate aspects of supervisory control theory, and are summarized as follows:

1. Transformational approaches in supervisory control.

Typically, cyber-physical systems evolve over time. For example, a new bicycle detection sensor may be added to an existing traffic light system. As a result, the supervisor will progress to several iterations. Consequently, algorithmic computations that are applied to the system during synthesis-based engineering, need to be applied for each iteration. Traditionally, the computations are repeated from scratch for each iteration. These computations may take a long time to complete.

In the transformational approach, results from previous computations are reused in new computations when the system evolves. For example, supervisor synthesis has been

applied for some base system. Now, at a later point in time, the system is modified and a variant system is created. Instead of performing supervisor synthesis from scratch for this variant system, in the transformational approach the supervisor of the base system is transformed into a supervisor for the variant system. Which transformation operations need to be performed, depends on how the system has been modified. In this thesis, transformational approaches are studied for supervisor synthesis and supervisor localization.

2. Supervisory control for product lines.

A product line describes a collection of products that share commonalities, but also allow variability between them. For example a coffee machine that has variants that do or do not have the ability to pour milk or tea. There may be many possible configurations for a product in a product line. In this thesis, we study how to apply synthesis-based engineering for product lines. Feature models are used to represent the system's configurations. Behavior and requirements are modeled, and these are made dependent on the presence of features. The resulting model is suitable for supervisor synthesis, and the obtained supervisor can correctly control the system for all possible configurations. Furthermore, in this work we pay special attention to the case where the system may dynamically reconfigure, i.e., components may enter or leave the system during runtime.

3. Efficient symbolic supervisor synthesis.

When the size of a system grows, the computational effort required for supervisor synthesis grows exponentially. To mitigate this, the system can symbolically be represented using binary decision diagrams (BDDs), and supervisor synthesis can be applied to this symbolic representation. Minor changes in how symbolic synthesis is algorithmically performed, or the settings with which it is initiated, can have a major impact on the required time and memory for the computation. In this thesis, several approaches are studied to efficiently perform symbolic supervisor synthesis. They are: (1) a variable ordering heuristic, that based on the model picks an order in which the variables appear in the BDD, to reduce the BDD size and therefore the required memory; (2) an edge ordering heuristic, that picks an order in which the edges are evaluated during synthesis to reduce the computation time; and (3) an algorithm that efficiently applies state exclusion requirements by restricting transitions prior to synthesis. The methods are supported by elaborate experimental evaluation.

Samenvatting

Cyber-fysieke systemen bestaan uit een mechanisme dat in wisselwerking staat met de wereld (fysiek component), dat wordt gecontroleerd en bestuurd door een computationeel (cyber) component. Veel cyber-fysieke systemen zijn veiligheidskritisch: storingen kunnen ernstige gevolgen hebben voor het leven van mensen of voor het milieu. Daarom is correcte werking van cyber-fysieke systemen van vitaal belang.

Een cyber-fysiek systeem wordt aangestuurd door een supervisor. De supervisor is verantwoordelijk voor de hoog niveau controlestrategie en zorgt ervoor dat alle componenten samenwerken en het systeem zijn taken naar behoren uitvoert. Dit proefschrift draagt bij aan de op synthese gebaseerde ontwikkeling van supervisors. Op synthese gebaseerde ontwikkeling draait om het gebruik van wiskundige modellen. Deze modellen beschrijven het ongecontroleerde gedrag (wat het systeem *kan* doen) en gedragseisen (wat het systeem (niet) *mag* doen). Door supervisor synthese toe te passen op deze modellen, wordt een supervisor algoritmisch verkregen. Wanneer het systeem door deze supervisor wordt aangestuurd, wordt altijd aan de eisen voldaan en ook aan een aantal overige wenselijke gedragseigenschappen wordt voldaan. Het wiskundige raamwerk voor modellering en synthese wordt supervisory control theorie genoemd.

Dit proefschrift levert verscheidene bijdragen aan synthese gebaseerde engineering. Deze bijdragen bevorderen het gebruiksgemak, de toepasbaarheid, en de efficiëntie van supervisory control theorie. De bijdragen hebben betrekking op drie afzonderlijke aspecten van de theorie en zijn als volgt samengevat:

1. Transformationele aanpakken in supervisory control theorie.

Doorgaans evolueren cyber-fysieke systemen in de loop van tijd. Denk bijvoorbeeld aan een nieuwe detectiesensor voor fietsen die toegevoegd wordt aan een bestaand stoplichtensysteem. Dit betekent dat de supervisor meerdere iteraties doorloopt. Als gevolg moeten algoritmische berekeningen voor elke iteratie opnieuw worden toegepast. Deze berekeningen kunnen lang duren en traditioneel worden ze voor elke iteratie helemaal opnieuw uitgevoerd.

In de transformationele aanpak worden resultaten van eerdere berekeningen hergebruikt in nieuwe berekeningen wanneer het systeem evolueert. Bijvoorbeeld, supervisor

synthese is toegepast voor een bepaald basis systeem. Op een later tijdstip wordt het systeem aangepast en ontstaat er een variant systeem. In plaats van supervisor synthese helemaal opnieuw uit te voeren voor dit variant systeem, wordt in de transformationele aanpak de supervisor van het basis systeem getransformeerd in een supervisor voor het variant systeem, afhankelijk van hoe het systeem is aangepast. In dit proefschrift worden transformationele aanpakken bestudeerd voor supervisor synthese en supervisor lokalisatie.

2. Supervisory control voor productlijnen.

Een productlijn beschrijft een verzameling producten die gemeenschappelijke kenmerken hebben, maar waartussen ook variatie mogelijk is. Bijvoorbeeld een koffiema-chine waar varianten van bestaan die wel of niet thee of melk kunnen schenken. Er kunnen veel mogelijke configuraties zijn voor een product in een productlijn. In dit proefschrift bestuderen we hoe op synthese gebaseerde ontwikkeling kan worden toegepast op productlijnen. Feature modellen worden gebruikt om de configuraties in de productlijn te representeren. Gedrag en eisen worden gemodelleerd en afhankelijk gemaakt van de aanwezigheid van componenten. Het resulterende model is geschikt voor supervisor synthese en de verkregen supervisor kan het systeem correct besturen voor alle mogelijke configuraties. Bovendien besteden we in dit werk speciale aandacht aan het geval waarin het systeem dynamisch kan herconfigureren, dat willen zeggen dat componenten tijdens runtime het systeem kunnen binnenkomen of verlaten.

3. Efficiënte symbolische supervisor synthese.

Wanneer de omvang van een systeem groeit, neemt de rekenkracht die nodig is voor supervisor synthese exponentieel toe. Om hiermee om te gaan, kan het systeem symbolisch worden gerepresenteerd met behulp van binaire beslissingsdiagrammen (Engels: binary decision diagrams, BDDs) en kan supervisor synthese worden toegepast op deze symbolische representatie. Kleine veranderingen in de manier waarop het symbolische synthese algoritme wordt uitgevoerd, of in de instellingen waarmee het wordt geïnitieerd, kunnen van grote invloed zijn op de benodigde rekestijd en geheugen. In dit proefschrift worden verschillende benaderingen bestudeerd om symbolische supervisor synthese efficiënt uit te voeren. Dit zijn: (1) een variabele ordeningsheuristiek, die op basis van het model een volgorde kiest waarin de variabelen in de BDD verschijnen, om de BDD-grootte en dus het vereiste geheugen te verkleinen; (2) een transitie ordeningsheuristiek, die een volgorde kiest waarin transities tijdens synthese worden geëvalueerd om de rekestijd te verkorten; en (3) een algoritme dat op efficiënte wijze situatie-uitsluitingseisen toepast door transities voorafgaand aan synthese te beperken. De methodes worden ondersteund door uitgebreide experimentele evaluatie.

Dankwoord

Ondanks dat op de kaft van dit boekje alleen mijn eigen naam staat, zijn er veel mensen die er op directe of indirecte wijze aan hebben bijgedragen. Bij dezen wil ik hen bedanken.

Allereerst, Michel, bedankt voor je enthousiasme, geduld, toewijding, ondersteuning, en begeleiding. In onze wekelijkse meetings kwam ik vaak aan met abstracte schetsen, halve bewijzen, en algoritmes die ook maar de rol van definitie moesten vervullen. Ondanks deze soms dubieuze onderbouwingen konden we er telkens goed over discussiëren, kwamen we altijd tot overeenstemming, en heb je me steeds toegestaan pas in een laat stadium te concretiseren. Ik ben ontzettend dankbaar voor de vrijheid die je me daarmee hebt gegeven. Ook zal ik je regelmatig gegeven advies meenemen om alle moeilijke dilemma's pas te overwegen met een paar biertjes achter de kiezen.

Kai, thank you very much for your supervision and collaboration. I want to thank you especially for welcoming me in Osaka. Staying there was a great learning experience for me, both academically and personally. I believe for most of our meetings there I robbed you of your lunchtime, but in any case they were always very fruitful and made it so that we could have a successful project.

I thank the other members of the doctorate committee, prof. Francesco Basile, prof. Martin Fabian, dr. Anne-Kathrin Schmuck, prof. Jeroen Voeten, and dr. Tim Willems for reviewing this thesis, being flexible when I had to make last-minute changes, providing constructive feedback, and participating in the defense.

I thank my colleagues from the Arrowhead Tools project, Alireza, Ferry, Gökhan, Jan Friso, Jeroen, Loek, Marc, Michel, and Ramon. Thanks to you I learned how challenging it is to share knowledge between slightly different disciplines, but also how much can be done by collaborating. Ramon, dank voor jouw deel om mij te overtuigen een PhD te gaan doen en voor al je moeite om het onderzoek dat we uitvoeren te plaatsen in de industrie. Thank you Alireza, Ferry, and Gökhan for the (not so) very serious meetings. My thanks also extend to Sharare. I cherish the nights in which we enjoyed nice food and attempted to make music.

Dennis, bedankt voor de prettige samenwerking, checks op mijn voortgang, en hulp in het gebruik van CIF/ESCET. Samen met Dennis wil ik de andere ESCET ontwikkelaars, Albert, Bert, Ferdie, en Martijn, bedanken voor alle moeite in de ontwikkeling van de toolset en het gebruiksvriendelijk implementeren van methodes die ik onderzocht, zodat die nu door iedereen in gebruik kunnen worden genomen.

My officemates, Aida, Chris, Fabio, Ferdie, Jeroen, Joshua, Karlijn, Lars, Martijn, Marzhan, Maurizio, Merlijne, Nick, Olaf, Sjoerd, and Thomas, thank you for the many trips together to the coffee machine. Thanks to you I enjoyed coming to the office a lot. There was always someone that I could discuss the problem with that I was stuck on at that moment. Most of all I recall many moments of fun distractions at the office and the good times at the conferences I participated in with some of you. My thanks also extend to the people of our neighboring office 0.07.

Thank you to all students I supervised over the years, Bart, Jord, Max, Ramon, Saikat, Sam, and Twan. Thank you for your contributions to my research. For each of you I supervised you with great joy.

Secretaresses van de vakgroep, Nancy en Roos, bedankt voor jullie inzet, meedenken, en stimulering tot interactie binnen de groep.

Mijn hartelijke dank gaat uit naar al mijn trainingsmaatjes van Squadra Veloce. Ik dank jullie voor de mooie trips, toertochten, wedstrijden, borrels, koffierondjes die geen koffierondje bleken te zijn, witte parels, overvolle borden pasta, en de, op het moment van schrijven, minstens 9289 veilige rondjes over de baan. Met jullie fietsen was voor mij altijd de ideale manier om me sportief te ontwikkelen, mijn energie kwijt te kunnen, mijn hoofd leeg te maken, maar vooral vaak om gewoon even gezellig te kunnen keuvelen.

I thank the people of Enter the Now, in particular Dan, Steyn, and Thijmen, for organizing and guiding the weekly meditation sessions that always brought me inner peace.

Ik dank mijn burens van het Veemgebouw voor de spontane babbeltjes en biertjes en voor het houden van huisdieren die ik af en toe kan aaien. Daarbij Eva extra bedankt voor het ontwerpen van de prachtige kaft.

My thanks go out to the friends I made in Osaka, Alex, Beam, Bridget, Feng, Hiro, Ineko, Lexer, Moon, Risa, Satsuki, Sony, and Swe, arigato gozaimasu for the fun trips, hikes, and delicious meals and drinks. Thank you Justin for your kind help in accommodating my stay.

Bas, bedankt voor je lessen in relativiserings- en doorzettingsvermogen die me uit onze gesprekken bijblijven.

To the ex-ASML graduate students, Arjen, Bharat, Kevin, and Sam, thank you for the sporadic nights of drinks and good company.

Veel dank aan mijn vrienden uit het Zuiden, Donald, Fabian, en Tom. Bedankt voor elke keer dat jullie mij terug op mijn plaats zetten. Ik vind het altijd ontzettend leuk om samen terug te vallen op de klassieke grappen.

Mijn eeuwige dank gaat uit aan mijn hechte groep studievrienden: Job, Joey, Maarten, Mathijs, en Thijs. Dank voor de ontelbaar leuke uitstapjes, dinertjes, feestjes, biertjes, en koffies. Ik weet niet waarom jullie je nog steeds door mij van hipstertent tot technobunker laten meeslepen, maar bedankt voor het vertrouwen. Ook bedankt voor

jullie steun, ik kan altijd terecht bij jullie met mijn problemen en voor de afleiding. Bij mijn dank betrek ik ook graag Bente, Lizet, Niki, Rachel, en Anouk, het is altijd erg gezellig wanneer jullie aanhaken. Job, ik ben ontzettend blij dat ik jou als huisgenoot had tijdens corona, dat heeft me toen enorm veel steun gegeven. Joey, bedankt voor je altijd nuchtere commentaar waar ik stiekem toch veel waarde aan hecht. Maarten, dank voor je enthousiasme en tof dat je om mijn grapjes lacht. Mathijs, bedankt voor de checks hoe het met me gaat en de support wanneer ik het nodig heb. Thijs, ik dank je voor je altijd aanwezig en aanstekelijke positiviteit.

Mijn dank van onuitdrukbaar formaat is bestemd voor mijn moeder Annelies, vader Frank, en zus Eva. Ik dank jullie voor de ogenschijnlijk onuitputbare hoeveelheid onvoorwaardelijke liefde die jullie me geven. De weekenden met zijn viertjes thuis hebben een bijzonder plekje in mijn hart en zijn heel belangrijk om mezelf te gronden. Mama, dankjewel voor je zorg, adviezen, en alles wat je voor ons organiseert. Papa, dankjewel voor je compassie, toewijding, en je lekkere maaltijden. Eva, dankjewel voor je altruïsme en creativiteit. Je bent op meer manieren een voorbeeld voor me dan je denkt. Alle goeds heb ik aan jullie te danken.

Eindhoven, augustus 2023

Sander Thuijsman

List of publications

Peer-reviewed journal contributions

Thuijsman, Sander B.; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Discrete Event Dynamic Systems* 32 (2022), Nr. 2, p. 317–358. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-021-00354-0>. – Note: A Correction to this article is in press.

Thuijsman, Sander B.; Cai, Kai; Reniers, Michel A.: Transformational supervisor localization. In: *IEEE Control Systems Letters* 7 (2023), p.1682–1687. IEEE. – URL <https://doi.org/10.1109/LCSYS.2023.3278248>

Thuijsman, Sander B.; Reniers, Michel A.: Supervisory control for dynamic feature configuration in product lines. In: *Transactions on Embedded Computing Systems* (2023). ACM Press. – URL <https://doi.org/10.1145/3579644>. – In press.

Thuijsman, Sander B.; Hendriks, Dennis; Reniers, Michel A.: Reducing the computational effort of symbolic supervisor synthesis. Submitted to: *Discrete Event Dynamic Systems*.

Peer-reviewed conference contributions

Lin, Liyong; Thuijsman, Sander B.; Zhu, Yuting; Ware, Simon; Su, Rong; Reniers, Michel A.: Synthesis of supremal successful normal actuator attackers on normal supervisors. In: *Proceedings of the 2019 American Control Conference*, IEEE, 2019, p. 5614–5619. – URL <https://doi.org/10.23919/acc.2019.8814712>

Thuijsman, Sander B.; Hendriks, Dennis; Theunissen, Rolf J. M.; Reniers, Michel A.; Schiffelers, Ramon R. H.: Computational effort of BDD-based supervisor synthesis of extended finite automata. In: *Proceedings of the IEEE 15th International Conference on Automation Science and Engineering*, IEEE, 2019, p. 486–493. – URL <https://doi.org/10.1109/coase.2019.8843327>

Thuijsman, Sander B.; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2020, p. 309–316. – URL <https://doi.org/10.1016/j.ifacol.2021.04.030>

Reniers, Michel A.; Thuijsman, Sander B.: Supervisory control for dynamic feature configuration in product lines. In: *Forum for Specification and Design Languages*, IEEE, 2020, p. 1–8. – URL <https://doi.org/10.1109/fdl150818.2020.9232937>

Lousberg, Sam A. J.; Thuijsman, Sander B.; Reniers, Michel A.: DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2020, p. 429–436. – URL <https://doi.org/10.1016/j.ifacol.2021.04.058>

Thuijsman, Sander B.; Reniers, Michel A.; Hendriks, Dennis: Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis. In: *Proceedings of the IEEE 17th International Conference on Automation Science and Engineering*, IEEE, 2021, p. 777–783. – URL <https://doi.org/10.1109/case49439.2021.9551593>

Thuijsman, Sander B.; Reniers, Michel A.; Cai, Kai: Transformational nonblocking verification. In: *Proceedings of the 16th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2022, p. 256–263. – URL <https://doi.org/10.1016/j.ifacol.2022.10.351>

Fokkink, Wan J.; Goorden, Martijn A.; Hendriks, Dennis; Beek, Dirk A. van; Hofkamp, Albert T.; Reijnen, Ferdie F. H.; Etman, Pascal F. P.; Moormann, Lars; Mortel-Fronczak, Joanna M. van de; Reniers, Michel A.; Rooda, Jacobus E.; Sanden, Bram J. van der; Schiffelers, Ramon R. H.; Thuijsman, Sander B.; Verbakel, Jeroen J.; Vogel, Han A.: Eclipse ESCET™: the Eclipse supervisory control engineering toolkit. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023, p. 44–52. – URL https://doi.org/10.1007/978-3-031-30820-8_6

Thuijsman, Sander B.; Cai, Kai.; Reniers, Michel A.: Transformational supervisor localization. In: *62nd IEEE Conference on Decision and Control*, IEEE, 2023. – In press.

Non peer-reviewed contributions

Lin, Liyong; Thuijsman, Sander B.; Zhu, Yuting; Ware, Simon; Su, Rong; Reniers, Michel A.: Synthesis of successful actuator attackers on supervisors. In: *arXiv*, Cornell University Library, 2019. – URL <https://doi.org/10.23919/acc.2019.8814712>

Thuijsman, Sander B.; Reniers, Michel A.: Supervisory control for product lines with dynamic feature configuration. In: *Proceedings of the 39th Benelux Meeting on Systems and Control*, 2020, p. 83. – URL <https://beneluxmeeting.nl/2020/uploads/papers/boa.pdf>

Thuijsman, Sander B.; Reniers, Michel A.: Conversion of LSAT behavioral specifications to automata. In: *arXiv*, Cornell University Library, 2020. – URL <https://arxiv.org/abs/2011.03249>

Thuijsman, Sander B.; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *The 19th Belgium-Netherlands Software Evolution Workshop*, 2020. – URL <https://benevol2020.github.io>

Thuijsman, Sander B.; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Proceedings of the 40th Benelux Meeting on Systems and Control*, 2021, p. 50. – URL <https://beneluxmeeting.nl/2021/uploads/bmsc/boa.pdf>

Thuijsman, Sander B.: Tool interoperability for model-based system design. In: *Proceedings of the 41st Benelux Meeting on Systems and Control*, 2022, p. 104. – URL https://beneluxmeeting.nl/2022/uploads/images/2022/boa_BeneluxMeeting2022_Web_betaV2.pdf

Thuijsman, Sander B.; Kahraman, Gökhan; Mohamadkhani, Alireza; Timmers, Ferry; Cleophas, Loek G. W. A.; Geilen, Marc C. W.; Groote, Jan Friso; Reniers, Michel A.; Schiffelers, Ramon R. H.; Voeten, Jeroen P. M.: Tool interoperability for model-based systems engineering. In: *arXiv*, Cornell University Library, 2023. – URL <https://arxiv.org/abs/2302.03503>

Contents

Abstract	v
Samenvatting	vii
Dankwoord	ix
List of publications	xiii
Contents	xvii
1 Introduction	1
1.1 Engineering of cyber-physical systems	1
1.1.1 Engineering approaches	2
1.1.2 Supervisory control theory	5
1.2 Research questions	7
1.3 Contributions	8
1.4 Thesis outline	12
2 Transformational supervisor synthesis	13
2.1 Introduction	13
2.2 Preliminaries	16
2.2.1 Supervisor synthesis algorithm	19
2.3 Model delta	23
2.4 Atomic adaptations	24
2.4.1 Added initial property	25
2.4.2 Removed initial property	26
2.4.3 Added marked property	27
2.4.4 Removed marked property	28
2.4.5 Added transition	29
2.4.6 Removed transition	30
2.4.7 Other atomic adaptations	32

2.5	Transformational Supervisor Synthesis for any model delta	32
2.5.1	Iterative Transformational Supervisor Synthesis	33
2.5.2	Grouped Transformational Supervisor Synthesis	35
2.6	Experiments	39
2.6.1	Practical notes	39
2.6.2	Transfer Line	41
2.6.3	Lithography Machine Wafer Logistics	43
2.6.4	Correction	47
2.7	Conclusions	48
3	Transformational supervisor localization	49
3.1	Introduction	49
3.2	Preliminaries	51
3.3	Supervisor localization	52
3.4	Problem definition	55
3.5	Transformational supervisor localization	56
3.5.1	Isolating conflicts	56
3.5.2	General procedure	58
3.6	Case study: Cat and Mouse Tower	60
3.7	Conclusions	63
4	Supervisory control for dynamic feature configuration in product lines	67
4.1	Introduction	67
4.1.1	Related work	69
4.1.2	Structure	70
4.2	Preliminaries	71
4.2.1	Feature Models	71
4.2.2	CIF	73
4.3	Static feature models in CIF	75
4.4	Dynamic configuration	78
4.4.1	Single feature reconfiguration	79
4.4.2	Multi feature reconfiguration	79
4.4.3	Strictness of the feature constraints	80
4.5	Modeling of uncontrolled behavior	82
4.5.1	Behavior of the uncontrolled system	82
4.5.2	Component reappearance	85
4.6	Specification of requirements	86
4.6.1	Behavioral requirements	86
4.6.2	Requirements during configuration	90
4.7	Supervisory controller synthesis	91
4.8	Case study: Body Comfort System	93
4.9	Concluding Remarks	98

5	Reducing the computational effort of symbolic supervisor synthesis	101
5.1	Introduction	101
5.2	Symbolic supervisor synthesis	105
5.2.1	Automata	105
5.2.2	Symbolic supervisor synthesis	107
5.2.3	Binary Decision Diagrams	109
5.2.4	CIF	110
5.3	Evaluating computational effort in symbolic supervisor synthesis	110
5.3.1	Peak used BDD nodes	111
5.3.2	BDD operation count	112
5.3.3	Relevance of metrics	112
5.3.4	Impact of variable- and edge order on computational effort . . .	116
5.4	DCSH variable ordering heuristic	121
5.4.1	Transition relation, variable order, and computational effort . . .	121
5.4.2	Dependency Structure Matrix reordering	122
5.4.3	Experiments	125
5.5	Edge order	128
5.6	Efficiently enforcing requirements	130
5.6.1	Current application of requirements	133
5.6.2	Efficient application of requirements	136
5.6.3	Experiments	140
5.7	Conclusion	141
6	Conclusions	143
6.1	Answers to the research questions	143
6.2	Contribution to synthesis-based engineering and future work	144
6.2.1	Transformational approaches in supervisory control	144
6.2.2	Supervisory control for product lines	145
6.2.3	Efficient symbolic supervisor synthesis	145
	References	147
	Proofs for atomic TSS algorithms	159
	Proofs for ITSS	165
	Proofs for GTSS	169
	Curriculum vitae	173

Chapter 1

Introduction

This thesis is placed in the context of cyber-physical systems engineering, which we discuss first in this introductory chapter. We explain several engineering approaches and introduce supervisory control theory, which is the theoretical framework this thesis is based on. Research questions are posed in the subsequent section. The contributions of this thesis, that tackle the research questions, are discussed next. Finally in this chapter, a short outline of the thesis is presented.

1.1 Engineering of cyber-physical systems

A cyber-physical system consists of a mechanism interacting with the world (physical component), which is monitored and controlled by a computational (cyber) component. Cyber-physical systems can be found in nearly all industrial sectors. Some examples are:

1. Medical systems, for example the Preceyes' surgical robot used for eye operations (de Smet et al. 2018), or Philips Medical System's MRI scanner (Theunissen et al. 2014);
2. Manufacturing systems, for example ASML's lithography machine used in semiconductor manufacturing (van der Sanden et al. 2015), a lithium-ion battery manufacturing system (Liu et al. 2021), or a pick-and-place robot used in PCB manufacturing as displayed in Figure 1.1;
3. Infrastructural systems, for example the Algora complex that consists of a waterway lock and movable bridge (Reijnen et al. 2020), or the Eerste Heienoord tunnel (Moormann et al. 2020);
4. Distribution systems, for example autonomous vehicles in a warehouse (D'Andrea and Wurman 2008; Basile et al. 2019);
5. Agricultural systems, for example MIT's distributed robot garden (Correll et al. 2009), or a greenhouse with automated environment control (Yoo et al. 2007);

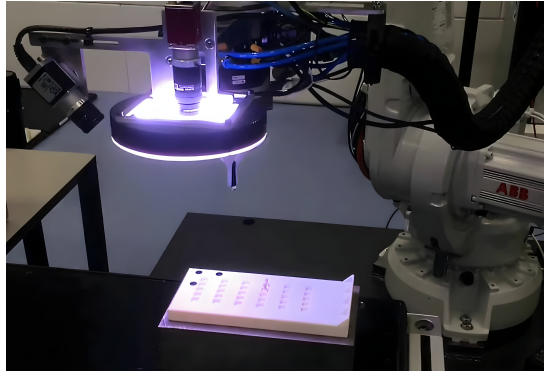


Figure 1.1: Example of a cyber-physical system: A pick-and-place robot is taking a picture of a tray to locate the slot in which it is about to place the electronic component it is holding. Picture taken by Sander Thuijsman in 2017 at Applied Micro Electronics “AME” BV, Eindhoven, The Netherlands.

6. Transportation systems, for example automated driving systems in cars (Selvaraj et al. 2022), or control systems in airplanes (Demirci 2021);
7. Energy systems, for example power distribution (Li et al. 2019), or power conversion (Guerin et al. 2012).

Many of these systems can be considered safety-critical: a failure or malfunction may lead to death, serious injury, environmental harm, or damages to property. Therefore, the correct functioning of cyber-physical systems is vital.

1.1.1 Engineering approaches

There are several layers in the control structure of a cyber-physical system, as is schematically presented in Figure 1.2. On the lowest level are the mechanical components. These mechanical components are typically steered by actuators and monitored by sensors. A low-level controller directly manages the actuation and sensing signal, and can perform tasks like trajectory tracking or signal processing. The supervisory controller (supervisor) steers the low-level controllers and receives information from them. The supervisor is responsible for the high-level control strategy, making sure all components cooperate and the system properly performs its tasks.

While not depicted in Figure 1.2, also the supervisory control layer may be distributed over a set of supervisors, and there may be hierarchy within each layer.

Each layer shown in Figure 1.2 has its own design approaches. In this work, we focus on the design of supervisory controllers. Traditionally, supervisory controller design is performed as depicted in Figure 1.3(a). Essentially, controllers are first specified in documents. These documents state functional and safety requirements,

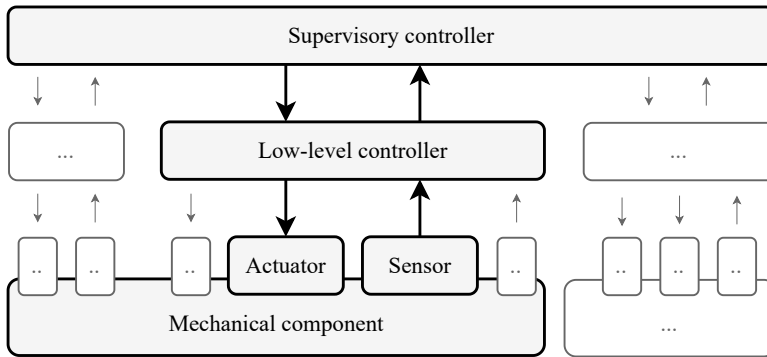


Figure 1.2: Control architecture of a cyber-physical system.
Drawing inspired from Fokkink et al. (2023).

and which signals the supervisor should actuate, depending on the sensor signals. Through programming, the controller is manually implemented on a physical system. Finally, the implementation is verified and validated by means of testing, to ensure the requirements are adhered to. When at the implementation stage something is revealed to not work, this is often a fault in the design documents, which then have to be updated.

There are several downsides to this traditional approach of engineering. Some examples are: (1) It is difficult to write down requirements in a (textual) document, often these requirements can be interpreted in multiple ways. Also, (2) frequently the set of requirements is inconsistent or incomplete. For example, not all possible hardware failures are taken into account. (3) Cyber-physical systems are designed by multi-disciplinary teams. Engineers from different domains use different technical languages. A software engineer making the controller implementation may not always understand the requirements constructed by system engineers. (4) When the implementation is updated, the design documentation may not be amended at the same time, leading to outdated documentation. Finally, (5) verification and validation can only be performed once the controller is implemented. Mistakes or improvement areas are only discovered late during the engineering process. These downsides make traditional engineering of supervisory controllers a labor-intensive and error-prone process, particularly for large or complex systems.

Essentially, many of the downsides of traditional engineering arise from the large gap between the specification and implementation. By applying model-based engineering, this gap is bridged by using models, as depicted in Figure 1.3(b). As an intermediate step between specification and implementation, a mathematical model of the controller is made. This model unambiguously specifies how the controller acts in every situation. The model can be used for verification and validation. Once a satisfactory model is obtained, it can be implemented. This can be done through manual implementation, or, since it is now known how the controller should act in every situation, code for the controller can be generated automatically. Many of the downsides mentioned

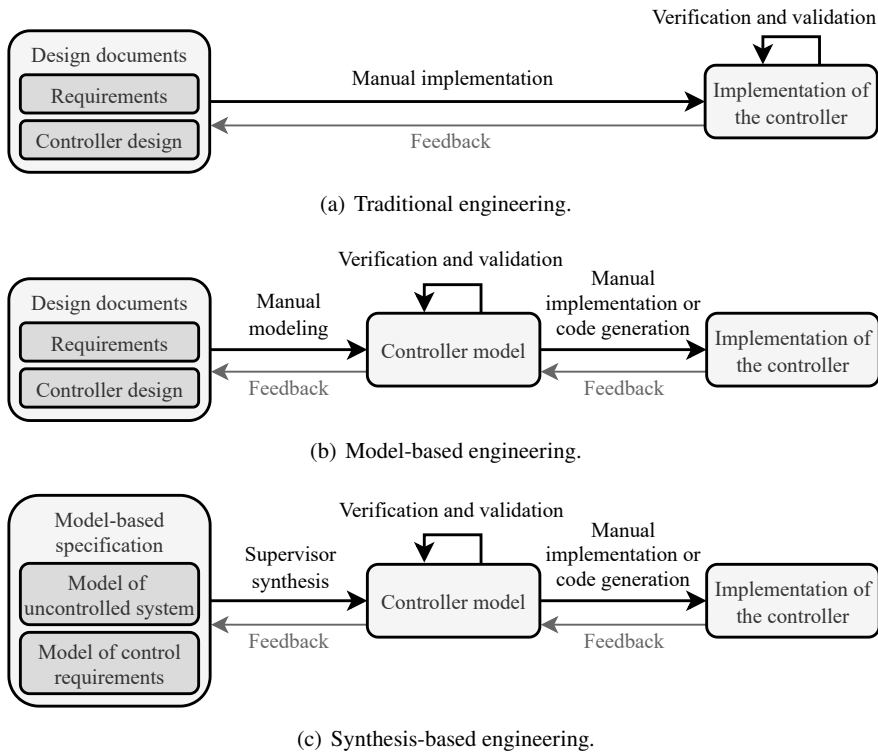


Figure 1.3: Engineering approaches for supervisory controller development.

Drawings are strongly inspired from: <https://eclipse.org/escet/v0.9/cif/synthesis-based-engineering/approaches/index.html>.

for traditional engineering are addressed using model-based engineering. Up-to-date models can act as a single source of truth, and therefore aid in the communication between engineers. These models also force engineers to specify how the system should act in every situation. Verification and validation can now be performed earlier in the engineering process using the models, making model-based engineering more efficient than traditional engineering. Nevertheless, model-based engineering still has its downsides: (1) There is little traceability between the model and documents. It can be unclear why certain parts of the model are as they are. This makes it difficult to know how to update the model when the system requirements change. Also, (2) the documentation might still be ambiguous or wrongly translated to the controller model, or (3) become out of date with respect to the model. Finally, (4) when faults are recognized in verification and validation, it may be unclear how to update the model to address the issue.

Figure 1.3(c) shows the synthesis-based engineering approach. Just like model-based engineering, a mathematical controller model is at the core of synthesis-based

engineering. However, in the synthesis-based approach the controller model is not manually constructed. Instead, a model is made of the uncontrolled system (what the system *can* do) and a model is made of requirements (what the system *should* (not) do). By applying *supervisor synthesis* to these models, a control strategy is algorithmically obtained. The supervisory controller that results from synthesis is correct-by-construction: the specified requirements are always adhered to. Essentially, the model-based specification is the single source of truth, replacing the text-based documentation of the previous engineering approaches. The controller model can easily be kept up to date since it can be derived automatically. Using mathematical models rather than textual documents removes ambiguity from the specification. By applying supervisor synthesis, the error-prone and time-consuming task of manual control design is avoided, and replaced by structured design of formal models, thereby leading to controllers for which we can give much better guarantees on their correct functioning.

After obtaining the supervisor model, and verifying and validating its functioning, further analysis can be performed to optimize the behavior. E.g., to produce products as quickly as possible. See Ghallab et al. (2016); Ware and Su (2016); van Putten et al. (2020); and Vilela and Hill (2022) for some examples. There are various methods to go from a supervisor model to a deployed supervisory controller in a cyber-physical system. We refer to Fabian and Hellgren (1998); de Queiroz and Cury (2002); Reijnen et al. (2021); and Thuijsman et al. (2023b) for some examples.

Despite many use cases in literature showcasing the benefits of synthesis-based engineering, e.g., in Forschelen et al. (2012); Theunissen et al. (2014); van der Sanden et al. (2015); Korssen et al. (2018); Basile et al. (2019); Reijnen et al. (2020); Moormann et al. (2020); and Selvaraj et al. (2022), industrial acceptance of supervisory control theory is still scarce. Cao et al. (2002) and Wonham et al. (2018) point to a lack of experience among engineers with applying the supervisory control framework, lack of software suitable for industry, and poor algorithmic scalability when considering large systems as reasons for the scarcity of industrial acceptance.

1.1.2 Supervisory control theory

Supervisory control theory, also named the Ramadge-Wonham framework as a result of its introduction in Ramadge and Wonham (1987, 1989), is the mathematical framework that the models and algorithms of synthesis-based engineering are based on. In this section, we give a high-level overview of supervisory control theory.

For the design of supervisory controllers, certain details about how actions are performed, e.g., implementation details of low-level controllers, are inessential. Therefore, for the purpose of designing the supervisory controller, the system can be abstractly viewed as a *discrete-event system*. In a discrete-event system, the possible *states* of the system are described by a countable set, and transitions between these states are instantaneous. These transitions can be associated with *events* that indicate behavior

or actions in the system that change the state. States, events, and transitions can be modeled using *automata*.

In Figure 1.4, a graphical representation of an automaton is given, that models the gripper of the pick-and-place robot of Figure 1.1. States are drawn by circles, with their name in them, and transitions are drawn by arrows, with an event label displayed next to them. Initially, the gripper is in state `Open`, shown by the dangling incoming arrow. Event `close` can occur, upon which the automaton goes to the `Closing` state. Eventually, the gripper finishes moving (`finished_moving`), and the automaton transitions to state `Closed`.

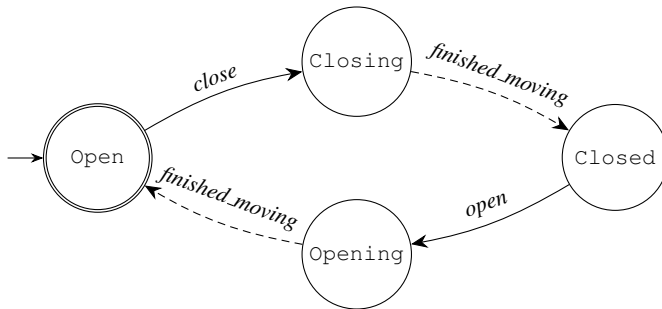


Figure 1.4: Gripper automaton for pick-and-place robot.

A point of importance in supervisory control theory is the *controllability* of events. The supervisory controller can restrict when the gripper is allowed to close, so event `close` is controllable. In the automaton drawing in Figure 1.4 we display this using a solid arrow for the transition labeled by `close`. However, after this command is performed, the supervisory controller has no influence on when the gripper actually finishes the action. Finishing of the action occurs when the force sensor notices the item has been picked, and the low-level controller stops powering the gripper. The supervisory controller is only notified about this event, but cannot restrict it from occurring, therefore `finished_moving` is an uncontrollable event. In the automaton drawing in Figure 1.4 we display this using a dashed arrow for the transitions labeled by `finished_moving`.

Another essential factor in supervisory control theory is the marking of states. *Marked* states are also often called *accepting*, *final*, or *desired* states in literature. A marked state typically indicates a situation in which the system is stable or has finished a task. For the gripper of the pick-and-place robot, state `Open` is a marked state because in this state the gripper is ready to grab another component. Graphically, this is displayed in Figure 1.4 by using a double circle for this state.

In supervisory control theory, two models are made for a system. The first is a model of the *uncontrolled system*, also called *plant*, which contains all possible behavior that the system can do, even if some of that behavior is undesirable, e.g., would result in collisions. The second model is a *requirements specification*, that defines what

behavior is desirable, and what behavior is undesirable. For example, for the pick-and-place robot we can model the gripper using the states mentioned above, and also the robot arm that can move between positions. In the uncontrolled system, the gripper and robot arm may move independently from each other. As a requirement, we can specify that the robot arm is only allowed to move away from the picking position after the gripper has finished moving.

By performing supervisor synthesis on the plant and requirement models, a supervisor model is obtained. By construction, some desirable behavioral properties are satisfied when this supervisor steers the system, which are: *safety*, *nonblockingness*, *controllability*, and *maximal permissiveness*. Safety indicates that the requirements are always adhered to. Nonblockingness indicates that the system can always go to a marked state. Controllability indicates that the supervisor only restricts controllable events from occurring, and uncontrollable events can always take place as is the case in the plant. Maximal permissiveness indicates that the supervisor restricts no more behavior than strictly necessary to satisfy safety, controllability, and maximal permissiveness, i.e., the supervisor never unnecessarily disallows an event from occurring.

1.2 Research questions

The following research questions are addressed in this thesis. The aim of answering the questions is to improve ease of use, applicability, and efficiency in supervisory control theory.

Research question 1

Lehman (1996) has defined the *laws of software evolution*. These describe what changes typically occur during a software's lifetime. The laws themselves have evolved over the years, but *the law of continuing change* has consistently been a part of them. This law states that software controlling a cyber-physical system must continually be adapted, otherwise its functioning becomes progressively less satisfactory. In other words, systems evolve over time. An example of a system that evolves over time could be an existing traffic light system to which a new bicycle detection sensor is added.

Conventionally, supervisory control theory starts from a clean sheet. Information from previous iterations in the controller's lifetime is not used. In this thesis it is investigated how to make use of the information from these previous instances, to address the following question:

How can supervisory control theory be efficiently applied to systems that evolve over time?

Research question 2

Modern-day companies often do not design just a product, they design a *product line*. This is a set of products that share a lot of commonalities, but also allow variability between them. For example, when you buy a certain type of coffee machine, there may be many optional components, like the ability to pour milk or tea. As a result, a system may have many possible unique configurations, each of which needs to be correctly controlled. Furthermore, in some circumstances the configuration may dynamically change during runtime, e.g., the component that pours milk is installed on a coffee machine that is already deployed. It is unrealistic to design supervisors one by one for all product configurations in a large product family, especially when these also need to deal with reconfiguration between these configurations. This leads to the following research question:

How can supervisory control theory be efficiently applied to a product family with dynamic reconfiguration?

Research question 3

When the size of a system grows, the computational effort required for supervisor synthesis grows exponentially. A way to mitigate this, is by symbolically representing the models using *binary decision diagrams* (BDDs), and performing supervisor synthesis on this symbolic representation. As a result, symbolic supervisor synthesis can deal with much larger systems than conventional non-symbolic synthesis. Nevertheless, minor changes in how the synthesis algorithm is performed, and the settings with which it is initiated, can have a major impact on the amount of time and memory required to perform synthesis. Since the scalability of synthesis is a major hurdle in its industrial acceptance, it is of practical benefit to seek ways to reduce its computational effort, which leads to the following research question:

How can BDDs be efficiently applied in symbolic supervisor synthesis in order to reduce the computational effort?

1.3 Contributions

This thesis has the following main contributions. These contributions address the above research questions in respective order.

Contribution 1: Transformational approaches in supervisory control

For two algorithms that are common in supervisory control theory, transformational approaches are investigated. The idea is to reuse results from a previous computation to more efficiently produce the new output, rather than performing the new computation completely from scratch.

In this setup, it is assumed that some algorithm (e.g., supervisor synthesis) has already been performed for a base system. Now, at a later point in time, the base system is modified and a variant system is created. Consequently, the result obtained for the base system (e.g., a supervisor), may not be valid anymore for the variant system. Instead of performing the same algorithm for the variant system, a transformational version of the algorithm can be used, that uses the result of the base system as a starting point. We prove that the transformational versions of the algorithms produce the same results as the conventional versions of the algorithms.

In this thesis a transformational supervisor synthesis algorithm and a transformational supervisor localization algorithm are discussed. Supervisor localization is an algorithmic approach to divide a single (large) supervisor automaton into several smaller supervisor automata, to obtain a behaviorally equivalent decomposed control structure.

The purpose of the transformational algorithms is to reduce the computation time relative to their basic version. By means of experiments on industrial systems, the computational benefit is evaluated.

In Figure 1.5 a sketch is provided that shows how the contributed approach compares to the conventional method. In the conventional method, we see that a car is modeled and on which a computation, e.g., supervisor synthesis, is applied. The car evolves over time, we see that first an (actively controlled) spoiler is added, and later also a headlight. Conventionally, the entire computation is repeated from scratch every time. By using a transformational method, the output from the previous computation is reused and transformed to more efficiently obtain the output for the adapted instances of the car.

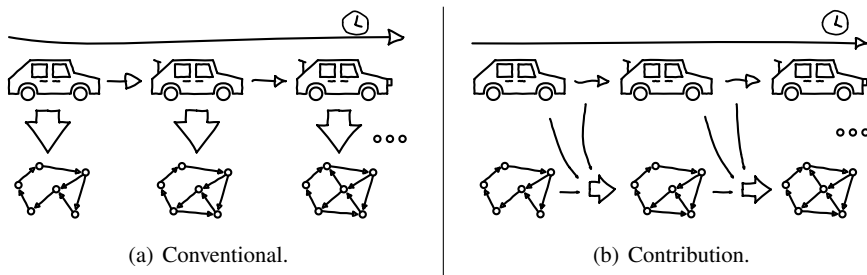


Figure 1.5: Contribution 1: Instead of starting each computation from scratch every time a system is updated, in the transformational approach previous results are used in new computations.

Contribution 2: Supervisory control for product lines

A framework for engineering supervisory controllers for product lines with dynamic feature configuration is presented. The approach consists of the following steps:

1. *Modeling a product line using a feature model.* A feature model defines which combinations of features are considered valid product configurations in a product line.
2. *Representing the feature model in extended finite automata.* This is prerequisite, because we apply supervisory controller synthesis that is based on automata specifications.
3. *Capturing dynamic configuration of features in the models.* We pay special attention to the situation where features might enter or leave the system during runtime.
4. *Modeling uncontrolled system behavior such that it properly takes the current configuration into account.* A component-wise specification of the system behavior is made, where the component behavior is linked to the presence of features in the configuration.
5. *Modeling behavioral requirements depending on the presence of features.* The requirements of the behavior are dependent on the current configuration. Additionally, different requirements may apply when the system is in a transitional phase between valid configurations.
6. *Applying supervisory controller synthesis.* A correct-by-construction supervisory controller is obtained from the developed models. It can control all configurations in the product line, as well as dynamic reconfiguration between those.

The framework is demonstrated by an industrial use case.

Figure 1.6 sketches how this contribution compares to the conventional method. There is a product line that contains many unique configurations of a car. Conventionally, the synthesis-based engineering approach needs to be performed for each instance. Using the contribution, the product line can be specified in one model, and

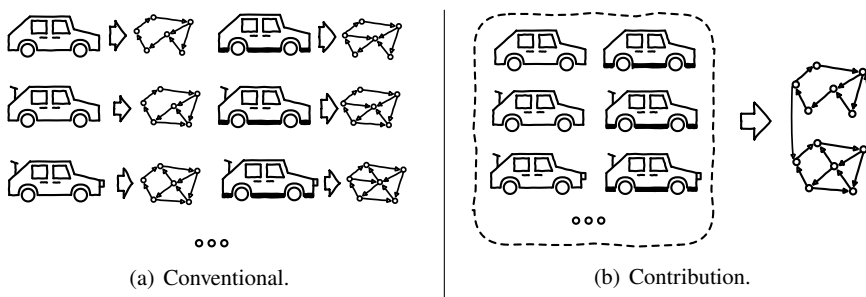


Figure 1.6: Contribution 2: Instead of designing controllers one by one, a supervisor is obtained that can control an entire product line.

synthesis is only performed once. Furthermore, dynamic configuration is possible, so the supervisory controller can deal with the situation that, e.g., a headlight is added to a car that is already deployed. Note that in this case all possible configurations are known a priori, which is a different assumption as is made in Contribution 1, where the system's instances are not known beforehand.

Contribution 3: Reducing the computational effort of symbolic supervisor synthesis

Two BDD-based metrics are introduced that express the computational effort of a symbolic supervisor synthesis in a deterministic and platform-independent way. These metrics can be used to analyze the efficiency of the synthesis algorithm. Based on this analysis, modifications can be made to the way the BDDs are handled during synthesis, improving the synthesis efficiency. In this thesis the following three methods are discussed that reduce the computational effort of symbolic supervisor synthesis:

1. The BDD size is heavily dependent on the variable order (the order in which the variables appear when traversing the BDD from top to bottom). A variable ordering heuristic algorithm is introduced that finds a variable order for which low computational effort is expected.
2. The order in which edges are analyzed impacts the computational effort of synthesis. Some edge ordering heuristics are analyzed. The edge ordering heuristic that performs best on average is recommended.
3. One may specify a requirement that reaching a particular set of states is not allowed. A method is introduced that efficiently enforces such requirements during synthesis.

All methods are supported by extensive experimentation using a variety of models from literature.

Figure 1.7 provides a sketch that presents the contribution. During synthesis, the system is symbolically represented as a BDD: the tree-like structure in the middle of Figures 1.7(a) and 1.7(b). By applying the proposed techniques, the BDD size is smaller, indicating that supervisor synthesis is now more efficient.

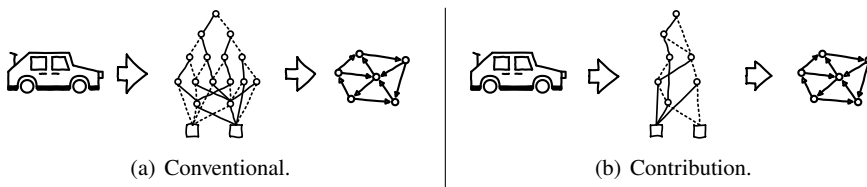


Figure 1.7: Contribution 3: Efficiency of symbolic supervisor synthesis is improved by reducing BDD sizes.

1.4 Thesis outline

The above contributions are discussed in the following chapters. Transformational supervisor synthesis and transformational supervisor localization are respectively discussed in Chapters 2 and 3. To improve readability, the (elaborate) proofs for transformational supervisor synthesis are placed in the appendix at the end of this thesis. Supervisory control for product lines is studied in Chapter 4. In Chapter 5 the reduction of computational effort of symbolic supervisor synthesis is investigated. Finally, conclusions are provided in Chapter 6.

Since each chapter uses mathematical definitions relevant for that part of the research, mathematical preliminaries are provided separately for each chapter. For instance, Chapters 2 and 3 use finite state automata as a mathematical framework, as they are easy to understand and very commonly used in supervisory control theory. In Chapters 4 and 5, *extended* finite state automata are used, which are finite state automata augmented with variables, because the use of the variables is required for the methods described in those chapters.

Each chapter is strongly based on a scientific journal paper, to which we refer in a footnote on the first page of the chapter. Minor modifications are made to fit the paper in this thesis' format, but otherwise the contents are kept consistent.

Chapter 2

Transformational supervisor synthesis

Abstract Supervisory controller synthesis is a means to compute correct-by-construction controllers for discrete-event systems. As these systems and their requirements evolve over time, an updated supervisor needs to be computed each time an adaptation takes place. We consider the case that a supervisor has been synthesized for a given model, after which this model is (slightly) adapted. We investigate if we can make use of the previous synthesis result, in order to more efficiently compute the supervisor for the adapted model. We introduce model deltas as a means to describe the difference between pairs of models. Using the model deltas, a notion of atomic adaptations is introduced. For these atomic adaptations, algorithms are provided to compute the supervisor for the adapted model in a transformational manner from the previous synthesis result, rather than performing a completely new synthesis. These atomic adaptations can be iterated over, to transformationally compute a supervisor for model deltas that contain a number of atomic adaptations. To improve efficiency, it is shown how atomic adaptations can be grouped together based on their required computations and be processed at the same time. A running example is used to support the explanations on the functioning of the algorithms. The efficiency of the method is evaluated by means of both an academic and an industrial use case. Unfortunately these experiments point out that the transformational approach is less efficient than basic supervisor synthesis, and therefore the transformational approach is not recommended.

2.1 Introduction

Supervisory control theory, as introduced by Ramadge and Wonham (1987, 1989), is a model-based approach to control discrete-event (dynamic) systems. Given a plant

This chapter is strongly based on: Thuijsman, Sander B.; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Discrete Event Dynamic Systems* 32 (2022), Nr. 2, p. 317–358. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-021-00354-0>. – Note: A Correction to this article is in press, this chapter contains the corrected results.

model (that defines all possible system behavior) and a requirement specification (which defines what plant behavior is allowed), a supervisor can be computed algorithmically (*synthesized*) that restricts the plant's behavior so that it is in accordance with the requirements. Depending on the synthesis algorithm, the supervised system has some useful properties, such as *safety*, *nonblockingness*, *controllability* and *maximal permissiveness*. The benefit of supervisory control theory has been shown in literature for varying fields of industry. Some examples where it is applied to controller design are; A patient support table of a magnetic resonance imaging scanner in Theunissen et al. (2014), chemical process control in Rawlings et al. (2014), lithography machines in van der Sanden et al. (2015), a waterway lock and movable bridge combination in Reijnen et al. (2020), construction robotics in Rosa et al. (2020), and tactical planning for automated vehicles in Krook et al. (2020). Despite the advantages of applying this technique, and the examples thereof shown in case studies, industrial acceptance is still scarce compared to other topics of control theory. Wonham et al. (2018) point to the *state space explosion* as one of the barriers to industrial acceptance. When the size of the system grows, the time and space (memory) required for synthesis grows exponentially.

We consider the situation sketched in Figure 2.1; A supervisor has been synthesized for a particular specification of plant and requirements. Later, a (slight) adaptation is made to the specification, so that we are going to need a new supervisor. In state of practice, a completely new synthesis would be performed on the adapted model. We investigate how to reuse the initial model and synthesis result, in order to more efficiently synthesize a new supervisor, while the supervisor's desired properties are retained.

The reuse of artifacts during (software) development is considered in (*software*) *Product Line Engineering (PLE)*. Pohl et al. (2005) define: 'Software product line engineering is a paradigm to develop software applications using platforms and mass customisation.' By reusing domain artifacts and exploiting product line variability, companies can employ PLE to increase product individualization, reduce development costs, reduce time-to-market, and enhance product quality. Pohl et al. (2005) point to model-based software development as an ideal candidate for employing PLE.

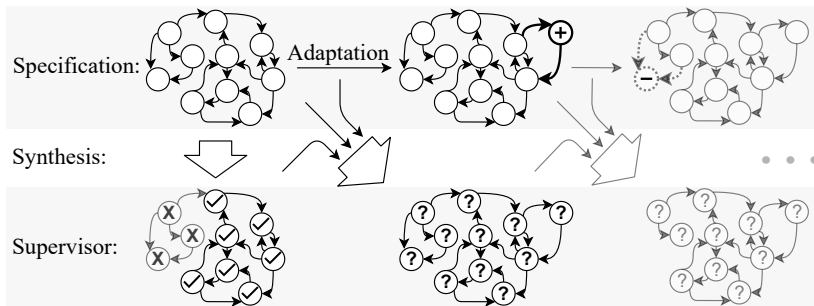


Figure 2.1: Schematic overview synthesis for evolving system.

Within the context of PLE, Schaefer et al. (2012) characterize *Delta Modeling* as a modular approach to model the variability of a system using transformations. A *model delta* explicitly specifies an adaptation that can be applied to some *base model*, in order to form a *variant model*. A particular variant model can be obtained by selecting one or more model deltas and applying them to the base model one-by-one.

Regarding adaptations that are made to software over time, Lehman (1996) has defined the *laws of software evolution*; these describe what changes typically occur during a software's lifetime. The laws themselves have evolved over the years, but *the law of continuing change* has consistently been a part of them. This law states that software controlling a cyber-physical system must continually be adapted, otherwise its functioning becomes progressively less satisfactory.

In this chapter, we elaborate on a *Transformational Supervisor Synthesis (TSS)* method. This type of synthesis uses a base model, its synthesis result and model delta to obtain a supervisor. This supervisor is the same as the would-be supervisor if a completely new synthesis was performed for the variant model, which is defined by the base model and model delta. Note that in this problem statement the model delta is unknown before performing synthesis on the base model. This is a realistic constraint, following from Lehman's law of continuing change, as well as in the case of *iterative and incremental development* (Larman and Basili 2003), where system and requirement definitions are adapted during controller development. We introduce a supervisor synthesis algorithm that outputs relevant data that can be used for TSS. We present a notion for model delta, which defines adaptations made between two models, and use this notion to identify atomic adaptations, that are the smallest possible model deltas. For different types of atomic adaptations, we provide TSS algorithms that use the result from a previous synthesis to transformationally compute a supervisor. We show how we can iterate over these atomic adaptations to transformationally obtain a supervisor when multiple atomic adaptations specify the difference between any base and variant model. To improve the efficiency, we will then present an algorithm that groups atomic adaptations together based on their required computations and processes them at the same time. These algorithms are then first applied in an academic experiment in order to analyze their effectiveness. Next, an industrial case study is presented for evolution of a controller that is used in lithography machines. Finally, conclusions are provided based on these results.

Related work:

This work is based on, and can be seen as an extension to, Thuijsman and Reniers (2020), where the TSS method was first introduced. The extension we present here includes more elaborate examples and explanations, an additional industrial case study, as well as theorems and their accompanying proofs. The algorithms we present here have been updated with respect to Thuijsman and Reniers (2020), some modifications were made on account of obtaining correct results, others for the sake of improving computational efficiency. Tjisse Claase (2020) is also closely related, in which a first attempt of applying TSS to symbolic supervisor synthesis is made, where binary

decision diagrams are used to represent the system for efficient supervisor synthesis (Fei et al. 2014).

Within the research area of discrete-event systems, PLE is mostly considered in the topic of formal verification or model checking. For example, efficient verification of linear-time temporal logic for variability-intensive systems in Classen et al. (2010) or feature-oriented modular verification of software product lines in ter Beek and de Vink (2014). Khan (2013) investigates evolving Algebraic Petri Nets, how to perform verification on the parts of the system that are affected by the property that is analyzed, and how to identify evolutions that require verification. In ter Beek et al. (2016) and Reniers and Thuijsman (2020), PLE has been applied in supervisory control. In these works a supervisory controller is synthesized for all possible product configurations given by a feature model. The output is one controller with multiple initial locations, where each initial location corresponds to a product configuration. In Reniers and Thuijsman (2020), runtime evolution of the system behavior over the configurations is studied. In contrast to this work, we do not assume a priori knowledge of the possible system configurations and the evolution takes place at design time.

2.2 Preliminaries

We consider finite state automaton A defined as a 5 -tuple: $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, where X is the finite set of states, of which $X_0 \subseteq X$ is the set of initial states and $X_m \subseteq X$ is the set of marked states. Σ is the finite set of events, also called the alphabet, which is partitioned into sets of controllable and uncontrollable events, respectively Σ_c and Σ_u . Σ^* denotes all possible finite strings using events in Σ . \longrightarrow is the finite set of transitions, a transition is a 3 -tuple: $(x_{or}, \sigma, x_{tar}) \in X \times \Sigma \times X$, specifying a transition from origin state x_{or} to target state x_{tar} over event σ . We denote the existence of a transition $(x_{or}, \sigma, x_{tar}) \in \longrightarrow$ by: $x_{or} \xrightarrow{\sigma} x_{tar}$. Likewise, the existence of a sequence of transitions over intermediate states can be addressed by: $x_{or} \xrightarrow{s} x_{tar}$, for $s \in \Sigma^*$.

The *synchronous product* of automata $A_1 = (X_1, \Sigma_1, \longrightarrow_1, X_{0,1}, X_{m,1})$ and $A_2 = (X_2, \Sigma_2, \longrightarrow_2, X_{0,2}, X_{m,2})$ is defined as: $A_1 || A_2 = (X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \longrightarrow_{12}, X_{0,1} \times X_{0,2}, X_{m,1} \times X_{m,2})$, where \longrightarrow_{12} is constructed by:

$$\begin{aligned}
 & ((x_{or,1}, x_{or,2}), \sigma, (x_{tar,1}, x_{tar,2})) \in \longrightarrow_{12}, \\
 & \quad \text{if } \sigma \in \Sigma_1 \cap \Sigma_2, x_{or,1} \xrightarrow{\sigma}_1 x_{tar,1}, x_{or,2} \xrightarrow{\sigma}_2 x_{tar,2} \\
 & ((x_{or,1}, x_2), \sigma, (x_{tar,1}, x_2)) \in \longrightarrow_{12}, \\
 & \quad \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2, x_{or,1} \xrightarrow{\sigma}_1 x_{tar,1} \\
 & ((x_1, x_{or,2}), \sigma, (x_1, x_{tar,2})) \in \longrightarrow_{12}, \\
 & \quad \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1, x_{or,2} \xrightarrow{\sigma}_2 x_{tar,2}
 \end{aligned} \tag{2.1}$$

For a given automaton A , we apply supervisor synthesis to generate a supervisor *subautomaton* S of A that is *reachable*, *coreachable*, *controllable*, and *maximally permissive*.

An automaton $S = (Y, \Sigma_S, \longrightarrow_S, Y_0, Y_m)$ is a subautomaton of $A = (X, \Sigma, \longrightarrow, X_0, X_m)$ if $Y \subseteq X$, $\Sigma_S = \Sigma$, $\longrightarrow_S \subseteq \longrightarrow$, $Y_0 \subseteq X_0$, and $Y_m \subseteq X_m$. In this work, the subautomata we encounter are restricted to $\longrightarrow_S = \longrightarrow \cap (Y \times \Sigma \times Y)$, $Y_0 = Y \cap X_0$, and $Y_m = Y \cap X_m$.

A state $x_r \in X$ is *reachable* if it can be reached from some initial state; $x_0 \xrightarrow{s} x_r$ for some $x_0 \in X_0$, $s \in \Sigma^*$. A state $x_{cr} \in X$ is *coreachable* if from it a marked state can be reached; $x_{cr} \xrightarrow{s} x_m$ for some $x_m \in X_m$, $s \in \Sigma^*$. Supervisor automaton S is called (co-)reachable for plant automaton A , if all its states can be defined as such. An automaton for which all reachable states are coreachable is commonly called nonblocking in literature. We say that for automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor automaton $S = (Y, \Sigma_S, \longrightarrow_S, Y_0, Y_m)$ is *controllable* if $\longrightarrow \cap (Y \times \Sigma_u \times X) \subseteq \longrightarrow_S$. If S is controllable, the states in Y are also called controllable. Maximally permissive says that S is the maximal subautomaton of A for which coreachability, reachability, and controllability are ensured. Meaning the supervisor does not disable any transitions that do not strictly need to be disallowed.

In addition to the properties of the supervisor mentioned above, problem formulations for supervisor synthesis often include a *safety* constraint; Along with the plant, some requirement specification on the plant's behavior is given. The supervisor should restrict the behavior of the plant so that the requirement specification is always satisfied. In such a case, a *plantified* requirement automaton can be constructed by introducing a non-coreachable *sink state*. Transitions that are not in accordance with the specification are redirected to this sink state. Given a requirement automaton $R = (X, \Sigma, \longrightarrow, X_0, X_m)$, the plantified requirement automaton is obtained as follows (Flordal et al. 2007):

$$R^\perp = (X \cup \{\perp\}, \Sigma, \longrightarrow^\perp, X_0, X_m), \quad (2.2)$$

where $\perp \notin X$ is the new sink state and

$$\longrightarrow^\perp = \longrightarrow \cup \{(x, \sigma_u, \perp) \mid x \in X, \sigma_u \in \Sigma_u, \nexists (x_{tar} \in X) x \xrightarrow{\sigma_u} x_{tar}\}.$$

A safe supervisor can be obtained by synthesizing a coreachable and controllable supervisor on the synchronous product of the plant automata and plantified requirement automata, as proven in Flordal et al. (2007). Other ways of specifying requirements can be plantified as well. For example state exclusion requirements discussed in Markovski et al. (2010) are plantified by removing excluded controllable transitions from the plant, and directing the excluded uncontrollable transitions to the sink state. Therefore, in this work we will consider synthesizing a coreachable supervisor for a single automaton, without loss in generality regarding safety constraints or networks of automata.

We allow automata to be non-deterministic. In the case of non-determinism, we allow the supervisor to be able to select from which of the initial states the system is allowed to start, and disable individual controllable transitions as a result of the

removal of unsafe states. So if in the plant a state has two outgoing transitions over the same controllable event, the supervisor subautomaton may contain this state with only one of these outgoing transitions. This is unlike some traditional supervisory control definitions, by for example Ramadge and Wonham (1989) or Cassandras and Lafortune (2021), where multiple outgoing events over the same event cannot be disabled individually. The distinction between these paradigms is further discussed in Flordal et al. (2007). This interpretation is required for our method as we may encounter non-deterministic automata as intermediate results. However, if the automata that we input to our synthesis algorithms are deterministic, as we will show the same result as in traditional synthesis is obtained, and the supervisor can be practically deployed without needing to perform the operations mentioned at the start of this paragraph.

Example 2.1 Running example

We will consider the plant automaton P of Figure 2.2 and requirement automaton R of Figure 2.3 as a running example throughout this chapter. A solid or dashed arrow respectively indicates a transition by a controllable or an uncontrollable event. The initial states are indicated by the incoming arrows, and the marked states are indicated by a double circle. R has alphabet $\Sigma = \{a, b\}$, so it does not synchronize on events c and d . Requirement automaton R has been plantified according to (2.2), resulting in the plantified requirement automaton R^\perp in Figure 2.4, which has the same alphabet as R . Constructing the synchronous product $P||R^\perp$ yields automaton A of Figure 2.5. Note that for the remainder of this chapter, when we discuss model deltas to this example, they are always to automaton A directly, not to P and R with an implied delta on A .

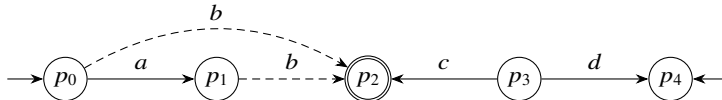


Figure 2.2: Plant automaton P .

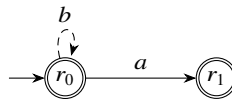


Figure 2.3: Requirement automaton R .

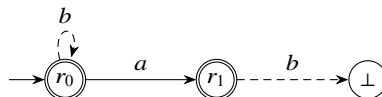
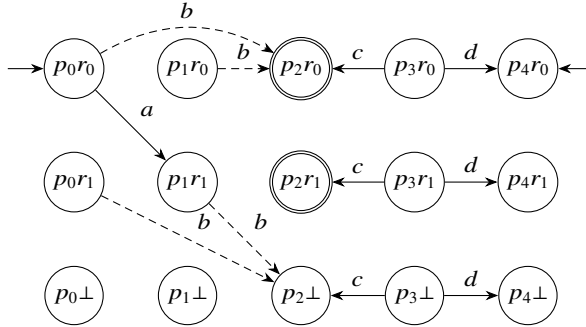


Figure 2.4: Plantified requirement automaton R^\perp .

Figure 2.5: Automaton $A = P || R^\perp$.

2.2.1 Supervisor synthesis algorithm

In Algorithm 2.1 a supervisor synthesis algorithm is presented. It is the same as the algorithm introduced in Ouedraogo et al. (2011), however, since we consider Finite State Automata rather than Extended Finite Automata, we have rewritten the algorithm assuming there are no data-variables. We also omitted the use of forbidden states in the algorithm, as we plantify the requirements. In Ouedraogo et al. (2011) predicates are constructed that define sets of states, here we represent these sets of states directly. In case the plant behavior is given by multiple plant automata, the input automaton for this algorithm can be obtained by calculating the synchronous product of the plant automata. We present this algorithm using function calls to other algorithms to facilitate reuse of these (sub-)algorithms later in this chapter.

The algorithm uses a fixpoint computation, provided in Algorithm 2.2, which iteratively calculates a set of *coreachable* states G , followed by a set of *bad* states B , that are non-coreachable or have a sequence of uncontrollable transitions to a non-coreachable state. The calculation to obtain G and B is done by means of a *Backward Reachability Search (BRS)*, given in Algorithm 2.3, for which Lemma 2.1 holds. This is a Breadth First Search algorithm taken from Kleinberg and Tardos (2005) that has a worst-case time complexity of $\mathcal{O}(|X| + |\rightarrow|)$. All found states are added to X_ω . The state space is searched in layers. For each state in the current layer, all undiscovered states that have a transition to this state are added to the next layer. After all states in the current layer have been evaluated, the algorithm moves to evaluating the states in the next layer. These steps are repeated until no more new states are found. The algorithm is slightly adapted from Kleinberg and Tardos (2005) to allow a set of starting states, instead of a singular starting state. Also, at the start of the algorithm the transitions are pruned, so that only transitions between states in the input state set X are considered. Transitions from states in the starting set X_α are also removed, as these states are already discovered as the starting set, so analyzing these transitions is not necessary. The functioning of this algorithm is well known so Lemma 2.1 is not proven here.

Algorithm 2.1 Supervisor Synthesis (SS)

Input: Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$ **Output:** Supervisor $S = (Y, \Sigma, \longrightarrow_S, Y_0, Y_m)$, good states G

- 1: $(Y, G) = \text{computeFixpoint}(A)$
 - 2: $S = (Y, \Sigma, \longrightarrow \cap (Y \times \Sigma \times Y), X_0 \cap Y, X_m \cap Y)$
 - 3: **return** (S, G)
-

Algorithm 2.2 Compute Fixpoint (`computeFixpoint`)

Input: Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$ **Output:** Supervisor states Y , good states G

- 1: $G' = X$
 - 2: **repeat**
 - 3: $G = G'$
 - 4: $G' = \text{BRS}(G, \Sigma, \longrightarrow, X_m)$
 - 5: $B = \text{BRS}(G, \Sigma_{uc}, \longrightarrow, G \setminus G')$
 - 6: $G' = G \setminus B$
 - 7: **until** $G' = G$
 - 8: $Y = \text{FRS}(G, \Sigma, \longrightarrow, X_0)$
 - 9: **return** (Y, G)
-

Lemma 2.1 For state set X , alphabet Σ , set of transitions \longrightarrow , and starting state set X_α ; $\text{BRS}(X, \Sigma, \longrightarrow, X_\alpha)$ contains all states in X from which a state in $X_\alpha \cap X$ can be reached, using transitions in \longrightarrow , over states in X , that have an event in Σ .

The bad states B are removed from G . The removal of these states can induce other states to become non-coreachable. Therefore, the algorithm repeats these steps until no further states get removed. At this point, the set of remaining states is defined as *good states* G , which is the maximal set of controllable and coreachable states, see Lemma 2.2. Proof for Lemma 2.2 is provided in Ouedraogo et al. (2011).

Lemma 2.2 For automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $(Y, G) = \text{computeFixpoint}(A)$; G is the maximal controllable and coreachable set of states in X .

Then, in order to generate a reachable supervisor, a *Forward Reachability Search* (FRS) (Algorithm 2.4, Lemma 2.3) is carried out from the set of initial states, resulting in states Y . Essentially BRS is performed with all transitions reversed to search forward instead of backward. Same as for BRS, the accompanying lemma is not proven here. Y is the maximal controllable, coreachable, and reachable subset of X , see Lemma 2.4.

Lemma 2.3 For state set X , alphabet Σ , set of transitions \longrightarrow , and starting state set X_α ; $\text{FRS}(X, \Sigma, \longrightarrow, X_\alpha)$ contains all states in X that can be reached from a state in $X_\alpha \cap X$, using transitions in \longrightarrow , over states in X , that have an event in Σ .

Lemma 2.4 For automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $(Y, G) = \text{computeFixpoint}(A)$; Y is the maximal controllable, coreachable, and reachable set of states in X .

Algorithm 2.3 Backward Reachability Search (BRS)**Input:** State set X , alphabet Σ , finite set of transitions \longrightarrow , starting set X_α **Output:** State set X_ω in X from which a sequence of transitions \longrightarrow exists through states in X , using events in Σ , to a state in $X_\alpha \cap X$

```

1:  $\longrightarrow_p = \{(x_{or}, \sigma, x_{tar}) \in \longrightarrow \mid (x_{or} \in X \wedge x_{tar} \in X) \wedge x_{or} \notin X_\alpha \wedge \sigma \in \Sigma\}$ 
2:  $X_\omega = X_\alpha \cap X$ 
3: currentlayer =  $X_\alpha \cap X$ 
4: while currentlayer  $\neq \emptyset$ 
5:   nextlayer =  $\emptyset$ 
6:   for all  $x \in$  currentlayer do
7:     for all  $\{(x_{or}, \sigma, x_{tar}) \in \longrightarrow_p \mid x_{tar} = x\}$  do
8:       if  $x_{or} \notin X_\omega$ 
9:          $X_\omega = X_\omega \cup \{x_{or}\}$ 
10:        nextlayer = nextlayer  $\cup \{x_{or}\}$ 
11:       end if
12:     end for
13:   end for
14:   currentlayer = nextlayer
15: end while
16: return  $X_\omega$ 

```

Algorithm 2.4 Forward Reachability Search (FRS)**Input:** State set X , alphabet Σ , finite set of transitions \longrightarrow , starting set X_α **Output:** State set X_ω in X to which a sequence of transitions \longrightarrow exists through states in X , using events in Σ , from a state in $X_\alpha \cap X$

```

1:  $\longrightarrow^{-1} = \{(x_{tar}, \sigma, x_{or}) \mid (x_{or}, \sigma, x_{tar}) \in \longrightarrow\}$ 
2:  $X_\omega = \text{BRS}(X, \Sigma, \longrightarrow^{-1}, X_\alpha)$ 
3: return  $X_\omega$ 

```

Proof Y is the maximal reachable set in G , following from Lemma 2.3. It is shown by Lemma 2.2 that G is the maximal controllable and coreachable subset of X . Thus, the maximal reachable subset Y in G is the maximal controllable, coreachable, and reachable subset of X . \square

Together, Y , alphabet Σ , the transitions of A between the states in Y , the initial states in Y , and the marked states in Y define the supervisor automaton S . Supervisor S is the maximal subautomaton of A that is reachable, coreachable, controllable; see Theorem 2.1. The algorithm always computes a supervisor automaton. If there are no reachable, coreachable, and controllable states then the supervisor automaton will contain no states, and hence no transitions.

Theorem 2.1 For automaton A , and $(S, G) = \text{SS}(A)$; S is maximally permissive, controllable, coreachable, and reachable with respect to A .

Proof By construction, S is the maximal subautomaton of A over states in Y . Y is the maximal controllable, coreachable and reachable set of states in A (Lemma 2.4). It follows that S is maximally permissive, controllable, coreachable, and reachable with respect to A . \square

Next to supervisor S , the synthesis algorithm outputs good state set G , in order to facilitate reuse of this set in other computations. Note that this state set is computed anyways during synthesis, it is not computed specifically for the facilitation of reuse.

Example 2.2

When applying the supervisor synthesis algorithm to automaton A of Figure 2.5, first the supervisor states and good states are calculated by `computeFixpoint` (Algorithm 2.2). The supervisor states are $\{p_0r_0, p_2r_0\}$, the good states are $\{p_0r_0, p_1r_0, p_2r_0, p_3r_0, p_2r_1, p_3r_1\}$. Next, the supervisor automaton is constructed, which provides the supervisor automaton given in Figure 2.6.

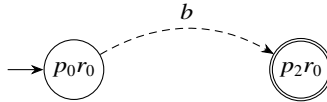


Figure 2.6: Supervisor S , for $(S, G) = \text{SS}(A)$.

For convenience we also provide a visualization of automaton A , where the states are color coded depending on their containment in the state sets resulting from synthesis, in Figure 2.7. Supervisor states (Y) (that are also good states by definition) are displayed white, good states that are not supervisor states ($G \setminus Y$) are displayed grey, and non-good states ($X \setminus G$) are displayed black.

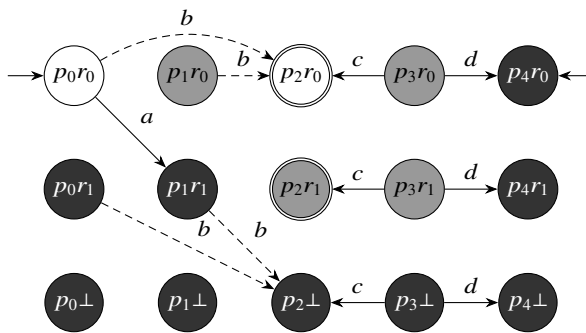


Figure 2.7: Automaton A , color coded by synthesis result.

2.3 Model delta

For the purpose of TSS we wish to model the difference between the base and variant model. We can represent any adaptation from base to variant automaton as *model delta* as 10-tuple: $\Delta = (X^+, X^-, \Sigma^+, \Sigma^-, \longrightarrow^+, \longrightarrow^-, X_0^+, X_0^-, X_m^+, X_m^-)$, which for each of the sets in the 5-tuple definition of automaton A , defines the added (+) and removed (-) elements of that set. Σ^+ and Σ^- are both partitioned into sets of controllable and uncontrollable events that are added or removed. The following constraints apply to the model delta:

1. Any removed element within the model delta, must exist in the base model:
 $X^- \subseteq X, \Sigma^- \subseteq \Sigma, \longrightarrow^- \subseteq \longrightarrow, X_0^- \subseteq X_0, X_m^- \subseteq X_m$.
2. Any added element within the model delta, must not yet exist in the base model:
 $X^+ \cap X = \emptyset, \Sigma^+ \cap \Sigma = \emptyset, \longrightarrow^+ \cap \longrightarrow = \emptyset, X_0^+ \cap X_0 = \emptyset, X_m^+ \cap X_m = \emptyset$.
3. Elements cannot simultaneously be added and removed: $X^- \cap X^+ = \emptyset, \Sigma^- \cap \Sigma^+ = \emptyset, \longrightarrow^- \cap \longrightarrow^+ = \emptyset, X_0^- \cap X_0^+ = \emptyset, X_m^- \cap X_m^+ = \emptyset$.
4. The initial and marked states of the variant model must exist in the variant state set: $X_0^+ \subseteq (X \cup X^+) \setminus X^-, X_m^+ \subseteq (X \cup X^+) \setminus X^-$.
5. Transitions must go to-and-from states, by defined events: $\longrightarrow' \subseteq X' \times \Sigma' \times X'$, where $X' = (X \cup X^+) \setminus X^-, \Sigma' = (\Sigma \cup \Sigma^+) \setminus \Sigma^-,$ and $\longrightarrow' = (\longrightarrow \cup \longrightarrow^+) \setminus \longrightarrow^-$.

When these constraints are met, we call the model delta *valid*. Note that the third condition directly follows from the first two conditions.

Note that, e.g., $x^\delta \in X_0^+$ only indicates that x^δ is a new initial state, but not (necessarily) a newly added state. If one wants to add a new state to the system, that is also an initial state, the model delta should contain both $x^\delta \in X^+$ and $x^\delta \in X_0^+$.

Given base automaton A and model delta Δ , variant automaton A' is constructed by: $A' = (X', \Sigma', \longrightarrow', X_0', X_m')$, where: $X' = (X \cup X^+) \setminus X^-, \Sigma' = (\Sigma \cup \Sigma^+) \setminus \Sigma^-, \longrightarrow' = (\longrightarrow \cup \longrightarrow^+) \setminus \longrightarrow^-, X_0' = (X_0 \cup X_0^+) \setminus X_0^-, X_m' = (X_m \cup X_m^+) \setminus X_m^-$. For a valid model delta, and well-defined base automaton, the constructed variant automaton is well-defined.

Furthermore, for any pair of well-defined automata ($A = (X, \Sigma, \longrightarrow, X_0, X_m), A' = (X', \Sigma', \longrightarrow', X_0', X_m')$), a valid model delta is constructed as follows: $X^+ = X' \setminus X, X^- = X \setminus X', \Sigma^+ = \Sigma' \setminus \Sigma, \Sigma^- = \Sigma \setminus \Sigma', \longrightarrow^+ = \longrightarrow' \setminus \longrightarrow, \longrightarrow^- = \longrightarrow \setminus \longrightarrow', X_0^+ = X_0' \setminus X_0, X_0^- = X_0 \setminus X_0', X_m^+ = X_m' \setminus X_m, X_m^- = X_m \setminus X_m'$.

The change of controllability of an event can be modeled by removing all transitions that are labeled by this event, and adding these transitions back with an added event with modified controllability.

We may address a model delta with only its non-empty part. So if we mention a model delta with $X_0^+ = \{x^\delta\}$, and no other information, this implies that the other elements in the model delta tuple are empty. In the remainder of this chapter, when considering a model delta Δ it is implied that this is a model delta from base automaton A to variant automaton A' .

2.4 Atomic adaptations

In this section we consider *atomic adaptations*, where the difference between the base and variant model can be described by a single, indivisible change in the automaton specification. Formally we can say that a model delta Δ is an *atomic adaptation* when only one of the tuple-elements is a set of size one, and all other elements are empty; $|X^+| + |X^-| + |\Sigma^+| + |\Sigma^-| + |\rightarrow^+| + |\rightarrow^-| + |X_0^+| + |X_0^-| + |X_m^+| + |X_m^-| = 1$.

We consider several types of atomic adaptations, e.g., removing a transition, or adding the marked property to a state, for which we provide an atomic TSS algorithm. The purpose of these algorithms is to calculate the supervisor states and good states of the variant automaton, using the base automaton, its synthesis result, and model delta. Theorem 2.2 holds for the algorithms. Essentially, the properties of the supervisor ((co-)reachability, controllability, and maximal permissiveness) are retained during atomic TSS. For sake of cohesion, proofs of Theorem 2.2 for each algorithm are given separately in Appendix A.

Theorem 2.2 Given base automaton A , fixpoint result $(Y, G) = \text{computeFixpoint}(A)$, and atomic adaptation Δ for which the atomic TSS algorithm is given, the atomic TSS algorithm provides a supervisor state set Y' and good state set G' such that they are equal to the fixpoint result of the variant automaton; $(Y', G') = \text{computeFixpoint}(A')$.

Figure 2.8 shows an overview of the atomic TSS method. It is similar to Figure 2.1, only now the names of the artifacts and algorithms that have been introduced are shown. The overview shows that the fixpoint for the variant supervisor can be computed in two ways, either by (1) performing `computeFixpoint` on the variant automaton directly, or by (2) performing atomic TSS using the base automaton, base supervisor fixpoint, and atomic model adaptation. Either way, the fixpoint result is the same.

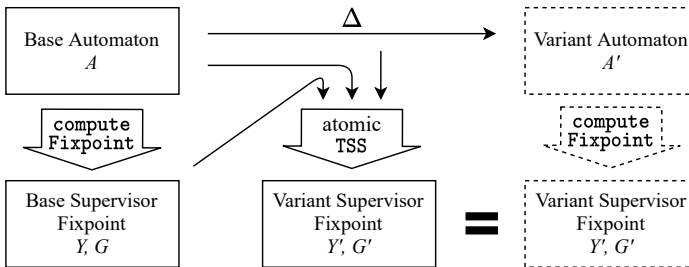


Figure 2.8: Inputs and outputs for atomic TSS for atomic adaptation Δ .

In the following subsections we consider adding or removing the initial property to some state, adding or removing the marked property to some state, and adding or removing a transition respectively. For the cases of removed or added states and events

no algorithms are provided. These atomic adaptations are discussed in Section 2.4.7. After presenting the atomic TSS algorithms in this section, we will show how we can iterate over them in Section 2.5, where we are also going to group atomic adaptations together to process them at once.

The algorithms are strongly based on Thuijsman and Reniers (2020). In some places minor modifications are made for sake of correctness and efficiency. Some of these modifications are discussed in Tijssse Claase (2020). Note that these modifications influence the experimental results presented in Thuijsman and Reniers (2020), however only to a small enough extent that they do not influence the conclusions made on those results. All algorithms are also modified to compute the supervisor state set Y instead of the supervisor S , leading to shorter notations. S can simply be computed from Y , as in line 2 of Algorithm 2.1.

2.4.1 Added initial property

We assume the situation that $(Y, G) = \text{computeFixpoint}(A)$ has been calculated for base automaton A . Some state of base automaton A has been made an initial state, which is the only adaptation to create variant automaton A' . In Algorithm 2.5 the atomic TSS algorithm is provided to compute supervisor states Y' and good states G' for the variant model, given A, Y, G , and the state with added initial property x^δ . The algorithm uses a switch statement, where the value of a variable, in this case x^δ , is tested for multiple cases. Once a case match is found, the statements associated with the particular case are executed. In case no match is found, the default statements are executed. For all atomic TSS algorithms the switch cases are mutually exclusive, which means that for the given atomic adaptation only one switch case holds, or none and then the default statement is executed.

Algorithm 2.5 Atomic Transformational Supervisor Synthesis for Added Initial Property (TSSAIP)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , state with added initial property $x^\delta \in X_0^+$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $x^\delta$ 
2:   case  $x^\delta \in G \setminus Y$  do
3:      $Y' = \text{FRS}(G, \Sigma, \longrightarrow, Y \cup \{x^\delta\})$ 
4:      $G' = G$ 
5:   default do
6:      $Y' = Y, G' = G$ 
7:   end switch
8:   return  $(Y', G')$ 

```

In Algorithm 2.5 it can be seen that two cases are considered, the first being that x^δ is in $G \setminus Y$. A state in $G \setminus Y$ is coreachable and controllable in the base model. It was not reachable in the base model, as in that case it would be part of Y . Due to addition of the initial property, we now know that x^δ is reachable, so it should become part of Y' . It is possible that more states in $G \setminus Y$ have become reachable due to x^δ being reachable, so an FRS is carried out over states G . We already know that states in Y were reachable, and they will remain reachable in the variant model. As we do not want to reinvest computational effort in finding these states in Y again, the FRS is already initiated with states Y in the starting set along with x^δ , essentially we already start closer to the fixpoint that we wish to find. States in $X \setminus G$ are not considered, as they remain non-coreachable or non-controllable in the variant model. In our running example, we can consider the adaptation to make p_3r_1 initial, which would fit under this particular case. As p_2r_1 is a reachable good state from p_3r_1 , both p_3r_1 and p_2r_1 will be added to the supervisor states Y to construct the supervisor states for the variant model Y' .

Alternatively, x^δ may be in $X \setminus G$. As we just noted, the adaptation of initial states does not influence the set of coreachable and controllable states. So in this case, we already performed the FRS over the same set G to compute Y . As G did not change, the supervisor and good states remain the same for the variant model. In our example we can consider making p_4r_1 an initial state as such an adaptation. In that variant model, p_4r_1 will remain a non-good state in $X \setminus G'$.

Finally, x^δ may be in Y . If this is the case, just like in the previous example the default statement will be executed. x^δ was already found in the FRS of the base model, so also all reachable states from x^δ in G are included in Y . Thus, the supervisor states and good states remain the same for the variant model. In the running example, making p_2r_0 an initial state would be of this case.

2.4.2 Removed initial property

We consider a similar situation as the previous section, only this time the initial property has been removed from a state instead of added. The atomic TSS algorithm for this case is shown in Algorithm 2.6.

In the first case of the algorithm, an initial state in Y is removed. This might lead to some states in Y being unreachable in the variant model. However the good states G are not influenced by the initial states, so they remain the same. Also unreachable states will remain unreachable. So an FRS is carried out over the previously reachable states Y , from the new set of initial states for the variant model to compute Y' . In the running example, the removal of initial property from p_0r_0 would fit in this case. Consequently, there are no initial states left in Y , so for the variant model there are no supervisor states; $Y' = \emptyset$. The good states remain the same.

The other case is that the removed initial state is not in Y , considered under the default statement of Algorithm 2.6. Actually we know that in this case the removed initial state is in $X \setminus G$, as states in $G \setminus Y$ could not be an initial state, they would already

have been in Y as a reachable state in G . As x^δ is not in the maximal controllable and coreachable set in this case, it does not matter if it is reachable, or initial. It will not be part of the good states and supervisor states. In the running example this could be demonstrated by the removal of the initial property from state p_4r_0 .

Algorithm 2.6 Atomic Transformational Supervisor Synthesis for Removed Initial Property (TSSRIP)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , state with removed initial property $x^\delta \in X_0^-$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $x^\delta$ 
2:   case  $x^\delta \in Y$  do
3:      $Y' = \text{FRS}(Y, \Sigma, \longrightarrow, X_0 \setminus \{x^\delta\})$ 
4:      $G' = G$ 
5:   default do
6:      $Y' = Y, G' = G$ 
7:   end switch
8:   return  $(Y', G')$ 

```

2.4.3 Added marked property

In Algorithm 2.7 the atomic TSS algorithm is provided for the case that a state was given the marked property in the model delta from A to A' .

In the first case the circumstance is considered that a non-good state is now marked. This means that some non-good states may become good and/or supervisor states because they are coreachable in the variant model. All states that were supervisor states and good states will remain so. So a fixpoint computation is instantiated where the supervisor states are already added as marked states and initial states, so this part of the states does not have to be found again in the reachability searches. In the running example we could consider the case that p_1r_1 was made a marked state. During the first iteration of the fixpoint computation it will be found as a good state, as it is marked. It will however be removed from the good states since it has an uncontrollable transition to a bad state. So for this specific example the supervisor and good states will remain the same from base to variant model.

If the state with added marked property is a good state, it was already coreachable, as well as all good states that can reach this state. So giving it a marked property is not going to change the coreachability. Thus the supervisor states and the good states remain the same. In the running example, this would be the case if for instance p_1r_0 was added to the set of marked states.

Algorithm 2.7 Atomic Transformational Supervisor Synthesis for Added Marked Property (TSSAMP)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , state with added marked property $x^\delta \in X_m^+$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $x^\delta$ 
2:   case  $x^\delta \in X \setminus G$  do
3:      $(Y', G') = \text{computeFixpoint}((X, \Sigma, \longrightarrow, Y \cup X_0, Y \cup X_m \cup \{x^\delta\}))$ 
4:   default do
5:      $Y' = Y, G' = G$ 
6:   end switch
7:   return  $(Y', G')$ 

```

2.4.4 Removed marked property

Now the case is considered that a marked state in the base model, is not a marked state in the variant model. The atomic TSS algorithm for this case is given in Algorithm 2.8.

In the first case the removal of a marked state in $G \setminus Y$ is considered. The supervisor states Y will remain the same, as x^δ is not reachable from such a state, the removal of its marked property does not influence their coreachability. Some good states might not be good states for the variant model, as they may not be coreachable anymore. Therefore a fixpoint computation is performed, with all supervisor states already added as initial states and marked states, so this part of the state space does not have to be searched anymore. In the running example we can consider removing the marked property from p_{2r1} . Consequently, the supervisor for the variant model will remain the same, but

Algorithm 2.8 Atomic Transformational Supervisor Synthesis for Removed Marked Property (TSSRMP)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , state with removed property $x^\delta \in X_m^-$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $x^\delta$ 
2:   case  $x^\delta \in G \setminus Y$ 
3:      $(Y', G') = \text{computeFixpoint}((G, \Sigma, \longrightarrow, Y \cup X_0, (Y \cup X_m) \setminus \{x^\delta\}))$ 
4:   case  $x^\delta \in Y$ 
5:      $(Y', G') = \text{computeFixpoint}((G, \Sigma, \longrightarrow, X_0, X_m \setminus \{x^\delta\}))$ 
6:   default do
7:      $Y' = Y, G' = G$ 
8:   end switch
9:   return  $(Y', G')$ 

```

p_2r_1 and p_3r_1 are not good states for the variant model as they are not coreachable anymore.

The second case considers the situation that the removed marked state is in Y . A new fixpoint computation is performed for the variant model, only the non-good states are not taken into account, as the removal of a marked state will not increase the maximal set of coreachable and controllable states. In the example we can consider removing the marked property of state p_2r_0 . As a result, there will be no supervisor states in the variant model, $Y' = \emptyset$, and only p_2r_1 and p_3r_1 remain in G' .

The final case will occur when the marked property is removed from a non-good state. This does not influence the coreachability and controllability of the good states, as they must be coreachable for marked states in G . Also the reachable part of the good states remains the same. So the variant model has the same supervisor states and good states as the base model.

2.4.5 Added transition

Now we consider the case that only a single transition has been added to the base automaton A to create variant automaton A' . So all elements in Δ are empty except \longrightarrow^+ which only contains the transition $(x_{or}, \sigma, x_{tar})$. Algorithm 2.9 computes the supervisor and good states for the variant model according to the type of adaptation that is made.

The first case considers an added transition from a state in Y to a state in $G \setminus Y$. The target and origin state were already coreachable and controllable, so this is not influenced by the addition of the transition. However the target state is now reachable, which it was not before. To find all good states that are now reachable, an FRS is performed to compute the variant supervisor states. The good states remain the same. In the running example we can consider the addition of a transition from p_0r_0 to p_3r_1 . In that case, p_3r_1 and p_2r_1 would become supervisor states in the variant model in the addition to the states already in Y . The good states remain unchanged.

Next, the addition of an uncontrollable transition from a good state to a non-good state is considered. The target state was non-coreachable or non-controllable, so transitions to this state need to be disabled. Because an uncontrollable transition is added, it cannot be disabled by the supervisor. This means that the origin state is not a good state anymore, and we wish to remove it for the variant model. The state is made non-coreachable by removing all outgoing transitions from it, and removing it from the set of marked states in case it was a marked state. The fixpoint computation is then performed on the state space spanned by the good states, with the origin state of the added transition as non-coreachable. In the running example this could be an added uncontrollable transition from p_2r_1 to $p_2\perp$. For that adaptation p_2r_1 and p_3r_1 would be removed from the good states to construct the variant good states, and the supervisor states remain the same.

In case the origin state is not a good state, adding the transition that is not a self-loop ($x_{or} \neq x_{tar}$) might influence the coreachability of this and other non-good states.

Algorithm 2.9 Atomic Transformational Supervisor Synthesis for Added Transition (TSSAT)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , added transition $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^+$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $(x_{or}, \sigma, x_{tar})$ 
2:   case  $x_{or} \in Y \wedge x_{tar} \in G \setminus Y$  do
3:      $Y' = \text{FRS}(G, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y)$ 
4:      $G' = G$ 
5:   case  $x_{or} \in G \wedge x_{tar} \in X \setminus G \wedge \sigma \in \Sigma_u$  do
6:      $(Y', G') = \text{computeFixpoint}((G, \Sigma, \longrightarrow \cap ((G \setminus \{x_{or}\}) \times \Sigma \times G), X_0 \setminus \{x_{or}\}, X_m \setminus \{x_{or}\}))$ 
7:   case  $x_{or} \in X \setminus G \wedge x_{or} \neq x_{tar}$ 
8:      $(Y', G') = \text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, Y \cup X_m))$ 
9:   default do
10:     $Y' = Y, G' = G$ 
11:  end switch
12:  return  $(Y', G')$ 

```

States that were supervisor states in the base model will remain so. Thus, a fixpoint computation is performed over the entire state set, where the supervisor states are added to the marked and initial states so that this part of the state space does not need to be searched to reduce computational effort. For the running example we can consider the case that a transition is added from p_4r_1 to p_2r_0 , in that case the supervisor states will remain the same, and p_4r_1 is added to the good states to construct the variant good states.

In all other cases, the supervisor states and good states remain the same. For example the addition of a transition from-and-to a supervisor state, all states that are good states will remain coreachable and controllable, and states in $G \setminus Y$ will remain non-reachable. In the example this could be a transition from p_2r_0 to p_0r_0 . Another example is adding a self-loop to a non-good state not influencing its non-coreachability or non-controllability. In the running example this may be a transition from p_4r_1 to p_4r_1 .

2.4.6 Removed transition

Here we consider the case that only a single transition has been removed from base automaton A to create variant automaton A' . Algorithm 2.10 computes the supervisor states and good states for the variant model according to the state sets the origin and target state of this transition belong to, and the controllability of the transition.

Algorithm 2.10 Atomic Transformational Supervisor Synthesis for Removed Transition (TSSRT)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , removed transition $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^-$

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $(x_{or}, \sigma, x_{tar})$ 
2:   case  $x_{or} \in Y \wedge x_{tar} \in Y \wedge x_{or} \neq x_{tar}$  do
3:      $(Y', G') = \text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$ 
4:   case  $x_{or} \in G \setminus Y \wedge x_{tar} \in G \wedge x_{or} \neq x_{tar}$  do
5:      $(Y', G') = \text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, Y \cup X_m))$ 
6:   case  $x_{or} \in X \setminus G \wedge x_{tar} \in X \setminus G \wedge \sigma \in \Sigma_u \wedge x_{or} \neq x_{tar}$  do
7:      $(Y', G') = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, G \cup X_m))$ 
8:   default do
9:      $Y' = Y, G' = G$ 
10: end switch
11: return  $(Y', G')$ 

```

Let us consider a removed transition, that is not a self-loop, for which x_{or} and x_{tar} both were in Y . This case is considered first in Algorithm 2.10. It is possible that due to the removal of this transition, x_{or} and other states in G might not be coreachable anymore. Also x_{tar} might not be reachable anymore. However, bad states and non-reachable states will remain as such. Therefore, a fixpoint computation is performed for the variant model, only for the states in G of the base model. As this synthesis is on a reduced state-set, it will require less effort to perform than a completely new synthesis on the variant model. In the running example this could be the removal of transition (p_0r_0, b, p_2r_0) , which would lead to no supervisor states for the variant automaton. p_0r_0 would also be removed from the good state set to construct the good states G' .

Next, we consider a removed transition from a state x_{or} in $G \setminus Y$ to a state x_{tar} in G , that is not a self-loop. x_{or} , and other good states, might become non-coreachable after removal of this transition. However, as these states are not reachable from the supervisor states, these will remain the same for the variant model. To find the set of good states for the variant model, a fixpoint computation is performed over the good states, where the supervisor states are added to the marked and initial states so that this part of the state space is not searched. For the running example, removing transition (p_3r_1, c, p_2r_1) would fall under this case. As a result the supervisor states remain the same from the base to the variant model, but p_3r_1 is removed from the good states to construct the variant good states G' .

As a third case in Algorithm 2.10, an uncontrollable transition is removed with origin and target state as not good states. It is possible that the origin state and other states were not good states due to the existence of this transition. We need to perform some additional fixpoint computation in order to find these states. However, we know that all good states that have been found already, will remain good states for the variant model. The same goes for supervisor states. Therefore, `computeFixpoint`

is instantiated with the good states added as marked states, and the supervisor states added as initial states. In the running example the removal of transition $(p_{0r1}, b, p_{2\perp})$ would fall under this case. In that circumstance, the supervisor states and good states would remain the same for the variant model as the base model.

Finally, for all other cases the supervisor states and good states remain the same between variant and base model. For example, we remove transition $(x_{or}, \sigma, x_{tar})$ from base model A , for which $x_{or} \in X \setminus G$ and $x_{tar} \in Y$. We know that in A , the state x_{or} was coreachable, as the removed transition existed to a state in Y . As (coreachable state) x_{or} does not exist in the set of good states G , it must be non-controllable. We can reason that the removal of this transition is not going to make it controllable. Thus, Y and G of the base automaton remain the same for the variant automaton. This is also observed in Algorithm 2.10.

2.4.7 Other atomic adaptations

Some atomic adaptations were not discussed in the algorithms above. These atomic adaptations are: adding a state, removing a state, adding an event, and removing an event. When these model deltas occur as an atomic adaptation, they do not influence the supervisor states or good states. For example, an added state only influences the synthesis result if there are added transitions towards or from it. Or an event can only be removed, if there are no transitions that are labeled by that event. Otherwise the model delta is not an atomic adaptation, or it is not a valid model delta. Proofs for Lemma 2.5 are provided in Appendix A.

Lemma 2.5 For an atomic adaptation that is an: added state, removed state, added event, or removed event, the supervisor states Y' of the variant model are equal to the supervisor states Y of the base model, and the good states G' of the variant model are equal to the good states G of the base model.

2.5 Transformational Supervisor Synthesis for any model delta

In this section we will not restrict the model delta to atomic cases anymore; any valid model delta of any size is allowed. A method that iterates over all atomic adaptations is discussed in Section 2.5.1. A method that groups these atomic adaptations together based on their required computation and processes them at the same time is shown in Section 2.5.2.

2.5.1 Iterative Transformational Supervisor Synthesis

In Figure 2.9 a modified version of Figure 2.8 is shown, which provides a visualization of the idea on which Iterative TSS (ITSS) is based. A non-atomic model delta describes the difference between the base and the variant model. This model delta is split into atomic adaptations, on which the atomic TSS algorithms can be applied, and the variant supervisor fixpoint is computed by iterating over these atomic adaptations.

Algorithm 2.11 provides an ITSS algorithm, that iterates over the atomic adaptations in the model delta one-by-one. Each time an atomic adaptation is applied using the results of Section 2.4, and Y' and G' are computed accordingly. After an adaptation has been applied, the tuples $X'_0, X'_m, \longrightarrow'$ are updated, so that if we were to construct an intermediate automaton $A' = (X', \Sigma', \longrightarrow', X'_0, X'_m)$, this automaton is up-to-date for the adaptations applied until that point. This intermediate automaton is then used in the input for the next atomic TSS algorithm. Before iterating over the atomic TSS algorithms, the added states and events are added to the base supervisor and automaton. After the iterations, the set of removed events is removed from the supervisor, as at this point no transitions with this event are left in the supervisor, following from the restrictions specified in Section 2.3. Because A' will not have transitions to-or-from the removed states, and the removed states are not initial or marked, the set of removed states will not be a part of Y' (or G') at this point of the algorithm. So they do not need to be removed from the supervisor. Algorithm 2.11 also outputs the variant automaton A' of which it produces the synthesis result.

Theorem 2.3 holds for Algorithm 2.11. It is similar to Theorem 2.2 for the atomic model adaptations, only modified to apply for a supervisor automaton S , rather than supervisor states Y . It also considers that the algorithm provides the correct variant automaton as output. The proof for Theorem 2.3 on Algorithm 2.11 can be found in Appendix B.

Theorem 2.3 Given base automaton A , model delta Δ , synthesis result $(Y, G) = \text{computeFixpoint}(A)$, and $(\hat{S}, \hat{G}, \hat{A}) = \text{ITSS}(A, Y, G, \Delta)$; then $\hat{S} = S'$, $\hat{G} = G'$, and $\hat{A} = A'$, for $(S', G') = \text{SS}(A')$.

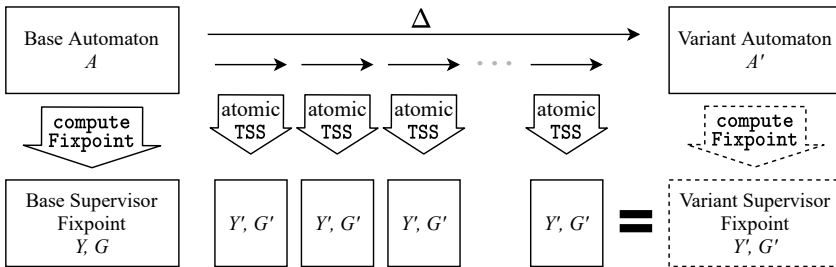


Figure 2.9: ITSS for non-atomic Δ .

Algorithm 2.11 Iterative Transformational Supervisor Synthesis (ITSS)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , model delta $\Delta = (X^+, X^-, \Sigma^+, \Sigma^-, \longrightarrow^+, \longrightarrow^-, X_0^+, X_0^-, X_m^+, X_m^-)$

Output: Variant supervisor $S' = (Y', \Sigma', \longrightarrow'_S, Y'_0, Y'_m)$, good states G' , variant automaton $A' = (X', \Sigma', \longrightarrow', X'_0, X'_m)$

- 1: $X' = X \cup X^+, \Sigma' = \Sigma \cup \Sigma^+, \longrightarrow' = \longrightarrow, X'_0 = X_0, X'_m = X_m$
- 2: $Y' = Y, G' = G$
- 3: **for all** $x^\delta \in X_0^+$ **do**
- 4: $(Y', G') = \text{TSSAIP}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', x^\delta)$
- 5: $X'_0 = X'_0 \cup \{x^\delta\}$
- 6: **end for**
- 7: **for all** $x^\delta \in X_0^-$ **do**
- 8: $(Y', G') = \text{TSSRIP}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', x^\delta)$
- 9: $X'_0 = X'_0 \setminus \{x^\delta\}$
- 10: **end for**
- 11: **for all** $x^\delta \in X_m^+$ **do**
- 12: $(Y', G') = \text{TSSAMP}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', x^\delta)$
- 13: $X'_m = X'_m \cup \{x^\delta\}$
- 14: **end for**
- 15: **for all** $x^\delta \in X_m^-$ **do**
- 16: $(Y', G') = \text{TSSRMP}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', x^\delta)$
- 17: $X'_m = X'_m \setminus \{x^\delta\}$
- 18: **end for**
- 19: **for all** $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^+$ **do**
- 20: $(Y', G') = \text{TSSAT}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', (x_{or}, \sigma, x_{tar}))$
- 21: $\longrightarrow' = \longrightarrow' \cup \{(x_{or}, \sigma, x_{tar})\}$
- 22: **end for**
- 23: **for all** $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^-$ **do**
- 24: $(Y', G') = \text{TSSRT}((X', \Sigma', \longrightarrow', X'_0, X'_m), Y', G', (x_{or}, \sigma, x_{tar}))$
- 25: $\longrightarrow' = \longrightarrow' \setminus \{(x_{or}, \sigma, x_{tar})\}$
- 26: **end for**
- 27: $S' = (Y', \Sigma' \setminus \Sigma^-, \longrightarrow \cap (Y' \times \Sigma \times Y'), X'_0 \cap Y', X'_m \cap Y')$
- 28: $A' = (X' \setminus X^-, \Sigma' \setminus \Sigma^-, \longrightarrow', X'_0, X'_m)$
- 29: **return** (S', G', A')

Order of applying atomic adaptations

There are some restrictions to the order in which the atomic adaptations can be applied during iterative TSS. Essentially, when an atomic adaptation is applied, this atomic adaptation needs to be a valid model delta. Let us consider the following model delta for the running example:

- $X^+ = \{p_5r_0\}$,

- $\Sigma^+ = \{e\}$,
- $\longrightarrow^+ = \{(p_4r_0, e, p_5r_0)\}$,
- $X^- = \emptyset, \Sigma^- = \emptyset, \longrightarrow^- = \emptyset, X_0^+ = \emptyset, X_0^- = \emptyset, X_m^+ = \emptyset, X_m^- = \emptyset$.

This model delta represents an added transition with an added event (e) to an added state (p_5r_0) from an existing state (p_4r_0). The model delta can be split into three atomic parts. We observe that the added state and added event need to be added to the automaton first, before the added transition can be added. Otherwise, the transition goes to an undefined state, or uses an undefined event, which means the model delta is not valid.

Therefore, the added states and added events are added first. Now, any state with added or removed initial property and any state with added or removed marked property will exist in this intermediate automaton. Also all added and removed transitions will go between defined states by defined events. Thus these atomic adaptations can be applied in any order. Once these adaptations have been applied, there will be no more transitions towards removed states, or transitions over removed events. Then finally the removed states and removed events can be removed, resulting in the final variant automaton.

The functioning of the atomic TSS algorithms is based on the assumption that the atomic adaptation is a valid model delta by the definitions in Section 2.3. To support the correct functioning of Algorithm 2.11, proof that each atomic adaptation that is applied is a valid model delta is given in Appendix B.

Note that even though the atomic adaptations of added/removed initial property, added/removed marked property, and added/removed transition can be applied in any order to come up with the same supervisor, the order in which they are applied may impact the computational efficiency. We do not optimize this order here, as the optimal order is likely highly dependent on the particular model and model delta. Therefore the adaptations are applied in the order shown in Algorithm 2.11, where the atomic TSS algorithms appear in the order in which they are introduced in Section 2.4.

2.5.2 Grouped Transformational Supervisor Synthesis

We observe that Algorithm 2.11 might not be very efficient when many atomic adaptations need to be considered. The main issue is that the SS and FRS algorithms are repeatedly called for input sets that are considerably similar to each other. This may notably occur when the plant description is given by a set of automata $\{P_1, P_2, \dots, P_n\}$. For our synthesis purpose, we take the synchronous product $A = P_1 || P_2 || \dots || P_n$, as mentioned in Section 2.2. If one of the automata P_i is adapted in an atomic manner, this might result in the model delta Δ to contain many atomic adaptations, due to synchronicity. A lot of these atomic adaptations in the synchronous system will be of the same type, e.g., an added transition in P_i can induce many added transitions in A . Therefore, we want to consider some adaptations at the same time as a group, rather than applying them one-by-one.

We partition the model delta into two disjoint subsets; $\Delta^\times \uplus \Delta^\circ = \Delta$, where \uplus denotes the disjoint union of the sets that are in the same field of both tuples. Δ^\times contains all atomic adaptations that, when applying the respective atomic TSS algorithm, require the SS or FRS algorithm. Δ° contains all other possible atomic adaptations outside Δ^\times , these require no reachability searches. As a result of the construction of the atomic TSS algorithms, all atomic adaptations in Δ° fit under the default cases of these algorithms, and all adaptations in Δ^\times match one of the (non-default) case statements in these algorithms. Formally, we can compute $\Delta^\times = (X^{+, \times}, X^{-, \times}, \Sigma^{+, \times}, \Sigma^{-, \times}, \longrightarrow^{+, \times}, \longrightarrow^{-, \times}, X_0^{+, \times}, X_0^{-, \times}, X_m^{+, \times}, X_m^{-, \times})$ for a model delta $\Delta = (X^+, X^-, \Sigma^+, \Sigma^-, \longrightarrow^+, \longrightarrow^-, X_0^+, X_0^-, X_m^+, X_m^-)$ as follows:

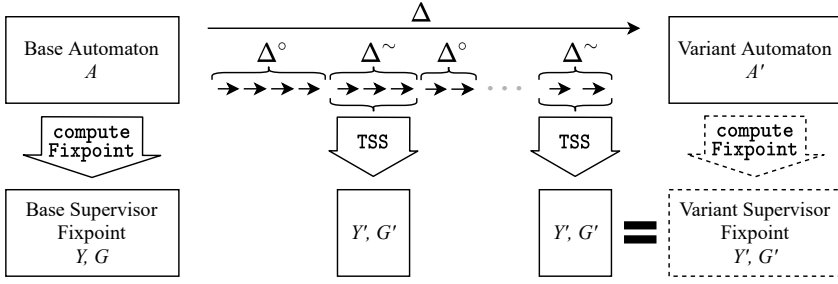
- $X^{+, \times} = \emptyset, X^{-, \times} = \emptyset, \Sigma^{+, \times} = \emptyset, \Sigma^{-, \times} = \emptyset$
- $\longrightarrow^{+, \times} = \{(x_{or}, \sigma, x_{tar}) | (x_{or} \in Y \wedge x_{tar} \in G \setminus Y) \vee (x_{or} \in G \wedge x_{tar} \in X \setminus G \wedge \sigma \in \Sigma_u) \vee (x_{or} \in X \setminus G \wedge x_{or} \neq x_{tar})\}$
- $\longrightarrow^{-, \times} = \{(x_{or}, \sigma, x_{tar}) | (x_{or} \in Y \wedge x_{tar} \in Y \wedge x_{or} \neq x_{tar}) \vee (x_{or} \in G \setminus Y \wedge x_{tar} \in G \wedge x_{or} \neq x_{tar}) \vee (x_{or} \in X \setminus G \wedge x_{tar} \in X \setminus G \wedge \sigma \in \Sigma_u \wedge x_{or} \neq x_{tar})\}$
- $X_0^{+, \times} = X_0^+ \cap (G \setminus Y)$
- $X_0^{-, \times} = X_0^- \cap Y$
- $X_m^{+, \times} = X_m^+ \cap (X \setminus G)$
- $X_m^{-, \times} = X_m^- \cap G$

Δ° is then constructed by all atomic adaptations in Δ outside Δ^\times .

When performing Grouped TSS (GTSS), we first want to apply all atomic adaptations in Δ° , as we can observe in the atomic TSS algorithms that no reachability searches need to be performed, and the supervisor states and good states remain unchanged. After all atomic adaptations in Δ° have been applied, we find the first atomic adaptation in Δ^\times for which, if we were to apply ITSS with model delta Δ^\times , one of the case conditions in Algorithms 2.5-2.10 holds. Instead of performing the corresponding case statements on only this atomic adaptation, within the case operation we first construct a set Δ^\sim , that contains all atomic adaptations in Δ^\times for which that same case condition in the same atomic TSS algorithm holds. So for example, if we have Δ^\times with nonempty set $X_0^{+, \times} \cap (G \setminus Y)$, then $X_0^{+, \times} \cap (G \setminus Y)$ is a set Δ^\sim . In addition to just atomic adaptations, the rationale applied in Algorithms 2.5-2.10 still holds for sets Δ^\sim . So, we take set Δ^\sim , and apply the respective case operation on this set, rather than doing this for each atomic adaptation one-by-one. This might influence the supervisor states and good states. Because the partitioning in Δ^\times and Δ° of Δ depends on those state sets, Δ° might be nonempty if we recompute it for the atomic adaptations in Δ that have not been applied yet. Thus, we once more apply adaptations in Δ° , and reiterate until all adaptations have been applied.

Figure 2.10 visualizes the grouped TSS method. First, all atomic adaptations are applied that require no reachability search. Then, a set of atomic adaptations Δ^\sim is applied at once. This repetition continues until the entire model delta is applied. At this point, the fixpoint for the variant supervisor has been found.

Algorithm 2.12 functions as described above. The added events and added states are added in line 1. After these have been applied, they are removed from the model

Figure 2.10: GTSS for non-atomic Δ .

Algorithm 2.12 Grouped Transformational Supervisor Synthesis (GTSS)

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_m, X_0)$, base supervisor $S = (Y, \Sigma, \longrightarrow_S, Y_0, Y_m)$, good states G , model delta $\Delta = (X^+, X^-, \Sigma^+, \Sigma^-, \longrightarrow^+, \longrightarrow^-, X_0^+, X_0^-, X_m^+, X_m^-)$

Output: Variant supervisor $S' = (Y', \Sigma', \longrightarrow', Y'_0, Y'_m)$, good states G' , variant automaton $A' = (X', \Sigma', \longrightarrow', X'_0, X'_m)$

- 1: $X' = X \cup X^+, \Sigma' = \Sigma \cup \Sigma^+, \longrightarrow' = \longrightarrow, X'_0 = X_0, X'_m = X_m$
 - 2: $\Delta = (\emptyset, X^-, \emptyset, \Sigma^-, \longrightarrow^+, \longrightarrow^-, X_0^+, X_0^-, X_m^+, X_m^-)$
 - 3: $Y' = Y, G' = G$
 - 4: **repeat**
 - 5: Compute $(\Delta^\times, \Delta^\circ = (X^{+, \circ}, X^{-, \circ}, \Sigma^{+, \circ}, \Sigma^{-, \circ}, \longrightarrow^{+, \circ}, \longrightarrow^{-, \circ}, X_0^{+, \circ}, X_0^{-, \circ}, X_m^{+, \circ}, X_m^{-, \circ}))$ for Δ, Y' , and G'
 - 6: $\longrightarrow' = (\longrightarrow \cup \longrightarrow^{+, \circ}) \setminus \longrightarrow^{-, \circ}, X'_0 = (X_0 \cup X_0^{+, \circ}) \setminus X_0^{-, \circ}, X'_m = (X_m \cup X_m^{+, \circ}) \setminus X_m^{-, \circ}$
 - 7: $\Delta = (\emptyset, X^-, \emptyset, \Sigma^-, \longrightarrow^+ \setminus \longrightarrow^{+, \circ}, \longrightarrow^- \setminus \longrightarrow^{-, \circ}, X_0^+ \setminus X_0^{+, \circ}, X_0^- \setminus X_0^{-, \circ}, X_m^+ \setminus X_m^{+, \circ}, X_m^- \setminus X_m^{-, \circ})$
 - 8: Compute $(Y', G', \longrightarrow', X'_0, X'_m)$ by applying one set of adaptations Δ^\sim in Δ^\times at once
 - 9: Remove Δ^\sim from Δ
 - 10: **until** $\Delta = (\emptyset, X^-, \emptyset, \Sigma^-, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
 - 11: $S' = (Y', \Sigma' \setminus \Sigma^-, \longrightarrow' \cap (Y' \times \Sigma \times Y'), X'_0 \cap Y', X'_m \cap Y')$
 - 12: $A' = (X' \setminus X^-, \Sigma' \setminus \Sigma^-, \longrightarrow', X'_0, X'_m)$
 - 13: **return** (S', G', A')
-

delta. As stated in Section 2.4.7, these adaptations do not influence the supervisor states and good states, seen in line 3. The model delta is partitioned in Δ^\times and Δ° as specified earlier in this section. All adaptations in Δ° are applied in line 6, and subsequently removed from the model delta in line 7. Note that Y' and G' remain unchanged. In line 8 a set of atomic adaptations Δ^\sim is applied; Y' and G' are calculated accordingly, and \longrightarrow', X'_0 , and X'_m are consequently updated. The calculation of Y' and G' is done by using slightly modified versions of Algorithms 2.5-2.10 that accept sets

of atomic adaptations. To avoid redundancy, we only provide the modified version of Algorithm 2.5 in the example below. The other atomic TSS algorithms are converted in the same manner. The set of atomic adaptations that has been applied, is removed from Δ in line 9. Now Δ^\times and Δ° are calculated once again, because the partitioning of the model delta is dependent on Y' and G' which are now modified. The steps are repeated until all atomic adaptations to \longrightarrow' , X'_0 , and X'_m have been applied. Finally, the supervisor automaton for the variant model and the variant automaton are constructed in lines 11 and 12 respectively.

Theorem 2.4 is proven for Algorithm 2.12 in Appendix C. In Appendix C also the proof is given that Algorithm 2.12 respects the order of applying adaptations discussed in Section 2.5.1.

Theorem 2.4 Given base automaton A , model delta Δ , synthesis result $(Y, G) = \text{computeFixpoint}(A)$, and $(\hat{S}, \hat{G}, \hat{A}) = \text{GTSS}(A, Y, G, \Delta)$; then $\hat{S} = S'$, $\hat{G} = G'$, and $\hat{A} = A'$, for $(S', G') = \text{SS}(A')$.

Example 2.3

Let us consider the case that Δ contains three atomic adaptations, all are states with added initial property; $X_0^+ = \{x^{\delta,1}, x^{\delta,2}, x^{\delta,3}\}$, with $x^{\delta,1} \in G \setminus Y$, $x^{\delta,2} \in G \setminus Y$, $x^{\delta,3} \in Y$. In our running example this could be states p_3r_0 , p_3r_1 , and p_2r_0 respectively. $x^{\delta,1}$ and $x^{\delta,2}$ require FRS , seen in line 3 of Algorithm 2.5, so they are in Δ^\times . $x^{\delta,3}$ triggers the default case in Algorithm 2.5, and thus it is in Δ° . Therefore, $x^{\delta,3}$ is applied first, resulting in:

- $X'_0 = X_0^+ \cup \{x^{\delta,3}\}$
- $X_0^+ = X_0^+ \setminus \{x^{\delta,3}\}$

Note that Y' and G' are not influenced as $x^{\delta,3}$ is in Δ° .

Now all adaptations in Δ° have been applied. $X_0^+ = \{x^{\delta,1}, x^{\delta,2}\}$ remains in the model delta. As $x^{\delta,2}$ and $x^{\delta,3}$ are both of the same case (line 2 of Algorithm 2.5), they are in a set $\Delta^\sim = \{x^{\delta,1}, x^{\delta,2}\}$. Consequently, these atomic adaptations will simultaneously be applied. Y' and G' are calculated in a modified version of Algorithm 2.5, given in Algorithm 2.13. X'_0 and X_0^+ are updated as follows:

- $X'_0 = X'_0 \cup \Delta^\sim$
- $X_0^+ = X_0^+ \setminus \Delta^\sim$

After applying these adaptations, all atomic model adaptations in the model have been applied and the model delta is empty, and Y' and G' are the fixpoint result for the variant automaton A' . For the running example, the good states would remain the same, $G' = G$, and Y' would be $\{p_0r_0, p_2r_0, p_3r_0, p_3r_1, p_2r_1\}$.

Algorithm 2.13 Grouped Transformational Supervisor Synthesis for Added Initial Property

Input: Base automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, supervisor states Y , good states G , added initial states X_0^+

Output: Variant supervisor states Y' , variant good states G'

```

1: switch  $X_0^+$ 
2:   case  $X_0^+ \cap (G \setminus Y) \neq \emptyset$  do
3:      $\Delta^- = X_0^+ \cap (G \setminus Y)$ 
4:      $Y' = \text{FRS}(G, \Sigma, \longrightarrow, Y \cup \Delta^-)$ 
5:      $G' = G$ 
6:   default do
7:      $Y' = Y, G' = G$ 
8:   end switch
9: return  $(Y', G')$ 

```

2.6 Experiments

As stated before, we can deal with any valid base automaton and model delta, so the TSS algorithm will always find a supervisor for the variant model. However, there are no guarantees that by applying TSS, we will find the supervisor more efficiently relative to simply performing a completely new synthesis. Therefore, we perform some experiments to investigate the potential reduction in computational effort by applying TSS.

For the experiments, a proof-of-concept implementation of the above synthesis algorithms and models of the case studies we describe below have been made in Matlab¹.

Before discussing case studies, we provide some practical notes in Section 2.6.1. In Section 2.6.2 we consider the Transfer Line model as an academic case study, and in Section 2.6.3 we consider a Lithography Machine Wafer Logistics controller as an industrial case study.

2.6.1 Practical notes

A proof-of-concept Matlab implementation of the above algorithms has been made. One modification from Thuijsman and Reniers (2020) to this chapter, is that now a linear

¹ The algorithms and models can be found here: https://github.com/sbthuijsman/JDEDS_TSS

complexity reachability search algorithm is used instead of quadratic. This is to enable a more realistic effort comparison between SS and TSS since linear search algorithms are more widely used than quadratic. In Thuijsman and Reniers (2020), a counter in the reachability search algorithms was introduced to express and compare the time effort of performing synthesis. After implementing the new reachability search algorithms, it was found that the previously used metric was not representative anymore of the time effort of performing synthesis. Experiments were performed with counters at several locations in the code, however none of these counters gave a proper representation of the time effort. Therefore, wall-clock time is used here to represent the time effort of performing synthesis. In order to enable fair comparisons between running times, obvious inefficient parts of the algorithm were improved. Still, using wall clock time instead of a counter makes the results more dependent on the implementation. The experiments were performed on an HP ZBook Studio G4 laptop, using an Intel i7 processor clocked at 2.8 GHz. Regarding memory, Matlab used around 1 GB of memory, regardless of the model size. Filesizes to store the automata and model deltas ranged from a few KB to a few MB.

Each synthesis algorithm requires a monolithic automaton as input, and the transformational synthesis algorithms require a model delta to the variant automaton. In practice, these inputs may not be readily available. E.g., the monolithic automaton A needs to be constructed from a component-wise specification, as discussed in Section 2.2. This is also the case for the conducted experiments. The inputs are computed in preparation of the experiments. We assume that for the transformational method, the input automaton A is maintained for the next iteration. For each transformational synthesis, A' is constructed beforehand in order to compute the model delta. Note that A' also needs to be constructed for the baseline case of performing a completely new synthesis. Computing the model delta is done by simple matrix subtractions and requires negligible computational effort. The preparatory computations are not included in the computational effort measurements, because this matches the experiments to the monolithic level discussed in the theoretical part and because computing the synchronous product is required for all methods.

The construction of A can be done by computing the complete synchronous product as shown in Equation 2.1. Practically however, often only the reachable part of A is constructed. Note that the computed supervisor is the same, as it only contains reachable states. The good state set G is influenced by only using the reachable part of A . Even when all states in X are reachable, it is still beneficial to store set G next to Y . When states are removed from G during synthesis, not all states in G may be reachable anymore through states in G . The model delta may be impacted by considering only the reachable parts of the automata or not. Also the computational effort of performing transformational or non-transformational supervisor synthesis may be impacted. Note that the transformational synthesis method works for both cases, as it allows for any pair of automata A and A' , independent of how they are constructed. We consider both options in the case studies below.

2.6.2 Transfer Line

We first consider the Transfer Line model from Wonham and Cai (2019) as an academic case study. In this model, products are being processed by two machines. Machine M1 takes products from the environment, and processes them. After processing, M1 places the product in buffer B1, which can hold up to three products. Machine M2 takes products from B1, processes them, and places them in buffer B2, which can hold only one product. Test unit TU takes products from B2, and tests them. If the product is accepted, it is released from the system. If the product is rejected, it goes back to B1. M1, M2, and TU start by a controllable event, and terminate by an uncontrollable event. Same as in Wonham and Cai (2019), M1, M2, and TU are modeled by plant automata and B1 and B2 by requirement automata. These automata are shown in Figure 2.11. The synchronous product over all automata is taken. For plantification, a single sink state is added to this synchronous product, to which all uncontrollable transitions are created whenever one of the requirement automata blocks an uncontrollable event of the plant. The resulting base automaton TL has 65 states and 200 transitions. Supervisor synthesis (Algorithm 2.1) for this base automaton requires 0.59 milliseconds, which is the mean runtime of 100 executions of SS .

The following five variant automata, TL'_1 to TL'_5 , have been generated by making adaptations to TL .

- TL'_1 : Reduced capacity of B1 to two products: state x_3 of automaton $B1$ removed, and transitions $(x_2, \text{stop_M1}, x_3)$, $(x_2, \text{reject}, x_3)$, $(x_3, \text{start_M2}, x_2)$ removed.
- TL'_2 : Increased capacity of B2 to two products: added a state x_2 and transitions $(x_1, \text{stop_M2}, x_2)$, $(x_2, \text{start_TU}, x_1)$ to B2.
- TL'_3 : B1 initially holds one product instead of zero: removed initial property of state x_0 , and added initial property to state x_1 in automaton B1.
- TL'_4 : TU may send the product to B2 upon completion: added event 'retest', added transition $(x_1, \text{retest}, x_2)$ to TU, and added transition $(x_0, \text{retest}, x_1)$ to B2.

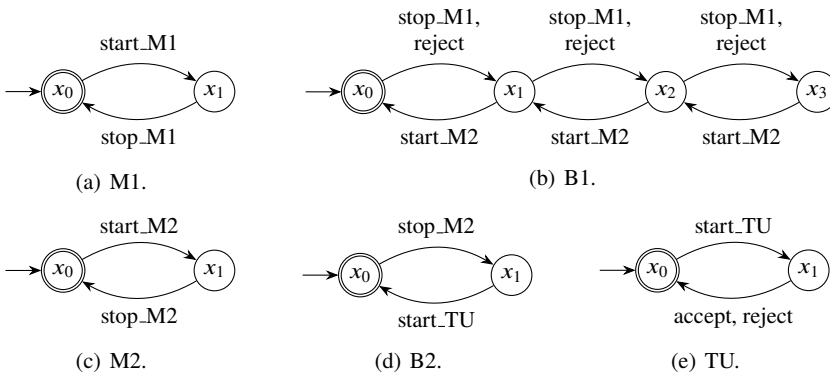


Figure 2.11: Transfer Line automata.

- TL'_5 : Capacity of B1 and B2 is two products each: removed state x_3 and transitions $(x_2, \text{stop_M1}, x_3)$, $(x_2, \text{reject}, x_3)$, $(x_3, \text{start_M2}, x_2)$ from B1, state x_2 and transitions $(x_1, \text{stop_M2}, x_2)$, $(x_2, \text{start_TU}, x_1)$ are added to B2.

As the TSS methods are on a monolithic state space, these adaptations for the individual automata are converted to adaptations on the synchronous state space for the experiments, as discussed in Section 2.6.1. For the base model and each variant model, all states constructed during the synchronous product (Equation 2.1) are reachable. Therefore, for these results it does not matter if A is completely constructed as in Equation 2.1, or only the reachable part. The experiment cases we presented are the same as presented in Thuijsman and Reniers (2020), however the experiments are executed differently as pointed out in Section 2.6.1.

For each of the variant automata, the number of states and transitions, as well as the sizes of the nonempty sets in the model delta, are given in Table 2.1. One might expect that for constructing TL'_1 only states and transitions would be removed. However, it can be observed that some transitions have been added. These are new transitions towards the sink state, as the buffer now may overflow from different states. For each variant model, a supervisor has been synthesized three times. First, doing a completely new synthesis by applying SS given in Algorithm 2.1, the second and third by using the synthesis result of the base automaton and applying $ITSS$ (Algorithm 2.11), and $GTSS$ (Algorithm 2.12) respectively. For each model, the three synthesized supervisors have

Table 2.1: Experimental results of performing SS , $ITSS$, and $GTSS$ on five variant models of the Transfer Line model.

Evolution	Variant model size	Model delta size	Var. model runtime [ms]			% change GTSS from SS
			SS	ITSS	GTSS	
TL to TL'_1	$ X' = 49$ $ \rightarrow' = 148$	$ X^- = 16$ $ \rightarrow^+ = 16$ $ \rightarrow^- = 68$	0.41	15.08	2.40	487
TL to TL'_2	$ X' = 97$ $ \rightarrow' = 308$	$ X^+ = 32$ $ \rightarrow^+ = 124$ $ \rightarrow^- = 16$	0.63	53.35	4.22	573
TL to TL'_3	$ X' = 65$ $ \rightarrow' = 200$	$ X_0^+ = 1$ $ X_0^- = 1$	0.41	0.24	0.36	-13
TL to TL'_4	$ X' = 65$ $ \rightarrow' = 232$	$ \Sigma^+ = 1$ $ \rightarrow^+ = 32$	0.39	7.16	1.11	184
TL to TL'_5	$ X' = 73$ $ \rightarrow' = 228$	$ X^+ = 24$ $ X^- = 16$ $ \rightarrow^+ = 108$ $ \rightarrow^- = 80$	0.47	56.55	5.44	1058

the exact same automaton specification. For each of the syntheses, the runtime is shown in milliseconds in Table 2.1. This is the mean over 100 runs for each synthesis. The rightmost column shows the percentage change in runtime of using GTSS compared to SS. A positive value indicates an increase in computational effort, and a negative value indicates a reduction. Note that the efficiency of the TSS algorithms will be influenced by the order in which the adaptations are applied, this has not been optimized for the experiments as this would entail a completely new study.

We observe that in most cases applying ITSS requires considerably more runtime than applying SS. This was already addressed in Section 2.5, and led to the introduction of the GTSS algorithm. We observe that for this use case, for most variant models GTSS requires considerably less runtime than ITSS, but still requires more runtime than SS. Regardless, the absolute runtimes are very low for this small Transfer Line system. To further investigate the efficiency of the transformational algorithms, a larger system is studied next, in Section 2.6.3.

2.6.3 Lithography Machine Wafer Logistics

Next we present an industrial case study. This case study is performed using models from ASML. ASML is the world-leading manufacturer of lithography machines, which are used in the semiconductor industry to produce integrated circuits. These circuits are printed on silicon wafers. The movement of these wafers through the machine is called the Wafer Logistics, which is studied in van der Sanden et al. (2015) and van der Schriek (2018). The controller of the Wafer Logistics is constructed using Analytical Software Design (ASD) (Broadfoot and Hopcroft 2003). van der Schriek (2018) presents a study on how these ASD models of the components of the Wafer Logistics controller evolve over time. In this study equivalent automata models are constructed in CIF (van Beek et al. 2014), CIF is part of the Eclipse Supervisory Control Engineering ToolkitTM, that are suitable for supervisory controller synthesis. These automata models are constructed for the variation points that the ASD models evolved to over a number of years. We use these automata models here, to investigate the efficiency of TSS in this industrial setting.

The control of the Wafer Logistics is modular, i.e., the various physical components are steered by multiple supervisory controllers. Together these controllers and components operate in synchrony to achieve the desired behavior. Since TSS is a monolithic method, for these experiments we just consider synthesizing a supervisory controller for a single component. *Component B* of van der Schriek (2018) is selected to perform the experiments on, based on its large but manageable state space size, the number of variation points, and the variety within the model deltas. The first 11 variation points of this model are taken, to investigate 10 adaptations. Opposed to the Transfer Line experiment, where each variant model was an adaptation of the same base model, we now consider incremental adaptations. So we start with the evolution from B_1 to B_2 , then from B_2 to B_3 , from B_3 to B_4 , and so on. To provide an indication of the model size, B_1 contains 5 plant automata, that respectively have:

1. 2 states and 2 transitions,
2. 2 states and 4 transitions,
3. 2 states and 3 transitions,
4. 15 states and 72 transitions,
5. 16 states and 128 transitions.

Model *B1* also contains 4 requirement automata, that respectively have:

1. 2 states and 2 transitions,
2. 2 states and 2 transitions,
3. 3 states and 5 transitions,
4. 3 states and 5 transitions.

Additionally 3 state transition exclusion invariant requirements (Markovski et al. 2010) are specified. After computing the synchronous product (Equation 2.1), and adding a sink state for plantification, *B1* has 69 121 states and 1 038 228 transitions. Performing *SS* (Algorithm 2.1) requires 129 milliseconds, which is the mean over 100 executions of *SS*.

Unlike the Transfer Line model, for the models of Component B not all states are reachable when automaton *A* is constructed by Equation 2.1. We perform two studies, in Section 2.6.3.1 we consider the case that the complete automaton is used as an input to the synthesis algorithms and in Section 2.6.3.2 we consider that only the reachable part of the automaton is used as input.

2.6.3.1 Complete state space

In this section we perform synthesis on the complete state space that is constructed by taking the synchronous product of the Component B Wafer Logistics model. The experimental results are presented in Table 2.2. *ITSS* has not been performed, as its inefficiency compared to *GTSS* has been discussed in Section 2.5.2, and shown in the Transfer Line experiments. Each runtime value is the mean over 100 syntheses. We observe that for all evolutions in this case study, applying *GTSS* is less efficient than applying *SS* to compute the supervisor for the variant model.

2.6.3.2 Reachable state space

In this section we perform synthesis using only the reachable part of the synchronous product of the Component B Wafer Logistics model. The complete state space of model *B1* consists of 69 121 states and 1 038 228 transitions. The reachable part of this automaton has 59 185 states and 888 780 transitions. Note that also the model delta sizes are influenced by only considering the reachable part of the base and variant model automata. The results are shown in Table 2.3. Each runtime value is the mean over 100 syntheses. Once more, for all evolutions in this experiment applying *GTSS* is less efficient than applying *SS* to compute the supervisor for the variant model.

Compared to the experiment on the complete state space, discussed in Section 2.6.3.1, the absolute time to perform synthesis is reduced for both SS and GTSS.

Table 2.2: Experimental results of performing SS and GTSS for evolution of the complete Component B Wafer Logistics model.

Evolution	Variant model size	Model delta size	Var. model runtime [ms]		% change GTSS from SS
			SS	GTSS	
<i>B1 to B2</i>	$ X' = 69\,121$ $ \rightarrow' = 1\,042\,548$	$ \Sigma^+ = 1$ $ \rightarrow^+ = 4\,320$	138	140	1
<i>B2 to B3</i>	$ X' = 77\,761$ $ \rightarrow' = 1\,154\,772$	$ X^+ = 8\,640$ $ \Sigma^+ = 8$ $ \rightarrow^+ = 112\,224$	153	1256	723
<i>B3 to B4</i>	$ X' = 73\,441$ $ \rightarrow' = 1\,102\,980$	$ X^- = 4\,320$ $ \Sigma^- = 1$ $ \rightarrow^- = 51\,792$	147	1126	668
<i>B4 to B5</i>	$ X' = 73\,441$ $ \rightarrow' = 1\,121\,412$	$ \Sigma^+ = 4$ $ \rightarrow^+ = 18\,432$	146	230	58
<i>B5 to B6</i>	$ X' = 73\,441$ $ \rightarrow' = 1\,121\,412$	$ \Sigma^+ = 1$ $ \Sigma^- = 1$ $ \rightarrow^+ = 4\,320$ $ \rightarrow^- = 4\,320$	147	270	83
<i>B6 to B7</i>	$ X' = 73\,441$ $ \rightarrow' = 1\,130\,052$	$ \Sigma^+ = 3$ $ \Sigma^- = 1$ $ \rightarrow^+ = 12\,960$ $ \rightarrow^- = 4\,320$	147	318	117
<i>B7 to B8</i>	$ X' = 78\,337$ $ \rightarrow' = 1\,200\,980$	$ X^+ = 4\,896$ $ \Sigma^+ = 5$ $ \Sigma^- = 1$ $ \rightarrow^+ = 75\,824$ $ \rightarrow^- = 4\,896$	156	973	522
<i>B8 to B9</i>	$ X' = 73\,441$ $ \rightarrow' = 1\,102\,980$	$ X^- = 4\,896$ $ \Sigma^- = 10$ $ \rightarrow^- = 98\,000$	146	1650	1031
<i>B9 to B10</i>	$ X' = 78\,337$ $ \rightarrow' = 1\,200\,980$	$ X^+ = 4\,896$ $ \Sigma^+ = 10$ $ \rightarrow^+ = 98\,000$	155	960	521
<i>B10 to B11</i>	$ X' = 83\,233$ $ \rightarrow' = 1\,267\,012$	$ X^+ = 4\,896$ $ \Sigma^+ = 4$ $ \rightarrow^+ = 66\,032$	165	773	368

Table 2.3: Experimental results of performing SS and GTSS for evolution of the reachable part of the Component B Wafer Logistics model.

Evolution	Variant model size	Model delta size	Var. model runtime [ms]		% change GTSS from SS
			SS	GTSS	
<i>B1 to B2</i>	$ X' = 59\,185$ $ \rightarrow' = 892\,460$	$ \Sigma^+ = 1$ $ \rightarrow^+ = 3\,680$	94	109	16
<i>B2 to B3</i>	$ X' = 66\,545$ $ \rightarrow' = 988\,252$	$ X^+ = 7\,360$ $ \Sigma^+ = 8$ $ \rightarrow^+ = 95\,792$	106	1065	901
<i>B3 to B4</i>	$ X' = 62\,865$ $ \rightarrow' = 944\,036$	$ X^- = 3\,680$ $ \Sigma^- = 1$ $ \rightarrow^- = 44\,216$	100	848	751
<i>B4 to B5</i>	$ X' = 62\,865$ $ \rightarrow' = 959\,732$	$ \Sigma^+ = 4$ $ \rightarrow^+ = 15\,696$	100	188	88
<i>B5 to B6</i>	$ X' = 62\,865$ $ \rightarrow' = 959\,732$	$ \Sigma^+ = 1$ $ \Sigma^- = 1$ $ \rightarrow^+ = 3\,680$ $ \rightarrow^- = 3\,680$	101	182	80
<i>B6 to B7</i>	$ X' = 62\,865$ $ \rightarrow' = 967\,092$	$ \Sigma^+ = 3$ $ \Sigma^- = 1$ $ \rightarrow^+ = 11\,040$ $ \rightarrow^- = 3\,680$	100	227	126
<i>B7 to B8</i>	$ X' = 67\,033$ $ \rightarrow' = 1\,027\,520$	$ X^+ = 4\,168$ $ \Sigma^+ = 5$ $ \Sigma^- = 1$ $ \rightarrow^+ = 64\,596$ $ \rightarrow^- = 4\,168$	108	771	616
<i>B8 to B9</i>	$ X' = 62\,865$ $ \rightarrow' = 944\,036$	$ X^- = 4\,168$ $ \Sigma^- = 10$ $ \rightarrow^- = 83\,484$	99	1266	1175
<i>B9 to B10</i>	$ X' = 67\,033$ $ \rightarrow' = 1\,027\,520$	$ X^+ = 4\,168$ $ \Sigma^+ = 10$ $ \rightarrow^+ = 83\,484$	107	814	663
<i>B10 to B11</i>	$ X' = 71\,201$ $ \rightarrow' = 1\,083\,780$	$ X^+ = 4\,168$ $ \Sigma^+ = 4$ $ \rightarrow^+ = 56\,260$	112	649	478

2.6.4 Correction

Current chapter is based on Thuijsman and Reniers (2022). Unfortunately, the algorithmic implementation to obtain the experimental results presented in Thuijsman and Reniers (2022) was flawed. Although the algorithms were still functionally correct, because of the flawed implementation their computational efficiency was significantly worse than that of a proper implementation. Since we aim to improve computational efficiency, and use experiments to indicate that improvement, the implementation should be properly efficient. Note that the theoretical parts of Thuijsman and Reniers (2022) are still correct, the flaw only affects the timing measurements in the experimental part of the paper. The current chapter contains the results using an improved implementation. A Correction to Thuijsman and Reniers (2022) is in press. In this section, we explain the flaw and discuss what can be learned from comparing the efficiency of the two implementations.

The reachability search algorithms (Algorithms 2.3 and 2.4) were not properly implemented. This is because an improper data structure was used to store the transition relation. In the flawed implementation, an *edge list* structure was used: a list of triples, each specifying an origin state, event, and target state. Using an edge list, each time the neighbors of some state are evaluated (i.e., lines 6-7 in Algorithm 2.3), an iteration is performed over all transitions, leading to a complexity of $O(|X| \cdot |\longrightarrow|)$ for the algorithm overall. Instead, the transition relation should be stored using an *adjacency list*: each index in the list corresponds to a state index and stores all neighbors of that state. When an adjacency list is used, no iteration is required to find the neighbors of a state, and a complexity of $O(|X| + |\longrightarrow|)$ for the reachability search can be obtained (Kleinberg and Tardos 2005).

The implementation has been corrected. The code repository² has been updated accordingly. We have repeated the experiments from Thuijsman and Reniers (2022). Tables 2.1, 2.2, and 2.3 respectively present the updated results relative to Tables 1, 2, and 3 in Thuijsman and Reniers (2022).

Note that both SS and TSS complete the experimental cases much quicker using the improved implementation. Nevertheless, it is interesting to note that in Thuijsman and Reniers (2022), using the flawed implementation, generally TSS was more efficient than SS, and after correcting generally SS was more efficient than TSS. There are two reasons that explain why TSS becomes less efficient relative to SS when using the improved implementation instead of the flawed one:

1. TSS generally divides the synthesis problem into multiple smaller syntheses and reachability searches. Such a divide-and-conquer strategy can generally be expected to be more effective for algorithms with a higher (worst-case) complexity. For example, consider a problem of size $n = 100$, and an algorithm that solves this problem in n^2 steps, monolithically solving requires $100^2 = 10^4$ steps. If we can divide this problem into ten subproblems, each of size 10, then the problem can be solved in $10 \cdot 10^2 = 10^3$ steps, which is more efficient than the monolithic

² https://github.com/sbthuijsman/JDEDS_TSS

solution. However, if we have an algorithm that solves the problem in n steps, both the monolithic and divided solution require 100 steps. So, dividing the problem becomes less effective when lowering the algorithm's complexity. Hence, the efficiency of TSS relative to SS became worse when lowering the complexity of the reachability searches.

2. Operations necessary for TSS but not SS, such as evaluating the switch-cases in the atomic TSS algorithms, relatively take more time when using the adjacency list implementation. Note that just speeding up the reachability search already increased the time these overhead operations take relative to the total computation time. Furthermore, operations to change the transition function, that are likely more frequently performed in TSS than SS, are less efficient for the adjacency list implementation relative to the edge list implementation. For example, in the flawed implementation just a single edge list was maintained to keep track of the current system instance, but in the improved implementation three adjacency lists (all transitions, all transitions reverse, and all uncontrollable transitions reverse) need to be maintained throughout.

2.7 Conclusions

Supervisory controller synthesis is a means to compute correct-by-construction controllers for discrete-event systems. As these systems evolve over time, we want to be able to efficiently generate a supervisor each time the system is adapted. We consider the case that a supervisor has been synthesized for a given base model, after which this base model is adapted to some variant model. Model deltas are used to describe the difference between the base and the variant model. A notion of atomic adaptations is introduced, where the model delta can be described by a single, indivisible change in the automaton specification. For these atomic model adaptations, algorithms are provided to compute the supervisor for the variant model in a transformational manner. The atomic model adaptations can be iterated over to transformationally compute a supervisor for any model delta that contains a number of atomic model adaptations. It is discussed why, and shown in experiments that, only purely iterating over these atomic adaptations is not efficient. Therefore a method is presented where groups of adaptations are considered. Indeed it is experimentally shown that the grouped method, GTSS, was more efficient than the iterative method, ITSS.

Even though GTSS may be more efficient than ITSS, the experiments show that GTSS still remains less efficient than the basic SS method. Perhaps when the model delta is very small, or the model delta is such that (almost) no calculations need to be performed, i.e., for some model delta Δ^\times as constructed in Section 2.5.2 is empty, then possibly TSS may outperform basic SS. However, as the experiments point out, this is in practice typically not the case. Therefore, based on these results, it is not recommended to apply TSS as presented in this chapter, but rather just use the basic SS method, also for systems under evolution.

Chapter 3

Transformational supervisor localization

Abstract Supervisor localization can be applied to distribute a monolithic supervisor into local supervisors. Performing supervisor localization can be computationally costly. In this work, we consider systems that evolve over time. We study how to reuse the results from a previous supervisor localization, to more efficiently compute local supervisors when the system is adapted. We call this approach transformational supervisor localization, and present algorithms for the procedure. The efficiency of the procedure is experimentally evaluated.

3.1 Introduction

Supervisory control theory, as introduced by Ramadge and Wonham (1987), is a model-based approach to control discrete-event (dynamic) systems. Typically, cyber-physical systems are modeled. By applying *supervisor synthesis* on a model of an uncontrolled system (plant) and system requirements, a correct-by-construction supervisor is obtained. This supervisor enables/disables events such that the requirements are always adhered to, and some more behavioral properties apply to the controlled system such as: nonblockingness, controllability, and maximal permissiveness (Cassandras and Lafor-tune 2021). The most straightforward approach is *monolithic* supervisor synthesis, which computes a single global supervisor that controls all components and enforces all requirements.

Large, global controllers may be undesirable in practice. As such, many modern control systems are distributed over a number of agents (Moormann et al. 2021). These agents may act locally based on their own observations and control strategies. Through *supervisor localization* (SL), introduced by Cai and Wonham (2010), local supervisors for the individual agents are computed from the monolithic supervisor, that together

This chapter is strongly based on: Thuijsman, Sander B.; Cai, Kai; Reniers, Michel A.: Transformational supervisor localization. In: *IEEE Control Systems Letters* 7 (2023), p.1682–1687. IEEE. – URL <https://doi.org/10.1109/LCSYS.2023.3278248>

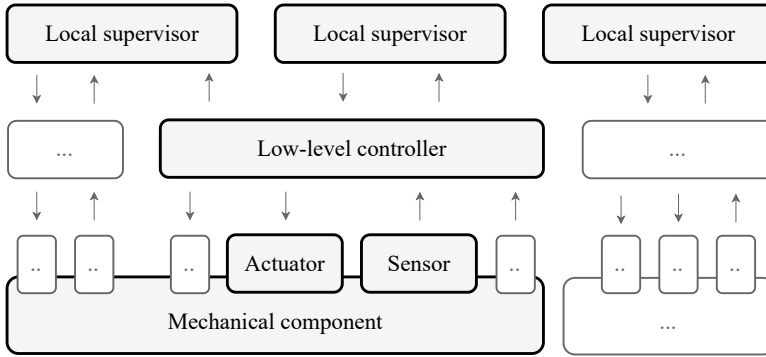


Figure 3.1: Control architecture using local supervisors.

achieve the same controlled behavior as the monolithic supervisor. Figure 3.1 shows a control architecture of a system using three local supervisors, which is a modification from the monolithic control architecture shown in Figure 1.2. The controlled behavior of the system using local supervisors is the same as that of the system using a monolithic supervisor. However, each of the local supervisors is smaller and relates to only a part of the system, improving the interpretability and maintainability of the control system.

SL is an extension to *supervisor reduction*, introduced by Su and Wonham (2004), which converts a supervisor automaton to a smaller automaton (with less states) that is control equivalent to the original automaton. We present preliminaries on supervisor localization/reduction in Section 3.2.

In this work, we slightly modify the SL algorithm from Su and Wonham (2004) and Cai and Wonham (2010) to be able to initialize it in a way such that it has to do less calculations/loops, which benefits the method that we are going to introduce. Furthermore, because it is desirable to obtain small (in terms of number of states) local supervisors, we show that the local supervisors obtained by SL are maximally reduced. These novel extensions to SL are presented in Section 3.3.

In Chapter 2 a transformational supervisor synthesis approach was introduced. Such a transformational approach deals with cyber-physical systems that evolves over time. For a system that evolves over time, results of previous computations, such as synthesis, may not be valid anymore once the system is adapted. In this case, transformational methods can be applied that reuse the output of previous calculations to more efficiently compute the result of some algorithm, rather than computing it from scratch. The general idea is that the previous result is *transformed* into the new result, using knowledge on how the system is adapted.

In this chapter, we investigate *transformational supervisor localization* (TSL). We assume a *base model*, on which (T)SL has already been performed. The base model is adapted such that a *variant model* is obtained. The goal is to use the localization output of the base model, to more efficiently compute local supervisors for the variant model. The formal problem definition is given in Section 3.4. We present algorithms

for TSL and prove their correctness in Section 3.5. The computational benefit of TSL is evaluated by a use case in Section 3.6.

3.2 Preliminaries

In the following we discuss the preliminaries on SL. We first provide automata definitions for plant and (monolithic) supervisor. The plant is assumed to be a composition of agents. The goal is to generate local supervisors that each supervise an agent. This is done by grouping states of the monolithic supervisor into cells, that are consistent in the enablement and disablement of events controlled by the respective agent. These cells are the states of the local supervisor. Together, the behavior of the system under control by the local supervisors is the same as that of the monolithic supervisor. For details we refer to Cai and Wonham (2010).

The plant is defined by a finite state automaton $G = (Q, \Sigma, \delta, q_0, Q_m)$, where Q is the finite state set, Σ is the finite event set, q_0 is the initial state, and Q_m is the subset of marked states. $\delta : Q \times \Sigma \rightarrow Q$ is the (partial) transition function. We denote $\delta(q, \sigma)!$ if $\delta(q, \sigma)$ is defined. We extend this notation to $\delta : Q \times \Sigma^* \rightarrow Q$, and write $\delta(q, s)$ for $s \in \Sigma^*$ to indicate sequences of transitions. We consider the case that G is formed by a composition of n agents, that each have local events: $\bigcup_{k \in \{1, \dots, n\}} \Sigma_k = \Sigma$, from which a subset are locally controllable $\Sigma_{c,k} \subseteq \Sigma_k$. So, agent k has local events Σ_k , from which $\Sigma_{c,k}$ are the locally controllable events. We assume a monolithic supervisor is provided for plant G , defined by finite state automaton $S = (X, \Sigma, \xi, x_0, X_m)$. For the purpose of the algorithms in this work, we assume the states are numbered/indexed, i.e., $X = \{x_0, x_1, \dots\}$.

We use the following functions from Cai and Wonham (2010):

- $E : X \rightarrow 2^\Sigma$, where $E(x) = \{\sigma \in \Sigma \mid \xi(x, \sigma)!\}$.
- $D_k : X \rightarrow 2^{\Sigma_{c,k}}$, and $D_k(x) = \{\sigma \in \Sigma_{c,k} \mid \neg \xi(x, \sigma) \wedge (\exists s \in \Sigma^*) : (\xi(x_0, s) = x \wedge \delta(q_0, s\sigma)!\}\}$.
- $M : X \rightarrow \{0, 1\}$, where $M(x) = 1$ iff $x \in X_m$.
- $T : X \rightarrow \{0, 1\}$, where $T(x) = 1$ iff $(\exists s \in \Sigma^*) : (\xi(x_0, s) = x \wedge \delta(q_0, s) \in Q_m)$.

E indicates events enabled by the supervisor in state x . D_k indicates the events from $\Sigma_{c,k}$ disabled by the supervisor in state x . M determines if a state is marked in S , and T determines if some corresponding state is marked in G .

We define control consistency relation $\mathcal{R}_k \subseteq X \times X$ (for agent k): for every $x, x' \in X$, $(x, x') \in \mathcal{R}_k$ iff:

$$E(x) \cap D_k(x') = \emptyset = E(x') \cap D_k(x) \quad (3.1)$$

$$T(x) = T(x') \implies M(x) = M(x') \quad (3.2)$$

Cover $C_k = \{X_i \subseteq X \mid i \in I_k\}$, where I_k is an index set suitable to the amount of sets X_i in cover C_k , is called a *control cover* with respect to some Σ_k iff:

- (i) $(\forall i \in I_k, \forall x, x' \in X_i)(x, x') \in \mathcal{R}_k$
(ii) $(\forall i \in I_k, \forall \sigma \in \Sigma) [((\exists x \in X_i)\xi(x, \sigma))! \implies$
 $(\exists j \in I_k)(\forall x' \in X_i)\xi(x', \sigma)! \implies \xi(x', \sigma) \in X_j]$

If a control cover C is a partition on X , it is called a *control congruence*.

In this work we frequently address a *singleton cover* $C = \{\{x\} | x \in X\}$, which trivially always is a control congruence.

We call a set of states in a cover a *cell*. In our notation we use $[x]_C$ to refer to the set of states contained in the same cell as x in cover C , or simply $[x]$ if there is no ambiguity.

Given a control congruence C_k , a local supervisor LOC_k is computed as follows (simplified from Su and Wonham (2004)): $LOC_k = (C_k, \Sigma, \eta_k, y_{0,k}, Y_{m,k})$, where: $\eta_k : C_k \times \Sigma \rightarrow C_k$, with $\eta_k(\pi_1, \sigma) = \pi_2$ iff $(\exists x \in \pi_1) : \xi(x, \sigma) \in \pi_2$; $y_{0,k} = [x_0]$; and $Y_{m,k} = \{[x] | x \in X_m\}$. A local supervisor is deterministic as a result of condition (ii) for the control cover.

The set of local supervisors $\{LOC_k | 1 \leq k \leq n\}$ constructed in this way is *control equivalent* to S with respect to G (Cai and Wonham 2010):

$$L(G) \cap \bigcap_{1 \leq k \leq n} L(LOC_k) = L(S) \cap L(G) \quad (3.3)$$

$$L_m(G) \cap \bigcap_{1 \leq k \leq n} L_m(LOC_k) = L_m(S) \cap L_m(G) \quad (3.4)$$

$L(A)$ and $L_m(A)$ respectively denote the language and the marked language of automaton A (Cassandras and Lafortune 2021).

3.3 Supervisor localization

In the process of SL, for each agent, a control congruence is computed and subsequently the local supervisor is generated. We can use the definitions and functions from Section 3.2 to perform the localization algorithm, shown in Algorithm 3.1, which makes calls to Algorithm 3.2 (Cai and Wonham 2010)¹. Note that, e.g., the X on line 1 implicitly originates from automaton S . A ‘continue’ ends current execution and the function goes to the next iteration of the nearest enclosing for-loop. A ‘return’ ends current call to the algorithm and the specified values are returned to the parent routine.

¹ Relative to Su and Wonham (2004) and Cai and Wonham (2010) some minor changes have been made to lines 1, 2, and 7 of Algorithm 3.2 for correctness.

Algorithm 3.1 localize**Input:** $G, S, \Sigma_{c,k}$, initial C_k **Output:** Control congruence C_k

```

1: for  $i = 0$  to  $|X| - 2$  do
2:   if  $i > \min(\{m|x_m \in [x_i]\})$  then continue; end
3:   for  $j = i + 1$  to  $|X| - 1$  do
4:     if  $j > \min(\{m|x_m \in [x_j]\})$  then continue; end
5:      $W = \emptyset$ 
6:      $(flag, W) = \text{check\_merge}(x_i, x_j, W, i, \xi, C_k)$ 
7:     if  $flag$  then
8:        $C_k = \left\{ [x] \cup \bigcup \{ [x'] \mid \{(x, x'), (x', x)\} \cap W \neq \emptyset \} \mid [x], [x'] \in C_k \right\}$ 
9:     end
10:  end
11: end
12: return  $C_k$ 

```

Algorithm 3.2 check_merge**Input:** x_i, x_j , waiting list W, i, ξ, C_k **Output:** mergeability Boolean $flag, W$

```

1: for all  $x_p \in [x_i] \cup \bigcup \{ [x] \mid \{(x, x'_i), (x'_i, x)\} \cap W \neq \emptyset, x'_i \in [x_i] \}$  do
2:   for all  $x_q \in [x_j] \cup \bigcup \{ [x] \mid \{(x, x'_j), (x'_j, x)\} \cap W \neq \emptyset, x'_j \in [x_j] \}$  do
3:     if  $\{(x_p, x_q), (x_q, x_p)\} \cap W \neq \emptyset$  then continue; end
4:     if  $(x_p, x_q) \notin \mathcal{R}_k$  then return  $(false, W)$ ; end
5:      $W = W \cup \{(x_p, x_q)\}$ 
6:     for all  $\sigma \in E(x_p) \cap E(x_q)$ 
7:       if  $[\xi(x_p, \sigma)] = [\xi(x_q, \sigma)]$  or  $\{(\xi(x_p, \sigma), \xi(x_q, \sigma)), (\xi(x_q, \sigma), \xi(x_p, \sigma))\} \cap W \neq \emptyset$  then continue; end
8:       if  $\min(\{m|x_m \in [\xi(x_p, \sigma)]\}) < i$  or  $\min(\{m|x_m \in [\xi(x_q, \sigma)]\}) < i$  then return  $(false, W)$ ; end
9:        $(flag, W) = \text{check\_merge}(\xi(x_p, \sigma), \xi(x_q, \sigma), W, i, \xi, C_k)$ 
10:      if not  $flag$  then return  $(false, W)$ ; end
11:    end
12:  end
13: end
14: return  $(true, W)$ 

```

Example 3.1

We consider the supervisor automaton shown in Figure 3.2(a). The states are represented by circles. The dangling incoming arrow indicates x_0 is the initial state. Transitions are shown by arrows between states with the respective event label. To simplify the examples, no states are marked and all events are controllable.

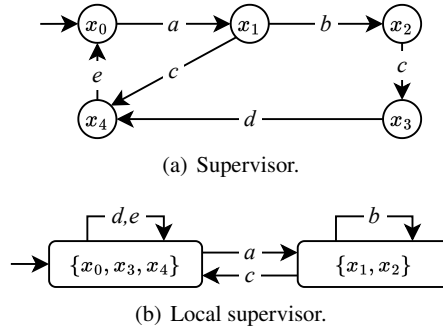


Figure 3.2: Automata of Example 3.1.

We consider the case that there is an agent (numbered 1) whose set of local controllable events includes all events, i.e., $\Sigma_{c,1} = \Sigma_1 = \Sigma = \{a, b, c, d, e\}$. Let us consider the case that the supervisor disables event c in state x_0 , and disables event a in state x_2 . There are no disablements in the other states, i.e., the supervisor permits the same events as the plant in those states. So, $D_1(x_0) = \{c\}$, $D_1(x_2) = \{a\}$, $D_1(x_1) = D_1(x_3) = D_1(x_4) = \emptyset$.

To compute the local supervisor, we perform the localization algorithm initialized with a singleton cover $\{\{x_0\}, \dots, \{x_4\}\}$. First, mergeability of x_0 and x_1 is checked. These states are not mergeable, since event c is disabled in x_0 but enabled in x_1 . Also x_0 and x_2 are not mergeable. x_0 is mergeable with x_3 and they are subsequently merged. Next, $\{x_0, x_3\}$ is merged with x_4 to form cell $\{x_0, x_3, x_4\}$. Finally x_1 and x_2 are merged, and no more merges are possible so the algorithm terminates. Using the resulting control congruence, a local supervisor is constructed, which is displayed in Figure 3.2(b).

In Cai and Wonham (2010) the localization algorithm is initiated with a singleton cover. However, in this work we will also initialize the algorithm with non-singleton covers, to benefit the efficiency of the transformational method that we are going to introduce. We present Lemma 3.1 on this initialization.

Lemma 3.1 If Algorithm 3.1 is initiated with a control congruence $C_{k,init}$, the output cover C_k is a control congruence.

Proof Correctness of Algorithm 3.1 initiated by a singleton cover is proven in Su and Wonham (2004). The singleton cover is a special instance of a control congruence. The same proof of Su and Wonham (2004) applies here, when we generalize the algorithm to be initialized with any control congruence. \square

It is desirable to have small (in terms of number of states) local supervisors. Therefore, we want to compute control congruences which cannot be reduced further, i.e., any further merging of cells would result in an invalid control cover. We call such a

cover *maximally reduced*, see Definition 3.1. Reducedness of the control congruences obtained by Algorithm 3.1 is addressed in Lemma 3.2.

Definition 3.1 Cover C_k is *maximally reduced* w.r.t. $G, S, \Sigma_{c,k}$ iff $\forall \pi_1, \pi_2 \in C_k$, if $\pi_1 \neq \pi_2$, then $(C_k \setminus \{\pi_1, \pi_2\}) \cup \{\pi_1 \cup \pi_2\}$ is not a control congruence w.r.t. $G, S, \Sigma_{c,k}$.

Lemma 3.2 C_k obtained by Algorithm 3.1, is maximally reduced w.r.t. $G, S, \Sigma_{c,k}$.

Proof Algorithm 3.1 iterates over all pairs of states, and only skips pairs of states when mergeability between some pair of states contained in the respective cells has already been checked. Thus, if Algorithm 3.1 outputs a control congruence containing individual cells π_1 and π_2 , then mergeability has been checked between some pair of states $x_1 \in \pi_1, x_2 \in \pi_2$. Let us say x_1, x_2 respectively were in cells ρ_1, ρ_2 at the point their mergeability was checked. Since x_1 and x_2 were not merged, `check_merge` has returned false for this evaluation, which means that some pair of states x_3, x_4 respectively in ρ_1, ρ_2 were not mergeable. Since Algorithm 3.1 only merges cells (i.e., never splits a cell), we know that for the resulting control congruence $x_3 \in \rho_1 \subseteq \pi_1$ and $x_4 \in \rho_2 \subseteq \pi_2$. Since x_3 and x_4 are not mergeable, π_1 and π_2 cannot be merged to form a control congruence. \square

Note that Lemma 3.2 does not mean that the smallest control congruence is found by Algorithm 3.1. A control congruence (and resulting local supervisor) is generally non-unique, and which is found by Algorithm 3.1 depends on the order in which mergeability of the states is checked, which depends on their indexing. Unfortunately, finding a control congruence with the smallest number of cells is an NP-hard problem Su and Wonham (2004).

Lemmas 3.1 and 3.2 are applicable for supervisor localization (Cai and Wonham 2010) and supervisor reduction (Su and Wonham 2004) (which also uses Algorithms 3.1 and 3.2, i.e., not only applicable to the transformational approach we present next).

3.4 Problem definition

We assume a base system G consisting of n agents, a supervisor S , and a partition $\dot{\bigcup}_{k \in \{1, \dots, n\}} \Sigma_{c,k} = \Sigma_c \subseteq \Sigma$ of controllable events. This base system has been localized, i.e., a control congruence C_k was obtained for each agent k .

Now the system changes to variant system G' consisting of n' agents, a supervisor S' , and a partition of controllable events $\dot{\bigcup}_{k \in \{1, \dots, n'\}} \Sigma'_{c,k} = \Sigma'_c \subseteq \Sigma'$. We compute C'_k and LOC'_k for all k from 1 to n' based on the control congruences of the base system, rather than starting localization from scratch. We call this procedure *transformational supervisor localization* (TSL). TSL is to correctly localize the variant system, as defined in Problem 3.1. Note that in this problem definition, any adaptation can be made to the base system (that generates a well-defined variant system).

Problem 3.1 Use C_k for k from 1 to n of the base system G, S to transformationally compute new local supervisors LOC'_k for all k from 1 to n' that are control equivalent (Equations 3.3 and 3.4) to S' with respect to G' .

Since a set of local supervisors can be constructed from a set of control covers, in our work we mainly focus on finding control covers (in this case, control congruences) for the variant system in a transformational approach.

Furthermore, it is desirable to have small local supervisors. Therefore, TSL will compute maximally reduced control covers to use in the construction of the local supervisors.

3.5 Transformational supervisor localization

In this section, we first discuss an algorithm that is used to transform a cover C_k to a control congruence in case the system has been adapted. Next, we use this algorithm in the general procedure used for TSL.

3.5.1 Isolating conflicts

We consider the case that a control congruence C_k has been computed for some base system $S = (X, \Sigma, \xi, x_0, X_m)$, $G = (Q, \Sigma, \delta, q_0, Q_m)$. Now the system is adapted to form variant system $S' = (X', \Sigma', \xi', x'_0, X'_m)$, $G' = (Q', \Sigma', \delta', q'_0, Q'_m)$. In our notation, we use E', D'_k, \dots to indicate that the function E, D_k, \dots are applied to the variant automaton. I.e., E' is a function $E' : X' \rightarrow 2^{\Sigma'}$.

Algorithm 3.3 constructs a control congruence C'_k based on C_k . First, states that are removed from X to create X' are removed from the cells they were in in C_k . New states are added as singleton cells. Next, the algorithm looks for states x that do not satisfy condition (i) or (ii) of a control cover from Section 3.2 anymore with a state x' in the same cell. If such a state x is found, it is *isolated*: it is removed from its initial cell and placed in a singleton cell. Note that conditions (i) and (ii) are always satisfied for states in a singleton cell. Finally, all states that induce such a control consistency conflict are isolated, and the resulting cover is a control congruence.

We first present Example 3.2 to demonstrate the functioning of Algorithm 3.3. Next, we prove correctness of Algorithm 3.3 in Theorem 3.1.

Example 3.2

Let us consider the case the system of Example 3.1 is adapted. In addition to the disablements $D_1(x_0)=\{c\}$, $D_1(x_2)=\{a\}$ in the base system, the variant system has an additional disablement: $D_1(x_3)=\{a\}$. As a result, for the variant system $(x_0, x_3) \notin \mathcal{R}_1$. Therefore, the cover found in Example 3.1 is not valid anymore. This conflict is found

in line 6 of Algorithm 3.3, and subsequently x_0 (or x_3 depending on order of iteration) is removed from its previous cell and placed in a singleton cell. No more conflicts exist in the resulting cover. Constructing a local supervisor for this cover yields the automaton shown in Figure 3.3.

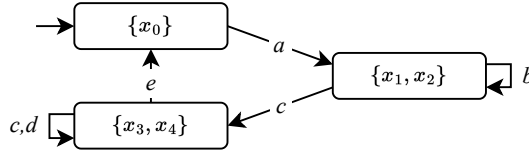


Figure 3.3: Isolated state x_0 .

Algorithm 3.3 isolate

Input: $C_k, S, G', S', \Sigma'_{c,k}$

Output: C'_k

- 1: $C'_k = \{\pi \setminus (X \setminus X') \mid \pi \in C_k\} \cup \cup \{\{x\} \mid x \in X' \setminus X\}$
 - 2: $flag = true$
 - 3: **while** $flag$ **do**
 - 4: $flag = false$
 - 5: **for all** $x \in X' \cap X$ **do**
 - 6: **if** $\exists x' \in [x]_{C'_k} : ((x, x') \notin \mathcal{R}'_k \vee (\exists \sigma \in E'(x) \cap E'(x')) : ([\xi'(x, \sigma)]_{C'_k} \neq [\xi'(x', \sigma)]_{C'_k}))$ **then**
 - 7: $flag = true$
 - 8: $C'_k = (C'_k \setminus \{[x]_{C'_k}\}) \cup \{[x]_{C'_k} \setminus \{x\}\} \cup \{\{x\}\}$
 - 9: **end**
 - 10: **end**
 - 11: **end**
 - 12: **return** C'_k
-

Theorem 3.1 Given $N = |X \cap X'|$, Algorithm 3.3 terminates, has a worst case time complexity of $O(|\Sigma| \cdot N^3)$, and the generated cover C'_k is a control congruence w.r.t. $G', S', \Sigma'_{c,k}$.

Proof A state in a singleton cell is trivially control consistent. If in the for-loop (lines 5-10) a state is found that is not control consistent with another state in the same cell, it is placed in a singleton cell and removed from its original cell, and the algorithm iterates over all states in $X \cap X'$ again. Eventually, since N is finite, there are no more

non-control consistent states, the for-loop terminates with $flag = false$, the while-loop breaks, and the algorithm terminates.

Checking the if-condition on line 6 has a worst case cost of $|\Sigma| \cdot N$. The for-loop (lines 5-10) is performed N times in worst case. The while-loop (lines 3-11) is performed N times in worst case. Therefore, the time complexity is $O(|\Sigma| \cdot N^3)$.²

The while-loop only breaks when conditions (i) and (ii) are both met for all states in $X \cap X'$. Also all states in $X' \setminus X$ are control consistent as they are placed in singleton cells. There is no overlap between cells in C'_k as all cells in $X' \setminus X$ are placed in singleton cells and no merges are performed for states in $X \cap X'$, which are initially partitioned by C_k . Thus, C'_k is a control congruence w.r.t. $G', S', \Sigma'_{c,k}$. \square

3.5.2 General procedure

In this section we present the TSL procedure, show in Theorem 3.2 that TSL solves Problem 3.1, and in Theorem 3.3 that the resulting control congruences are maximally reduced.

The TSL procedure is sketched in pseudo-code in Algorithm 3.4. We assume a mapping $\mathcal{M} : \{1, \dots, n'\} \rightarrow \{0, \dots, n\}$, that maps every agent in the variant system to either an agent of the base system, or to '0'. If $\mathcal{M}(k)=0$, it means no base control cover is selected and the initial control congruence is set to a singleton cover. In case $\mathcal{M}(k)$ is nonzero, control congruence $C_{\mathcal{M}(k)}$ is selected from the base system to perform `isolate` to find an initial control congruence. After performing `isolate`, the resulting cover might not be maximally reduced. This is why, after performing `isolate`, the cover is used to initialize `localize` in order to merge cells whenever possible. The reasoning for the TSL procedure is that `isolate` produces a control congruence in which generally states will already be merged into cells, limiting the work that needs to be done during `localize`. This is demonstrated in Example 3.3. TSL also returns covers $\{C'_k | 1 \leq k \leq n'\}$ so that they can be used in a next TSL if the system is further adapted.

Example 3.3

This is a continuation of Example 3.2, in which a variant system was presented to the base system of Example 3.1, and `isolate` was performed to compute a control congruence for the variant system, yielding the local supervisor of Figure 3.3. However, the cover can be further reduced, resulting in a local supervisor with fewer states. We perform `localize` initialized with the cover found in Example 3.2. $\{x_0\}$ cannot merge with $\{x_1, x_2\}$ for multiple reasons: x_1 and x_2 both enable event c , which is disabled in x_0 , and x_0 enables event a , which is disabled in x_2 . $\{x_0\}$ cannot merge with $\{x_3, x_4\}$

² To achieve this cost in implementation, instead of storing cells as state sets, a cell index number is stored for each state. A state can be isolated by simply assigning it with a new cell index. Since all cells are non-overlapping, comparing whether two cells are the same can be done by comparing the cell index of one state from each cell.

as x_0 enables a , which is disabled in x_3 in the variant system. $\{x_1, x_2\}$ can be merged with $\{x_3, x_4\}$: there are no conflicts. After merging these cells, no further merges are possible, leading to control congruence $\{\{x_0\}, \{x_1, x_2, x_3, x_4\}\}$. Constructing a local supervisor for this cover yields the automaton in Figure 3.4.

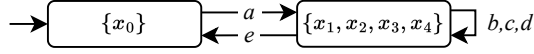


Figure 3.4: Local supervisor of variant system.

Algorithm 3.4 TSL

Input: $\{C_k | 1 \leq k \leq n\}$, S , G' , S' , $\{\Sigma'_{c,k} | 1 \leq k \leq n'\}$, \mathcal{M}

Output: $\{LOC'_k | 1 \leq k \leq n'\}$, $\{C'_k | 1 \leq k \leq n'\}$

```

1: for  $k = 1$  to  $n'$  do
2:   if  $\mathcal{M}(k) \neq 0$  then
3:      $C'_{k,init} = \text{isolate}(C_{\mathcal{M}(k)}, S, G', S', \Sigma'_{c,k})$ 
4:   else
5:      $C'_{k,init} = \{\{x\} | x \in X'\}$ 
6:   end
7:    $C'_k = \text{localize}(G', S', \Sigma'_{c,k}, C'_{k,init})$ 
8:   Compute  $LOC'_k$  based on  $C'_k$ 
9: end
10: return  $\{LOC'_k | 1 \leq k \leq n'\}$ ,  $\{C'_k | 1 \leq k \leq n'\}$ 

```

Theorem 3.2 Algorithm 3.4 terminates, has worst case complexity $O(n' \cdot |\Sigma'| \cdot |X'|^4)$, and solves Problem 3.1.

Proof Algorithm 3.4 terminates because `isolate` (Theorem 3.1) and `localize` (Su and Wonham 2004; Cai and Wonham 2010) terminate.

In worst case, `isolate` is called n' times, and its complexity is $O(|\Sigma'| \cdot |X \cap X'|^3)$ (Theorem 3.1). `localize` is called n' times, and its complexity is $O(|\Sigma'| \cdot |X'|^4)$ (Su and Wonham 2004; Cai and Wonham 2010). Therefore, the complexity of TSL is $O(n' \cdot |\Sigma'| \cdot |X'|^4)$.

For each agent in the variant system, `localize` is initiated with a control congruence, since line 3 constructs a control congruence (Theorem 3.1) and the singleton cover constructed in line 5 is a control congruence. Thus, the covers computed by `localize` are control congruences following from Lemma 3.1. It is shown in Cai and Wonham (2010) that local supervisors constructed from control congruences satisfy Problem 3.1. \square

Clearly, SL and TSL have the same complexity. The idea is that TSL is quicker in practice, when the variant system is sufficiently similar to the base system. Unfortunately, at the moment we cannot predict whether TSL will be quicker than SL. We present some experiments in Section 3.6 to study the computational benefit in practice.

In addition to correctness of the result, TSL also produces maximally reduced control congruences.

Theorem 3.3 All $C'_k \in \{C'_k | 1 \leq k \leq n'\}$ obtained by Algorithm 3.4 are maximally reduced w.r.t. $G', S', \Sigma'_{c,k}$.

Proof Every control congruence C'_k that is returned by Algorithm 3.4 is constructed by performing Algorithm 3.1. Control congruences constructed by Algorithm 3.1 are maximally reduced (Lemma 3.2). Thus, the theorem holds. \square

3.6 Case study: Cat and Mouse Tower

As a case study to evaluate the efficiency of TSL relative to SL, we take the Cat and Mouse Tower (CMT) model. This model is an extension of the cat and mouse example in Ramadge and Wonham (1989) by adding multiple floors. The CMT model was first published online as a part of a special session on benchmarking of software tools held during the 9th Workshop on Discrete Event Systems in 2008³. In literature, the CMT model first appears in Ma and Wonham (2008), Miremadi et al. (2008), and Moor et al. (2008), each in the proceedings of the mentioned conference.

In the CMT model, there are n floors, and on each floor of the tower there are five rooms as shown in Figure 3.5. Cats and mice can move between the rooms as indicated by the arrows. Between each floor there is a connection for both cats and mice. This connection is between room j of floor $5 \cdot i + j$ to room j of floor $5 \cdot i + j + 1$, for $i \in \mathbb{N}_0$, $j \in \{1, 2, 3, 4, 5\}$, and $5 \cdot i + j < n$. So room 1 floor 1 is connected to room 1 floor 2; room 2 floor 2 is connected to room 2 floor 3; and so forth, essentially forming a spiraling staircase. All doors can be controlled, except for the bidirectional cat door between rooms 2 and 4. There are k cats and k mice, and consequently each room can also hold between 0 and k cats and/or mice. The cats start in room 1 of floor 1, and the mice start in room 5 of floor n . The requirement of this system is that there can never be a cat and a mouse in the same room at the same time.

As base system, we take a tower with four floors, one cat, and one mouse. The monolithic supervisor of this system has 362 states and 1159 transitions. For localization, we consider each floor as a separate agent. An agent controls all events of the cat and mouse that originate in that floor, e.g., the floor 1 agent controls all doors on that floor, and the movements from floor 1 room 1 to floor 2 room 1 (but not the other way around; these are controlled by the floor 2 agent).

We construct five variant systems (each modifies the base system directly, i.e., the adaptations are not cumulative):

³ <http://www.alessandro-giua.it/WODES/WODES08/pages/benchmark.php>

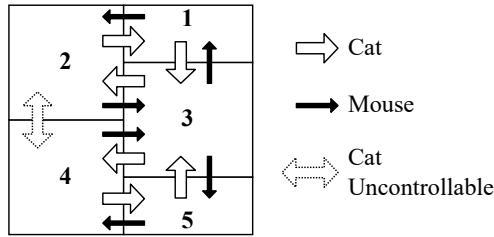


Figure 3.5: CMT room layout of a floor (Ramadge and Wonham 1989).

1. Removed cat door from room 3 to room 4 on floor 2.
2. Made all doors controllable.
3. Added requirement that cats should never reach floor 4.
4. Removed room 5 of floor 1.
5. Added a room 6 to floor 1 with bidirectional controllable doors for cat and mouse to room 5 of floor 1.

The models and a proof-of-concept implementation of the algorithms have been made in Matlab⁴. We performed SL for the base system, and SL and TSL for each variant system. For TSL, each agent (floor) of the variant system is mapped to the same floor in the base system. A standard personal computer with i7 processor was used. Matlab used less than 2 GB of memory. Since we draw conclusions on relative and not absolute runtimes, the conclusions are not influenced by the hardware. Because the results are influenced by state indexing order, the experiments are performed for ten random index orders and mean values over those runs are presented.

In the left side of Table 3.1 we compare the computation time in seconds of performing SL and TSL for the agents in the variant system. To provide further detail, we show how much time of performing TSL is spent on the `isolate` and `localize` portion of the procedure. The percentage change comparing TSL to SL is displayed, where a negative or positive value respectively indicates how much quicker or slower TSL is compared to SL.

In the right side of Table 3.1 we compare the number of cells between the result of SL and TSL for the agents in the variant system. The numbers under ‘initial guess’ indicate the number of cells of C'_k after line 1 of `isolate`, before any states are isolated. The numbers under ‘isolated’ indicate the number of cells after completing `isolate`, but before `localize` is performed.

For the first variant system, we observe that no states need to be isolated during `isolate` and no further merges of cells can be performed when performing `localize` initialized by the control cover of the base system. Compared to performing SL initialized by a singleton cover, TSL is much quicker. For the second variant system, there is much less computational benefit. Here, a local system is found where

⁴ All used models and algorithms can be found here: <https://github.com/sbthuijsman/TSL>.

Table 3.1: CMT experimental results, 4 floors, mean of ten measurements for each computation.

variant system	agent	mean runtime					mean # cells			
		SL [s]	isolate [s]	initialized localize [s]	TSL [s] (sum)	% change	SL	initial guess	isolated	TSL
1 (362 states, 1142 trans.)	1	1.22	0.05	0.15	0.20	-83%	11.6	11.6	11.6	11.6
	2	1.11	0.04	0.21	0.25	-78%	14.3	14.3	14.3	14.3
	3	1.16	0.04	0.15	0.18	-84%	13.7	13.7	13.7	13.7
	4	2.47	0.06	0.08	0.14	-94%	10.9	10.9	10.9	10.9
2 (375 states, 1214 trans.)	1	1.84	0.03	1.73	1.76	-4%	11.5	23.9	295.6	18.8
	2	1.10	0.05	1.26	1.30	+18%	15.0	27.8	231.6	22.1
	3	1.32	0.04	1.21	1.26	-5%	14.7	27.5	232.7	21.7
	4	3.78	0.03	1.92	1.95	-48%	11.1	24.3	292.7	17.2
3 (270 states, 853 trans.)	1	0.78	0.04	0.10	0.14	-82%	9.4	8.8	8.8	8.8
	2	0.71	0.02	0.09	0.11	-85%	12.8	14.1	14.1	13.9
	3	0.88	0.03	0.10	0.12	-86%	10.8	14.3	14.7	13.9
	4	6.90	0.04	3.68	3.72	-46%	2.5	10.0	10.0	9.0
4 (309 states, 986 trans.)	1	2.19	0.05	0.65	0.69	-68%	8.9	11.2	11.2	10.6
	2	0.82	0.03	0.10	0.12	-85%	12.8	14.3	14.3	14.3
	3	0.79	0.03	0.15	0.17	-78%	13.9	14.1	14.1	14.1
	4	1.78	0.04	0.08	0.13	-93%	10.6	11.4	11.4	11.4
5 (403 states, 1304 trans.)	1	2.23	0.05	1.49	1.54	-31%	12.1	51.5	327.4	14.7
	2	1.41	0.05	0.66	0.71	-50%	14.6	55.2	269.4	15.9
	3	1.79	0.05	0.61	0.66	-63%	14.3	55.1	262.0	14.4
	4	4.59	0.03	1.49	1.52	-67%	11.2	52.4	323.2	11.7

TSL is slower than SL, i.e., in this case localization is quicker when initialized by a singleton cover. At the moment, we have no way to predict when this will be the case. We observe that for this system a lot of states need to be isolated for all subsystems. Even so, isolation is performed relatively quickly. Because the isolated cover is relatively close to the singleton cover (which has 375 cells), TSL runtimes are relatively close to the SL runtimes. Another observation is that TSL computes covers with more cells than SL, because it starts with a coarser cover which limits the cell merges that can be made during *localize*. For variant systems 3, 4, and 5 TSL is consistently quicker than SL, even though for variant system 5 a lot of states require to be isolated.

The same experiments have been performed for larger instances of CMT, with 6 floors (842 states) and 8 floors (1525 states). The results are shown in Tables 3.2 and 3.3. Since the runtimes of these larger systems is considerably longer, each computation is only performed once, and the runtime of that computation is shown. The same conclusions can be made for these larger instances. Respectively, the average percentage change over all local systems for CMT with 4, 6, and 8 floors, were -61% , -54% , and -57% .

From a monolithic point of view the adaptations made to the CMT system are considerable (reflected in the change in number of states and transitions). Regardless, these experiments suggest that TSL is more efficient than performing SL from scratch.

3.7 Conclusions

We presented a TSL procedure, that reuses control congruences from a previous SL to more efficiently compute these control congruences for a system once it is adapted. Correctness of the algorithms is shown, and examples are provided. The method is evaluated by means of some experiments on the CMT system. For the experiments on this parameterized model, the runtime of TSL is shown to be generally lower than SL. For future work it would be interesting to evaluate the efficiency of TSL using more models.

From a high level TSL uses the same setup as Transformational Supervisor Synthesis (TSS), which is discussed in Chapter 2. They use the output of a computation on a base system to compute output for some variant system. Regardless, the specific functioning of TSL is very different than that of TSS, and the same goes for their nontransformational counterparts SL and SS. Therefore, the efficiency of TSL relative to SL cannot be related to the efficiency of TSS relative to SS.

Table 3.2: CMT experimental results, 6 floors, one measurement for each computation.

variant system	agent	runtime					# cells			
		SL [s]	isolate [s]	initialized localize [s]	TSL [s] (sum)	% change	SL	initial guess	isolated	TSL
1 (842 states, 2975 trans.)	1	54.1	0.4	13.3	13.7	-75%	12	12	12	12
	2	16.0	0.2	1.6	1.8	-89%	16	16	16	16
	3	37.1	0.3	3.9	4.1	-89%	13	13	13	13
	4	32.1	0.3	1.2	1.5	-95%	12	12	12	12
	5	72.5	0.4	0.5	0.9	-99%	12	12	12	12
	6	57.1	0.5	0.6	1.1	-98%	7	7	7	7
2 (864 states, 3004 trans.)	1	20.9	0.1	22.8	23.0	10%	15	32	791	22
	2	53.1	0.1	23.1	23.2	-56%	14	35	753	26
	3	58.5	0.1	21.4	21.5	-63%	12	33	756	19
	4	37.3	0.1	25.1	25.2	-32%	12	37	706	29
	5	18.5	0.1	22.3	22.4	21%	18	35	748	25
	6	49.6	0.1	25.2	25.3	-49%	15	36	729	29
3 (420 states, 1502 trans.)	1	5.3	0.1	1.8	1.9	-64%	7	7	7	7
	2	4.9	0.1	0.5	0.6	-89%	16	13	13	13
	3	6.0	0.1	0.6	0.7	-89%	12	14	15	14
	4	84.1	0.1	4.4	4.5	-95%	2	14	14	12
	5	112.0	0.1	67.0	67.2	-40%	1	15	15	1
	6	111.8	0.1	82.2	82.3	-26%	1	12	12	1
4 (759 states, 2707 trans.)	1	17.9	0.2	72.0	72.2	303%	14	12	12	12
	2	13.7	0.1	2.1	2.3	-84%	21	18	18	18
	3	14.8	0.1	1.0	1.1	-93%	21	21	21	21
	4	44.9	0.3	0.4	0.6	-99%	12	13	13	13
	5	27.8	0.3	0.8	1.1	-96%	13	13	13	13
	6	21.8	0.2	6.3	6.4	-70%	13	14	14	14
5 (903 states, 3230 trans.)	1	59.4	0.2	39.1	39.3	-34%	13	72	798	15
	2	19.4	0.1	14.1	14.3	-26%	20	80	632	19
	3	24.4	0.2	6.5	6.6	-73%	17	78	670	17
	4	40.5	0.2	6.4	6.6	-84%	18	78	694	21
	5	19.4	0.2	2.2	2.3	-88%	22	84	543	23
	6	378.1	0.1	99.0	99.1	-74%	10	72	828	11

Table 3.3: CMT experimental results, 8 floors, one measurement for each computation.

variant system	agent	runtime					# cells			
		SL [s]	isolate [s]	initialized localize [s]	TSL [s] (sum)	% change	SL	initial guess	isolated	TSL
1 (1525 states, 5407 trans.)	1	2978.8	2.9	7.3	10.2	-100%	9	9	9	9
	2	1445.2	1.9	3.7	5.6	-100%	11	11	11	11
	3	172.0	0.8	5.5	6.3	-96%	14	14	14	14
	4	550.3	0.4	9.6	10.0	-98%	23	23	23	23
	5	305.0	0.5	7.4	7.9	-97%	20	20	20	20
	6	654.6	1.5	5.7	7.2	-99%	11	11	11	11
	7	97.2	0.4	4.2	4.6	-95%	23	23	23	23
	8	294.9	0.5	27.2	27.7	-91%	18	18	18	18
2 (1547 states, 5446 trans.)	1	2822.4	0.2	426.5	426.7	-85%	11	34	1452	26
	2	389.0	0.2	144.3	144.6	-63%	18	38	1372	29
	3	277.9	0.3	185.7	186.0	-33%	21	47	1124	40
	4	647.4	0.2	247.5	247.7	-62%	15	37	1367	34
	5	2517.3	0.3	216.9	217.2	-91%	17	46	1130	39
	6	2066.4	0.2	421.8	422.0	-80%	13	34	1448	18
	7	325.2	0.2	1217.8	1218.0	275%	18	41	1296	27
	8	2855.3	0.2	2233.9	2234.1	-22%	10	32	1472	10
3 (570 states, 2032 trans.)	1	37.4	0.3	17.4	17.8	-52%	8	8	8	8
	2	25.1	0.1	2.2	2.2	-91%	11	20	20	20
	3	29.1	0.1	2.6	2.7	-91%	12	21	21	21
	4	440.1	0.2	521.1	521.2	18%	2	13	13	12
	5	729.9	0.2	422.9	423.1	-42%	1	13	13	1
	6	728.4	0.3	570.4	570.8	-22%	1	8	8	1
	7	729.4	0.2	433.9	434.1	-40%	1	16	16	1
	8	729.6	0.2	436.6	436.8	-40%	1	14	14	1
4 (1412 states, 5043 trans.)	1	3166.3	1.4	51.9	53.2	-98%	5	10	10	10
	2	101.7	0.4	11.5	11.9	-88%	23	21	21	21
	3	97.8	0.3	18.9	19.1	-80%	26	26	26	26
	4	162.4	1.3	4.0	5.3	-97%	22	13	13	13
	5	186.4	0.7	195.0	195.7	5%	25	27	914	40
	6	118.7	0.3	59.6	59.9	-50%	25	25	25	25
	7	2294.9	1.7	316.7	318.4	-86%	13	12	12	12
	8	156.2	0.3	4.3	4.6	-97%	20	23	23	23
5 (1606 states, 5748 trans.)	1	3795.8	0.6	3432.6	3433.1	-10%	10	89	1553	11
	2	132.1	0.4	25.1	25.5	-81%	28	108	1128	27
	3	279.2	0.3	137.1	137.5	-51%	17	97	1408	16
	4	145.7	0.5	29.6	30.1	-79%	29	109	1000	28
	5	196.4	0.3	30.3	30.6	-84%	18	99	1327	18
	6	1431.3	0.2	1277.1	1277.4	-11%	12	93	1484	12
	7	245.0	0.4	71.6	71.9	-71%	27	108	1055	27
	8	4696.0	0.4	4232.9	4233.3	-10%	9	90	1542	9

Chapter 4

Supervisory control for dynamic feature configuration in product lines

Abstract In this chapter a framework for engineering supervisory controllers for product lines with dynamic feature configuration is proposed. The variability in valid configurations is described by a feature model. Behavior of system components is achieved using (extended) finite automata and both behavioral and dynamic configuration constraints are expressed by means of requirements as is common in supervisory control theory. Supervisory controller synthesis is applied to compute a behavioral model in which the requirements are adhered to. For the challenges that arise in this setting, multiple solutions are discussed. The solutions are exemplified in the CIF toolset using a model of a coffee machine. A use case of the much larger Body Comfort System product line is performed to showcase feasibility for industrial-sized systems.

4.1 Introduction

In present day development of systems and products, reuse of both software and hardware components is sought to reduce development and production costs, and shorten time-to-market. The goal of software/system product line engineering is to facilitate reuse throughout all phases of systems engineering (Pohl et al. 2005). Adoption of this paradigm requires identification of the core assets of the products in the domain in order to exploit their commonality and manage their variability, often defined in terms of features. A feature is defined as a logical unit of behavior specified by a set of functional and non-functional requirements (Bosch 2000) or a distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder (Czarnecki and Eisenecker 2000). Feature models may be used to define which combinations of features are considered valid product configurations (Benavides et al. 2010).

This chapter is strongly based on: Thuijsman, Sander B.; Reniers, Michel A.: Supervisory control for dynamic feature configuration in product lines. In: *Transactions on Embedded Computing Systems* (2023). ACM Press. – URL <https://doi.org/10.1145/3579644>. – In press.

In literature there has been much attention for correct configuration of product lines (Benavides et al. 2010). Behavioral correctness is studied only recently, since Classen et al. (2010). Typically the approaches that are used for guaranteeing a proper functioning product lines (i.e., correct with respect to its requirements or specifications) are verification technologies such as theorem provers (Classen et al. 2013), model checkers (Baier and Katoen 2008), and correct-by-construction approaches such as supervisory controller synthesis (ter Beek et al. 2016). In ter Beek et al. (2016), for the first time supervisory controller synthesis (Ramadge and Wonham 1987) has been considered for constructing supervisory controllers for an product line described by a feature model.

Supervisory control theory, as introduced by Ramadge and Wonham (1987), is a model-based approach to control discrete-event systems. In this framework a model is created of the uncontrolled system, and behavioral requirements are specified that define what behavior is allowed. Using these models, a supervisory controller can be computed algorithmically (synthesized), such that it restricts the behavior of the system to always be in accordance with the requirements. Depending on the synthesis algorithm, the behavior of the system under control is guaranteed to have some useful properties, such as safety, nonblockingness, controllability, and maximal permissiveness. The benefits of supervisory control theory have been demonstrated in industrial use cases, such as for example supervisory control of lithography machines in van der Sanden et al. (2015), health-care systems in Theunissen et al. (2014), automotive applications in Korssen et al. (2018), and infrastructural systems in Reijnen et al. (2020). Typically, the models that are input to supervisory controller synthesis are discrete-event system models such as (extended) finite automata (Cassandras and Lafortune 2021; Sköldstam et al. 2007).

The contribution of this work is a model-based framework for the supervisory control of product lines. This approach consists out of the following steps:

1. *Representing the feature model in extended finite automata.* This is prerequisite, because we apply supervisory controller synthesis that is based on automata specifications.
2. *Capturing dynamic configuration of features in the models.* In this work, we pay additional attention to the situation where features might enter or leave the system during runtime.
3. *Modeling uncontrolled system behavior such that it properly takes the current configuration into account.* A component-wise specification of the system behavior is given, where the component behavior is linked to the presence of features in the configuration.
4. *Modeling behavioral requirements depending on presence of features.* The requirements of the behavior are dependent on the current configuration. Additionally, different requirements may apply when the system is in a transitional phase in between valid configurations.
5. *Applying supervisory controller synthesis.* A correct-by-construction supervisory controller is obtained from the developed models.

For most of these steps, there are multiple solutions and it depends on the case at hand which one is most appropriate. We mention the alternatives and illustrate them. To exemplify the method discussed in this chapter, we use the coffee machine system from ter Beek and de Vink (2014) as a running example. Modeling of automata and supervisory controller synthesis is performed using the tool CIF (van Beek et al. 2014; Fokkink et al. 2023). Scalability and applicability of the approach is later demonstrated using the Body Comfort System (BCS) from Lity et al. (2013).

4.1.1 Related work

This work is based on, and can be seen as an extension to, ter Beek et al. (2016) and Thuijsman and Reniers (2020). The basis of the approach we present here was first introduced in ter Beek et al. (2016), where feature models are modeled in CIF, behavioral models of the system components are defined, and a supervisory controller is obtained that takes into account the possible configurations as defined by the feature model. In Thuijsman and Reniers (2020), this work was extended by also considering the setting of dynamic configuration, where components are allowed to enter and leave the system. Relative to ter Beek et al. (2016) and Thuijsman and Reniers (2020), the extension we present here includes more modeling possibilities, considerations, explanations, and examples. Additionally, this work also provides a case study of the large BCS use case, showcasing applicability for industrial-sized product line systems.

Below we mention some more related work, which we divide into the following categories: (1) works that study dynamic reconfiguration during run-time, but do not apply supervisory control theory, (2) works that apply supervisory control theory for multiple configurations during design-time, (3) works that apply supervisory control for dynamic reconfiguration during run-time. Our work fits the latest category. However, our work differentiates from the mentioned works, as in none of them dynamic feature configuration in relation to supervisory control engineering with a clear separation of uncontrolled system behavior and specification of behavioral and dynamic reconfiguration requirements is discussed.

4.1.1.1 Dynamic configuration during run-time

In Kogekar et al. (2004) an approach for dynamic software reconfiguration in sensor networks is presented. The dynamic reconfiguration is based on formal constraints in terms of quality-of-service parameters that are measured at runtime.

Dynamic runtime variability of software product lines in embedded automotive software systems is applied to create adaptable and reconfigurable software architectures in Shokry and Babar (2008). Also Rosenmüller et al. (2011) discusses reconfiguration with the purpose of determining an optimal configuration at runtime. In both papers the dynamic configuration is under control, which is typically not the case in the present chapter.

In Shen et al. (2011), a feature-oriented method is proposed to support runtime variability reconfiguration by introducing an intermediate level between feature variations and implementations.

Gharsellaoui et al. (2021) deal with reconfiguration of real-time embedded systems to cope with hardware/software faults.

Sharifloo et al. (2016) argue that it is not reasonable to anticipate all relevant context changes during design-time and therefore propose a model that combines learning of adaptation rules with evolution of the configuration space, which can be applied during run-time.

4.1.1.2 Supervisory control and design-time system configuration

In Chapter 2, the assumption is made of no a-priori knowledge of the possible system configurations. Computation of a supervisory controller for an updated system, given knowledge over the base system, is studied.

In Basile (2019), priced featured automata were translated to extended finite automata and the structure of the product line was used to greatly reduce the number of supervisory controller syntheses required to solve game-based energy problems.

In Kahraman and Cleophas (2021) feature models are used to generate product instances for the model-based engineering tool LSAT. LSAT is a tool used to design supervisory controllers, but it cannot perform supervisory controller synthesis (van der Sanden et al. 2021).

In Verbakel et al. (2021) a method to obtain supervisory controllers for a product family is discussed through the use of a configurator, where synthesis is applied after selection of parameterized components.

4.1.1.3 Supervisory control and dynamic configuration during run-time

Basile et al. (2020) apply supervisor synthesis to featured modal contract automata. They synthesize orchestrations, that match service requests to service offers, for all valid products in a product line, by joining the orchestrations of a small subset of the valid products. By means of a composition operation, the product line can dynamically be updated and new services can join composite services.

4.1.2 Structure

In Section 4.2 we introduce feature models and the CIF language. Using the CIF language, we show how feature models can be represented in automata format in Section 4.3. In Section 4.4, the use of automata to represent dynamically configured feature models is discussed. Modeling of component behavior in the setting of dynamic configuration is discussed in Section 4.5. In Section 4.6, the modeling of behavioral



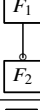
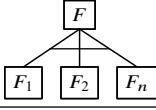
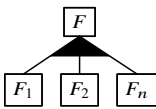
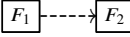
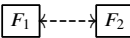
requirements that are dependent on the current configuration is discussed. Supervisory controller synthesis is applied in Section 4.7. An industrial-sized use case is discussed in Section 4.8. Section 4.9 concludes the chapter.

4.2 Preliminaries

4.2.1 Feature Models

A feature model (Heymans et al. 2008) is a graph with a collection of nodes representing features, and a number of relations between these features, called feature constraints. The feature constraints that can be expressed are summarized in Table 4.1, which is taken from ter Beek et al. (2016). The right column provides logical formulas that express how presence of the features (denoted by the F_i) is restricted by the different types of constraints.

Table 4.1: Different feature constraints of a feature model (ter Beek et al. 2016).

Constraint		Formula
root		$F_0 \iff true$
mandatory		$F_1 \iff F_2$
optional		$F_2 \implies F_1$
alternative		$(F_1 \iff (\neg F_2 \wedge \dots \wedge \neg F_n \wedge F))$ $\wedge \dots \wedge$ $(F_n \iff (\neg F_1 \wedge \dots \wedge \neg F_{n-1} \wedge F))$
or		$F \iff (F_1 \vee F_2 \vee \dots \vee F_n)$
requires		$F_1 \implies F_2$
excludes		$\neg (F_1 \wedge F_2)$

For any valid configuration a *root* feature needs to be present. *Mandatory* features are required to be present when their parents are, and optional features may be present when their parents are. For a set of *alternative* features, exactly one is present when their parent is present. And for a set of *or* features, at least one is present when their parent is present. It can also be defined that the presence of a certain feature *requires* or *excludes* another feature to be present.

In an *extended* feature model (Benavides et al. 2005), attributes can be assigned to features. An example of such an attribute could be the weight or price of a feature. Attributes are typically defined by a name, domain (such as integers, enumerations, etc.), and a value. Attribute constraints can be expressed using attributes of features. A constraint could be a maximal value for the total weight or price of the system. We may use the term *feature model* to refer to both ‘normal’ and extended feature models when the type of feature model is not relevant or clear from the context.

Example 4.1 Feature model for a coffee machine

We consider the product line for a coffee machine from ter Beek and de Vink (2014). An extended feature model that captures the allowed configurations for this product line is presented in Figure 4.1. In the solid boxes, the features’ names are shown with an abbreviation. We observe that the coffee machine always contains a sweet, coin, and beverage feature. Optionally, the machine may also be able to sound a ringtone or return change. The machine accepts euro or alternatively dollar coins. The machine always offers coffee as a beverage, but may optionally also offer cappuccino or tea. If the machine offers cappuccino, it cannot accept dollar coins. When the machine offers cappuccino, it is required the machine comes equipped with the ringtone feature.

Figure 4.1 also shows dashed boxes, in which the feature attributes are shown with a name, domain, and value. Some features have no attribute, and some features have a cost attribute. The cost is valued by an integer, and the cost values are between 3 and

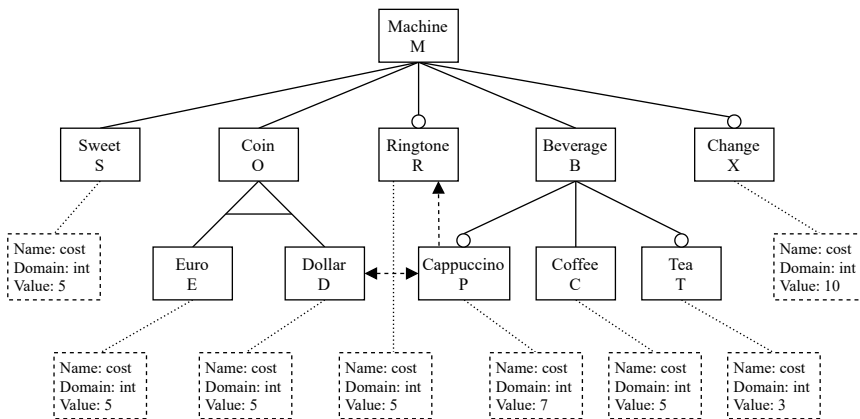


Figure 4.1: Feature model of the coffee machine (ter Beek and de Vink 2014).

10. The total machine cost is the sum of costs of all features that have a cost attribute. Using the attributes, one can formulate an attribute constraint, e.g., the total machine cost must be less than or equal to 30.

The feature model represents 20 valid configurations. If we include the previously mentioned cost constraint, there are 16 valid configurations.

4.2.2 CIF

There are two tool suites that support supervisory controller synthesis for models expressed as extended finite automata: Supremica (Åkesson et al. 2006) and CIF (van Beek et al. 2014; Fokkink et al. 2023). In ter Beek et al. (2016), it has been shown how the CIF language and tool set can be used for synthesizing a supervisory controller that is suited for an product line. The approach uses the concept of algebraic variables extensively, which is not available in Supremica.

CIF, part of the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™)¹, is a language and tool set that supports model-based engineering of supervisory controllers involving modeling, (visualized) simulation, synthesis, verification, and code generation (Fokkink et al. 2023). In the past years CIF has been applied to many industrial-size case studies (van der Sanden et al. 2015; Theunissen et al. 2014; Korssen et al. 2018; Reijnen et al. 2020). Although CIF allows modeling of real-valued variables that evolve continuously over time (as described by differential equations), for the purpose of this work our attention is restricted to discrete-event models.

Discrete-event models of the uncontrolled system (also called plant) can be developed in the form of a collection of *extended finite automata* (Sköldstam et al. 2007). The automata that comprise the plant synchronize over shared events and interact through the reading of each others' (discrete and finite) variables, and thus the global system behavior is achieved taking the synchronous product (Cassandras and Lafortune 2021). An automaton consists of locations and edges between these locations. The edges are labeled by an event, a guard, and an update. The guard describes a condition (in terms of the variables) that enables the occurrence of the event associated with the edge. The update describes how the values of the variables change in such a transition. In CIF variables are declared inside an automaton and follow the 'global read, local write' principle, which means that each variable may be inspected in any of the automata, but may only be adapted in its defining automaton. A CIF automaton has at least one initial location, and variables have at least one initial value. The *state* of an automaton is defined by its current location and variable valuations. Initial/marked locations implicitly define initial and marked states.

Events are defined to be *controllable* or *uncontrollable*. Uncontrollable events cannot be prevented from occurring by a supervisory controller, whereas controllable

¹ The ESCET toolset and documentation is open source and freely available at <https://eclipse.org/escet/>. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

Listing 4.1: Textual CIF model of an automaton.

```

1 plant automaton ExampleAutomaton:
2 controllable start, process;
3 uncontrollable finish;
4 disc int[0..5] c = 0;
5 location Idle: initial; marked;
6 edge start goto Busy;
7 location Busy:
8 edge process when c<5 do c:=c+1;
9 edge finish when c>4 do c:=0 goto Idle;
10 end

```

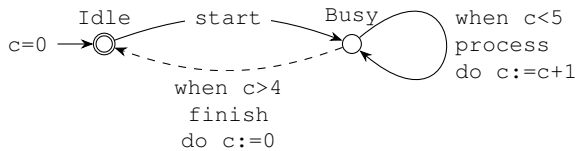


Figure 4.2: Graphical representation of the automaton from Listing 4.1.

events can be blocked. The extended finite automata may have *marked* states. Marked states are typically used to denote states in which the system has finished a task. By applying supervisor controller synthesis, the controllable events are restricted in such a way that from every reachable state, a marked state can eventually be reached.

An example of a CIF automaton is given in Listing 4.1. Its graphical representation is given in Figure 4.2. Locations are represented by small circles with their name next to them. Initial locations have a dangling incoming arrow and possibly an expression stating the initial values of the variables. In the example there is a c variable that is defined as a discrete (the value will update in a discrete manner) integer that can take values between 0 and 5, denoted with keywords `disc int[0..5]`, which has an initial value of 0. Marked locations have a double circle representation. Edges that are labeled by a controllable event are represented by a solid arrow and edges with an uncontrollable event by a dashed arrow. The optional guard is indicated by the keyword `when` and the update using `do`. In this chapter we use both textual and graphical representations as we see fit.

In CIF requirements are specified by means of automata that state in which orderings the contained events are allowed to occur, or by using state-based expressions such as event conditions and state invariants (Ma and Wonham 2006; Markovski et al. 2010). An event condition restricts the occurrences of an event to situations where a certain condition in terms of the variables of the model is satisfied. A state invariant expresses in which states the system is allowed to be.

CIF has several concepts that facilitate modeling of large systems, such as a definition/instantiation mechanism for automata and requirements and algebraic variables. For an algebraic variable, the value is defined to be identical to the value of some

expression (in terms of other variables). In this chapter algebraic variables are used abundantly.

With each location L in each automaton A , CIF associates a location variable $A.L$ that may be used in guards, in right-hand sides of updates and algebraic variables, and in state-based requirements. Updates of these variables are implicit and according to the location change of an automaton.

4.3 Static feature models in CIF

In this section we demonstrate modeling of feature models where features cannot configure dynamically, i.e., they are static, as proposed in ter Beek et al. (2016). We first discuss normal (non-extended) feature models, and then attributes to allow extended feature models. Next, in Section 4.4 we will show how these static models can be extended to feature models supporting dynamic feature configuration, i.e., features may enter or leave the system during run-time.

A CIF model representing all allowed configurations for a given feature model is obtained as follows. For each feature, an automaton is introduced that captures whether the feature is present or not. It uses a Boolean variable `present`, the value of which is fixed initially, that is `true` when the feature is present and `false` otherwise. This variable is also used to capture the feature constraints expressed in the feature model.

Since the feature automata have the same structure we use an automaton definition in CIF, which is then instantiated for each feature in the feature model. The CIF specification for this automaton definition is given in the first four lines of Listing 4.2. In CIF, every automaton needs to have at least one location, hence the dummy location (without a name) defined in Listing 4.2. Note that the initial value of `present` is left implicit (is allowed to be either `true` or `false` by using the keywords `in any`). For each feature in the feature model an instance of this feature automaton definition is obtained by a statement such as the ones in lines 6 and 7. These instances act just like separately defined automata. We can write `F1.present` to refer to the `present` variable in `F1`.

Listing 4.2: Automaton definition for features and instantiation for features.

```

1 plant def FEATURE():
2   disc bool present in any;
3   location: initial; marked;
4 end
5
6 F1: FEATURE();
7 F2: FEATURE();
8 ...

```

Feature constraints arising from a feature model can be modeled in CIF in such a way that the transformation from a feature model to a CIF model can easily be automated. For each of the constraint types in Table 4.1, an algebraic CIF expression

is shown in Listing 4.3. ‘//’ in the listing denotes that the remainder of the line is a comment, which we use to specify the constraint type. Also more complex constraints between features can be formulated in this way. These algebraic expressions can also be used to denote more complex constraints between features that are not considered here.

Listing 4.3: Several feature constraint expressions.

```

1 alg bool r1 = F0.present <=> true; //root
2 alg bool r2 = F1.present <=> F2.present; //mandatory
3 alg bool r3 = F2.present => F1.present; //optional
4 alg bool r4 = (F1.present <=> (not(F2.present) and F.present))
  and (F2.present <=> (not(F1.present) and F.present));
  //alternative
5 alg bool r5 = F.present <=> (F1.present or F2.present); //or
6 alg bool r6 = F1.present => F2.present; //requires
7 alg bool r7 = not (F1.present and F2.present); //excludes

```

A valid configuration is obtained if and only if all feature constraints are satisfied. To this end, the algebraic expressions for the separate feature constraints, such as in Listing 4.3, can be used. In Listing 4.4 we introduce an algebraic variable `sys_valid` that evaluates to true if and only if all feature constraints are satisfied. We also define automaton `Validity` in lines 3-5 in Listing 4.4. At the moment this automaton only states that the system initially is in a valid system configuration.

Listing 4.4: Validity of configuration.

```

1 alg bool sys_valid = r1 and r2 and r3 and ...;
2
3 plant automaton Validity:
4   location: initial sys_valid; marked;
5 end

```

In an extended feature model, an attribute may be assigned to a feature. To model an attributed feature as an automaton, next to the presence feature one or more variables need to be declared to represent the attribute. In Listing 4.5 an example is given for a ball feature. The ball feature is attributed with a color, that can either be red, yellow, or blue. When the ball feature is not present, the color is not available (NA). The domain of the color enumerator is defined in line 1 of Listing 4.5. In line 5 the `color` variable is defined for the ball feature; it is an algebraic variable that takes the value defined in the feature automaton instantiation (`clr`) when the feature is present, and is NA otherwise.

Listing 4.5: Attributed ball feature.

```

1 enum colordomain = red, yellow, blue, NA;
2
3 plant def BallFeature(alg colordomain clr):
4   disc bool present in any;
5   alg colordomain color = if present : clr else NA end;
6   location: initial; marked;
7 end

```

```

8
9 RedBall: BallFeature(red);
10 YellowBall: BallFeature(yellow);

```

Example 4.2 Static feature model for coffee machine in CIF

The CIF specification of the feature model for the coffee machine is given in Listing 4.6². Two different plant definitions for features are used; one without any attributes, and one with a cost attribute. The plant definition for the attributed feature can be instantiated with an input variable defining the cost value. An algebraic integer variable `cost_sum` is introduced in line 39, as the sum of costs of all (present) features. Then, an algebraic Boolean is defined that is true when `cost_sum` is 30 or less. In the `Validity` automaton, it is defined that initially both the feature and cost constraints are satisfied. Construction of the state space of this model in CIF results in a structure with 16 allowed configurations, each represented by an initial state (and nothing more as we have not yet modeled any behavior).

Listing 4.6: Feature instances of the coffee machine.

```

1 plant def FEATURE():
2   disc bool present in any;
3   location: initial ; marked;
4 end
5
6 plant def FEATURE_ATTRIBUTED(alg int x):
7   disc bool present in any;
8   alg int cost = if present : x else 0 end;
9   location: initial ; marked;
10 end
11
12 FM : FEATURE();
13 FS : FEATURE_ATTRIBUTED(5);
14 FO : FEATURE();
15 FR : FEATURE_ATTRIBUTED(5);
16 FB : FEATURE();
17 FX : FEATURE_ATTRIBUTED(10);
18 FE : FEATURE_ATTRIBUTED(5);
19 FD : FEATURE_ATTRIBUTED(5);
20 FP : FEATURE_ATTRIBUTED(7);
21 FC : FEATURE_ATTRIBUTED(5);
22 FT : FEATURE_ATTRIBUTED(3);
23
24 alg bool r1 = FM.present <=> true;
25 alg bool r2 = FM.present <=> FS.present;
26 alg bool r3 = FM.present <=> FO.present;
27 alg bool r4 = FR.present => FM.present;
28 alg bool r5 = FM.present <=> FB.present;
29 alg bool r6 = FX.present => FM.present;

```

² The CIF models used in this chapter are available here: https://github.com/sbthuijsman/TECS_PLE

```

30 alg bool r7 = (FE.present <=> (not(FD.present) and FO.present))
    and (FD.present <=> (not(FE.present) and FO.present));
31 alg bool r8 = FP.present => FB.present;
32 alg bool r9 = FB.present <=> FC.present;
33 alg bool r10 = FT.present => FB.present;
34 alg bool r11 = FP.present => FR.present;
35 alg bool r12 = not(FD.present and FP.present);
36
37 alg bool sys_valid = r1 and r2 and r3 and r4 and r5 and r6 and
    r7 and r8 and r9 and r10 and r11 and r12;
38
39 alg int cost_sum = FS.cost+FR.cost+FX.cost+FE.cost+FD.cost+FP.
    cost+FC.cost+FT.cost;
40 alg bool cost_valid = cost_sum <= 30;
41
42 plant automaton Validity:
43   location: initial sys_valid and cost_valid; marked;
44 end

```

4.4 Dynamic configuration

In the setting discussed in the previous section, the configuration is decided upon initialization of the system and cannot change at any later stage. In this section we consider the situation that features may configure dynamically.

Different types of reconfiguration can be imagined. For example, it can be decided if reconfigurations take place in isolation, or may occur simultaneously. We can for example consider the replacement of the euro feature with the dollar feature. If we do this by first removing the euro feature, and then adding the dollar feature in a next action, these are separate reconfigurations in isolation. If we replace the euro feature with the dollar feature in a single action, this is simultaneous reconfiguration. Both alternatives can be modeled in CIF, and are respectively discussed in Sections 4.4.1 and 4.4.2. The decision on which alternative to use for a model is generally case-specific: can simultaneous reconfiguration practically be achieved, or are reconfiguration actions always performed one-by-one as in single feature configuration? One can also create a model that contains a combination of simultaneous and single feature reconfiguration. Nevertheless, the supervisory controller that we will generate in Section 4.7 is always correct-by-construction for the model, and will operate correctly for the physical system if that is adequately represented in the model.

If one allows models to dynamically configure, there may be situations where a specific change in configuration would result in a violation of the feature constraints. For the example we mentioned above for reconfiguration in isolation, temporarily neither the euro nor the dollar feature is present. Hence, it must be decided if such violations of the feature constraints are allowed, this is discussed in Section 4.4.3.

4.4.1 Single feature reconfiguration

In Section 4.3, for each feature an automaton with a variable named `present` is introduced that captures whether the feature is present. To allow change of presence status of a feature, the value of the corresponding `present` variable needs to be able to change. This can be modeled with a relatively small adaptation to the current feature definition. For each feature a `come` and `go` event are introduced that represent the addition and removal of the feature from the configuration. The resulting feature definition is shown in Listing 4.7. The events `come` and `go` are defined inside the automaton. As a consequence, there is an instance of both events for each instance of the plant definition. These events are referred to by, e.g., `FM.come` or `FS.go`. The same method of adapting the feature automaton definition to allow for dynamic configuration applies for attributed features.

In our examples the `come` and `go` events are chosen to be uncontrollable. This essentially means the supervisory controller safeguards the behavior of the system in the presence of uncontrollable reconfiguration. When the `come` and `go` events are controllable this means the supervisory controller can influence when particular reconfigurations can happen or not dependent on the system state. It is system dependent whether the `come` and `go` events should be modeled controllable or uncontrollable. It is also possible to have a mix of controllable and uncontrollable reconfiguration in the same model.

Listing 4.7: Automaton definition for features with reconfiguration.

```
1 plant def FEATURE():
2   uncontrollable come, go;
3   disc bool present in any;
4   location: initial; marked;
5   edge come when not present do present:=true;
6   edge go when present do present:=false;
7 end
```

By instantiating the plant definition from Listing 4.7 for all features, a state space is obtained that contains each possible reconfiguration, also those that are invalid by the feature model. Restricting reconfiguration to valid configurations is discussed in Section 4.4.3.

4.4.2 Multi feature reconfiguration

In Section 4.4.1 we discussed a situation where features can come and go only one at a time. In some cases it may be desirable to update the presence of multiple features at a time. For example, the dollar feature of the coffee machine can be removed and simultaneously the euro feature can be added. In this way, this reconfiguration can take place without violating the feature constraints, which is impossible with single feature configuration.

To allow updating of the presence variable of multiple feature automata at the same time, a global event can be introduced on which the automata synchronize and their presence variable is updated. In Listing 4.8 features F1 and F2 are alternative features. The event `swap12` is declared to define exchanging F1 and F2. By the restriction of the `Validity` automaton, the system is initially in a valid configuration. Because of multi feature reconfiguration, F1 and F2 can be swapped without the system being in an invalid configuration, i.e., always either F1 or F2 is present.

Listing 4.8: Feature automata with multi feature configuration.

```

1 uncontrollable swap12;
2
3 plant F1:
4   disc bool present in any;
5   location: initial ; marked;
6   edge swap12 do present:=not (present);
7 end
8 plant F2:
9   disc bool present in any;
10  location: initial ; marked;
11  edge swap12 do present:=not (present);
12 end
13
14 alg bool r1 = (F1.present and not (F2.present)) or (F2.present
15   and not (F1.present));
16 alg bool sys_valid = r1;
17 plant automaton Validity:
18   location: initial sys_valid; marked;
19 end

```

4.4.3 Strictness of the feature constraints

By allowing features to enter or leave the system, the feature constraints may be violated temporarily. Two approaches towards the applicability of the feature constraints during reconfiguration are discussed: (1) violation of feature constraints is strictly prohibited, and (2) feature constraints may be violated temporarily.

4.4.3.1 Strict feature constraints

Restricting reconfigurations to valid configurations can be achieved by adding a plant invariant such as presented in Listing 4.9. Adding this invariant removes all states where `sys_valid` and `cost_valid` evaluates to `false`, and all transitions toward these states in the plant's behavior.

Listing 4.9: Invariant restricting reconfiguration to valid configurations.

```

| plant invariant sys_valid and cost_valid;

```

Example 4.3 Reconfiguration with strict feature configuration

Consider the coffee machine from Example 4.2, but now with single feature reconfiguration as discussed in Section 4.4.1, and reconfiguration restricted such that `sys_valid` and `cost_valid` are always true. The state space of this system is shown in Figure 4.3. Note that this is a single automaton, even though it consists of two unconnected parts. Just as for the static configuration, there are 16 (initial) states that represent the valid configurations. Now, switching between configurations is possible by the come and go events.

Since not all states in the state space are connected to each other, we conclude that for some initial configurations, it is not possible to reconfigure to some other configurations. The seven states on the left hand side are all configurations equipped with the dollar feature, the nine states on the right hand side are all configurations

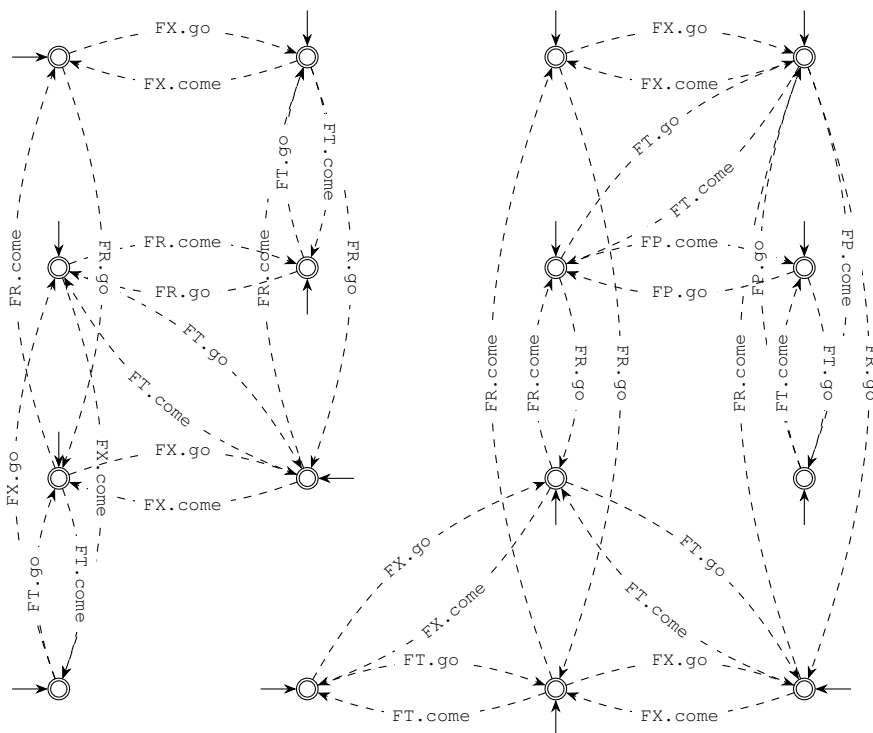


Figure 4.3: State space of the coffee machine with reconfiguration.

equipped with the euro feature. In this model, it is impossible to reconfigure from a euro to dollar feature and vice versa. This is because for single feature reconfiguration, during reconfiguration either both the euro and dollar feature are present or both are not present, which are invalid configurations.

4.4.3.2 Relaxed feature constraints

It may be desirable to temporarily allow violation of feature constraints during a reconfiguration phase, where the system configuration moves from one valid configuration to another. This would allow any feature to configure at any moment. Consequently, the system may get into a configuration that does not satisfy the feature constraints.

It should be noted that one may feel the need to express that some of the feature constraints really need to be satisfied at all times. Of course this can still be enforced.

Example 4.4 Constraints during reconfiguration

One may allow invalid configurations to be reached. However, for the coffee machine there may be a constraint when the change feature is present, the coin feature must always be present. This is achieved using the model fragment from Listing 4.10. The resulting state space consists of 1,364 states, among which 16 initial states. There are 13,440 come and go transitions. Given the possibilities offered by CIF and the modularly defined feature constraints, it is possible to make more complex constraints.

Listing 4.10: Coin feature present when change feature is present.

```
1 plant invariant FX.present => FO.present;
```

4.5 Modeling of uncontrolled behavior

Next, the modeling of uncontrolled behavior of the system components is discussed. In Section 4.6, requirements are defined on this behavior. This can then be used to synthesize a supervisory controller, which is discussed in Section 4.7.

4.5.1 Behavior of the uncontrolled system

The plant modeling aims at capturing all uncontrolled behavior regardless of features, solely focusing on the potential behavior of the physical components.

Example 4.5 Coffee machine component behavior

We model the uncontrolled behavior of the individual components of the coffee machine. The component-wise behavioral specification of this machine is taken from ter Beek et al. (2016).

The system constitutes of the following components: *Coffee*, *Tea*, *Sweet*, *Ringtone*, *Coin*, *Cancel*, and *Machine*. For each of the components an automaton is provided that describes its behavior, see Figure 4.4. Although the different models use the same event names (*done*), because the events are defined within the automata, they are different, and do not synchronize.

The system that is composed of these seven components has a state space of 18 states and 207 transitions, when there is no imposed (supervisory) control, i.e., the events can occur at any time that they are defined in the system.

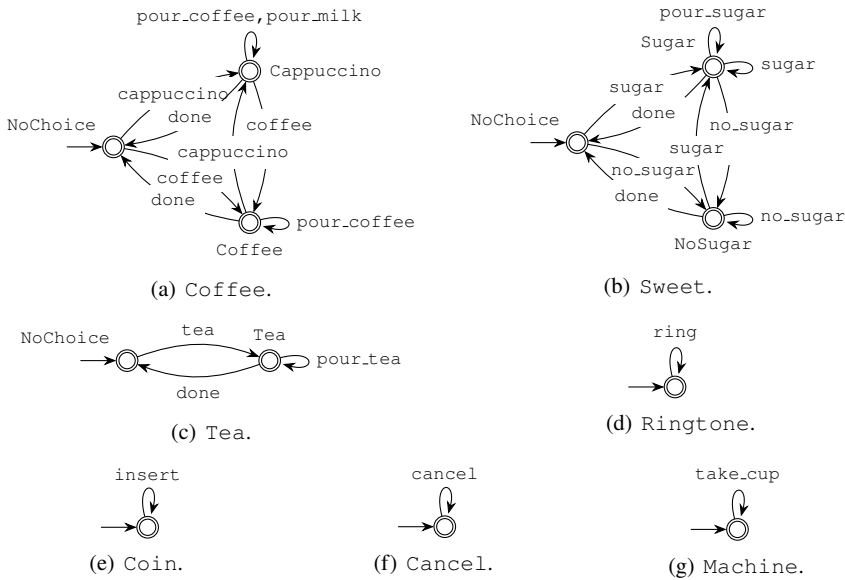


Figure 4.4: Plant automata for the coffee machine.

The CIF model consisting of the component automata does not yet take into account that in specific configurations specific components are not allowed to show behavior, because they are ‘connected’ to features that are not present. For example, the event *ring* of component *Ringtone* (denoted *Ringtone.ring*) is only available in case the *ringtone* feature is part of the configuration.

In the coffee machine example, for each component there is a one-to-one correspondence with the features. In general we require the modeler to indicate for each

event that occurs in a component model which features need to be present for that event to be able to occur. Note that in some cases the availability of an event may also be dependent on the value of some attribute. In CIF this can then be captured by means of additional conditions on such events. For an event e that requires the presence of feature $F1$ and an attribute value of x for attribute A of feature $F2$, this is achieved as shown in Listing 4.11. The connection is stated in the form of a plant, because it models the physical incapability to perform some events when certain features are not present.

Listing 4.11: Definition of link between events and features.

```

1 plant automaton event_feature_conditions:
2   location: initial; marked;
3     edge e when F1.present and F2.A = x;
4 end

```

Example 4.6 Connection between events and features for the coffee machine

For the coffee machine, the connection between events and features is captured by the plant `event_feature_link` in Listing 4.12.

Listing 4.12: Connection between events and features.

```

1 plant automaton event_feature_link:
2   location: initial; marked;
3     edge Coffee.cappuccino when FC.present and FP.present;
4     edge Coffee.coffee when FC.present;
5     edge Coffee.done when FC.present;
6     edge Coffee.pour_coffee when FC.present;
7     edge Coffee.pour_milk when FC.present and FP.present;
8
9     edge Tea.done when FT.present;
10    edge Tea.pour_tea when FT.present;
11    edge Tea.tea when FT.present;
12
13    edge Sweet.done when FS.present;
14    edge Sweet.no_sugar when FS.present;
15    edge Sweet.pour_sugar when FS.present;
16    edge Sweet.sugar when FS.present;
17
18    edge Ringtone.ring when FR.present;
19
20    edge Coin.insert when FO.present;
21
22    edge Cancel.cancel when FX.present;
23
24    edge Machine.take_cup when FM.present;
25 end

```

The way we expressed the availability of events in relation to the presence of features is conceptually similar to the solution adopted in featured transition systems (Classen et al. 2013). In these featured transition systems events are also available conditionally depending on feature presence. The most prominent difference between this chapter and the approach using featured transition systems is that here the description of the relation between features and events is separated from the behavioral models of the components. Another difference is that in the approach using featured transition systems not the uncontrolled system and requirements are modeled, but the supervisory controller is developed directly.

4.5.2 Component reappearance

As a result of reconfiguration, a component may leave or enter the system repeatedly. Initially, the components are in their initial state as defined in Section 4.5.1. Sometimes it may be required to reinitialize or reset the state of the component when it leaves or enters the system.

Until now, the appearance and disappearance of features is not directly affecting the states of the involved components. Therefore, when a feature disappears, and in a future configuration reappears, the components linked with this feature are still in the same state.

The modeler can easily adapt the plant model such that, for example, a component transitions to some desired reset state whenever the component enters or leaves the system. For example by adding an edge, labeled with the `come` or `go` event of the respective component, from each state in the plant model of the component to its desired reset state. Because of synchronization, upon occurrence of the `come` or `go` event (from the feature plant) the transition with the same label in the component is taken as well. When the plant model of the component has an outgoing transition labeled with this reconfiguration transition from each state, the proposed addition does not restrict reconfiguration possibilities. Otherwise, the reconfiguration event can only take place when the component is in a state where the transition is defined.

Example 4.7 Re-initialization in the coffee machine

Let us consider the case that we want the tea component to go to the `NoChoice` location when it leaves the system. Applying the proposed approach results in the adapted plant automaton shown in Figure 4.5. If the tea component leaves the system, i.e., event `FT.go` occurs, when the automaton is in the `Tea` location, it will transition to `NoChoice`. If the event occurs when the automaton is already in `NoChoice` it will remain there. Note that in this example the automaton does not use the event `FT.come` and is therefore not influenced when that particular event occurs.

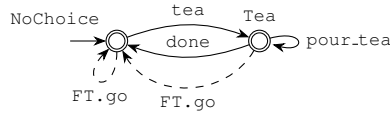


Figure 4.5: Adapted plant for resetting the tea component in case of removal from the configuration.

4.6 Specification of requirements

In the previous section, we have discussed how to model the uncontrolled system and how to link event occurrences to availability of features. In this section, we discuss the modeling of requirements. In Section 4.6.1 the specification of behavioral requirements is discussed. In Section 4.6.2 we elaborate on the specification of requirements in the setting of dynamic configuration.

4.6.1 Behavioral requirements

As explained in Section 4.2, requirements are specified by means of automata that synchronize with the plant or by using state-based expressions such as event conditions and state invariants (Markovski et al. 2010). The example below demonstrates several ways to specify the formal requirements, given a set of informal requirements.

Example 4.8 Requirements coffee machine

Below we state the informal system requirements and their corresponding CIF formulation. Requirement automata, event conditions, and state invariants are used. These requirements can be added to the model containing the plant behavior and the feature model.

1. The coffee and tea component cannot both be ready to pour:

```
1 requirement not (Coffee.Coffee and Tea.Tea);
```

2. Coffee, cappuccino, or tea can only be selected when no choice between them has been made:

```

1 requirement Coffee.coffee needs Coffee.NoChoice and Tea.NoChoice;
2 requirement Coffee.cappuccino needs Coffee.NoChoice and Tea.NoChoice;
3 requirement Tea.tea needs Coffee.NoChoice and Tea.NoChoice;

```

Note that, in addition to these requirements, these events can only occur when their respective feature is present, because of the connection between the events and feature presence discussed in Section 4.5.

3. When the ringtone feature is present, it may only ring (once) after the coffee or tea component is finished:

```

1 requirement automaton RingAfterBeverageCompletion:
2   location NotCompleted:
3     initial; marked;
4     edge Coffee.done when FR.present goto Completed;
5     edge Tea.done when FR.present goto Completed;
6     edge Coffee.done, Tea.done when not FR.present;
7   location Completed:
8     edge Ringtone.ring goto NotCompleted;
9 end

```

This requirement shows how the behavioral requirements can be made dependent on the presence of features. When the ringtone feature is not present, this requirement automaton will stay in `NotCompleted`. When the ringtone feature is present, after pouring coffee or tea is done, the automaton needs to transition to the `Completed` location before the ringtone can be performed.

4. A coin needs to be present to make a selection of beverage and sugar:

```

1 plant automaton CoinPresence:
2   monitor;
3   location NoCoinPresent:
4     initial; marked;
5     edge Coin.insert goto CoinPresent;
6   location CoinPresent:
7     edge Cancel.cancel goto NoCoinPresent;
8     edge Machine.take_cup goto NoCoinPresent;
9 end
10
11 requirement Coffee.coffee needs CoinPresence.CoinPresent;
12 requirement Coffee.cappuccino needs CoinPresence.CoinPresent;
13 requirement Tea.tea needs CoinPresence.CoinPresent;
14 requirement Sweet.sugar needs CoinPresence.CoinPresent;
15 requirement Sweet.no_sugar needs CoinPresence.CoinPresent;

```

This requirement showcases the use of a monitor automaton. In literature this is sometimes also called an observer automaton. The automaton `CoinPresence` never disables any event, but simply tracks whether a coin is present in the system. The state

of `CoinPresence` is used in the subsequent event conditions so that only a selection can be made when a coin is present.

5. Coffee is only poured once:

```

1 plant automaton CoffeePoured:
2   monitor;
3   location NotPoured:
4     initial;marked;
5     edge Coffee.pour_coffee goto Poured;
6   location Poured:
7     edge Machine.take_cup goto NotPoured;
8 end
9
10 requirement Coffee.pour_coffee needs CoffeePoured.NotPoured;

```

Once more, a monitor automaton is used. This automaton tracks whether Coffee has been poured. Note that we have similar informal requirements with CIF formalizations for only pouring tea and milk once; resulting in automata `TeaPoured` and `MilkPoured`.

6. Coffee and tea should not be mixed, and tea and milk should not be mixed:

```

1 requirement not (CoffeePoured.Poured and TeaPoured.Poured);
2 requirement not (TeaPoured.Poured and MilkPoured.Poured);

```

Here the previously defined states of the monitor automata are used in additional state invariants.

7. When coffee is selected, the coffee component is done after pouring only coffee. When cappuccino is selected, the coffee component is done after pouring both coffee and milk:

```

1 requirement Coffee.done needs (Coffee.Coffee and CoffeePoured.
   Poured) or (Coffee.Cappuccino and CoffeePoured.Poured and
   MilkPoured.Poured);

```

8. When sugar is selected, it is poured twice:

```

1 requirement automaton PourSugarTwice:
2   disc int[0..2] count=0;
3   location Idle:
4     initial; marked;
5     edge Sweet.sugar goto SugarNeeded;
6     edge Sweet.done when Sweet.NoSugar;
7     edge Machine.take_cup do count:=0;
8   location SugarNeeded:
9     edge Sweet.pour_sugar when count<2 do count:=count+1;
10    edge Sweet.done when count=2 goto Idle;
11    edge Machine.take_cup do count:=0;
12 end

```

Here we use a requirement automaton that is an extended finite automaton to keep track of a count of how many times sugar has been poured. Pouring sugar is only finished after it has been performed twice, and the counter is reset to zero when the cup is taken from the system.

9. Cancellation is only possible before anything has been poured:

```

1 requirement Cancel.cancel needs CoffeePoured.NotPoured and
   TeaPoured.NotPoured and MilkPoured.NotPoured and
   PourSugarTwice.count=0;

```

10. The cup can only be taken from the machine when the pouring of coffee, tea, and sugar is done and no new selection was made:

```

1 requirement automaton TakeCupWhenCoffeeOrTeaDone:
2   location NotPoured:
3     initial;marked;
4     edge Coffee.done goto Done;
5     edge Tea.done goto Done;
6   location Done:
7     edge Machine.take_cup goto NotPoured;
8 end
9
10 requirement automaton TakeCupWhenSugarDone:
11   location NotPoured:
12     initial;marked;
13     edge Sweet.done goto Done;
14   location Done:
15     edge Machine.take_cup goto NotPoured;
16 end
17
18 requirement Machine.take_cup needs Coffee.NoChoice and Sweet.
   NoChoice and Tea.NoChoice;

```


4.6.2 Requirements during configuration

As discussed in Section 4.4.3, it may be beneficial to allow invalid configurations so that reconfiguration can take place. In this case, decisions must be made about allowed behavior in such configurations. There are several ways to deal with the specification of allowed behavior during the configuration phase: (1) disable some events from occurring, and (2) additional requirements. Each of these approaches may be suitable for certain applications. In the following subsections these possibilities are investigated.

4.6.2.1 Disabling events

An approach to restrict the behavior during reconfiguration is to disable some events in case the system is in an invalid configuration. For each such an event, an event condition, such as the one presented for event e in Listing 4.13, can be defined that restricts that event to occur only when the system is in a valid configuration.

Listing 4.13: Disabling an event in an invalid system configuration.

```
1 requirement e needs sys_valid;
```

This approach assumes that the system will exhibit safe behavior by not exercising any of the events disabled in this way. As soon as the system returns to a valid configuration these events are no longer disabled. Such event disablement requirements can also be expressed for particular configurations, as discussed in the next example.

Example 4.9 Event disablement in coffee machine during reconfiguration

In the coffee machine, invalid configurations may sometimes be allowed. For example, exchanging the euro and dollar feature through two subsequent single feature reconfiguration events is achieved by either having none or both features present during the reconfiguration. Let us consider the case that both the euro and dollar feature are present. It may be unsafe to cancel the order in this situation, as it is unclear from which feature the coin should be returned. This can be avoided by adding a requirement as given in Listing 4.14.

Listing 4.14: Constraint during invalid configuration.

```
1 requirement Cancel.cancel needs not(FE.present and FD.present);
```

4.6.2.2 Additional requirements

Another approach is stating additional requirements for the transitional situation. For the coffee machine these are detailed in the next example.

Example 4.10 Dynamic configuration constraints for the coffee machine

We consider the situation that it is allowed that the sweet feature is not present during reconfiguration. However, in case the sweet feature is currently ready to pour sugar, it needs to always be present. Listing 4.15 shows a formalization of this requirement, that restricts the possible (invalid) configurations that can be reached depending on the current state of the components.

Listing 4.15: Constraint during invalid configuration.

```
1 requirement Sweet.Sugar => FS.present;
```

4.7 Supervisory controller synthesis

The automata model obtained from the feature model in Section 4.4, the uncontrolled behavior specification from Section 4.5, and the behavioral requirements from Section 4.6 can be placed into a single model. Next, supervisory control synthesis (Ramadge and Wonham 1987) can be applied. A supervisory controller is generated as an automaton that controls the system through synchronizing events. The system under control consists of the supervisor synchronized with all plant and requirement automata. By construction of the supervisor, the system under control is:

1. *Safe*: the requirements are always adhered to. If in the synchronous product the plant can execute an event, but it is not possible in a requirement automaton that synchronizes on that event, the event is prevented from occurring.
2. *Nonblocking*: a marked state can always be reached. Since we modeled the system using multiple automata, this means in the controlled behavior it is always possible to follow a sequence of events such that all automata are simultaneously in a marked state. For example, if we consider the `CoinPresence` requirement of Section 4.6.1, we are sure that the system can always return to the state that no coins are present.
3. *Controllable*: the supervisor does not disallow uncontrollable events to occur. In our model of the coffee machine all reconfiguration actions are uncontrollable. The supervisor always allows these to happen, and prevents the system from reaching states that are not allowed in a particular configuration, if we may uncontrollably reconfigure to that configuration.
4. *Maximally permissive*: no behavior is disabled that does not strictly need to be disallowed to satisfy the aforementioned properties. This makes sure the supervisory controller does not restrict any behavior that is perfectly fine to occur.

Note that a single supervisory controller is generated that applies to all system configurations.

Example 4.11 Supervisory controller synthesis for the coffee machine

We consider the coffee machine with dynamic single feature configuration, where invalid configurations are never allowed, and the requirements of Section 4.6 are applied. Applying supervisory controller synthesis to the described system results in a supervisory controller that applies the guards formulated in Listing 4.16 to the controllable events. Note that the guard for `Cancel.cancel` is not displayed because the expression is very long. The state space of the system under control contains 6,240 states and 35,336 transitions.

Listing 4.16: Additional guards provided by supervisory controller synthesis.

```

1 supervisor automaton sup:
2   alphabet Coin.insert, Cancel.cancel, Sweet.sugar, Sweet.
   no_sugar, Sweet.pour_sugar, Sweet.done, Ringtone.ring,
   Coffee.cappuccino, Coffee.coffee, Coffee.pour_coffee, Coffee
   .pour_milk, Coffee.done, Tea.tea, Tea.pour_tea, Tea.done,
   Machine.take_cup;
3   location:
4     initial;
5     marked;
6     edge Cancel.cancel when ...;
7     edge Coffee.cappuccino when CoinPresence.CoinPresent and
   not Coffee.Coffee and (Tea.NoChoice and
   TakeCupWhenCoffeeOrTeaDone.NotPoured);
8     edge Coffee.coffee when CoinPresence.CoinPresent and not
   Coffee.Cappuccino and (Tea.NoChoice and
   TakeCupWhenCoffeeOrTeaDone.NotPoured);
9     edge Coffee.done when not Coffee.Cappuccino and
   CoffeePoured.Poured or Coffee.Cappuccino and (CoffeePoured.
   Poured and MilkPoured.Poured);
10    edge Coffee.pour_coffee when CoffeePoured.NotPoured;
11    edge Coffee.pour_milk when MilkPoured.NotPoured;
12    edge Coin.insert when true;
13    edge Machine.take_cup when true;
14    edge Ringtone.ring when true;
15    edge Sweet.done when true;
16    edge Sweet.no_sugar when CoinPresence.CoinPresent and not
   Sweet.Sugar and (PourSugarTwice.Idle and
   TakeCupWhenSugarDone.NotPoured) or (CoinPresence.CoinPresent
   and (not Sweet.Sugar and PourSugarTwice.SugarNeeded) or
   CoinPresence.CoinPresent and (Sweet.Sugar and PourSugarTwice
   .count = 2));
17    edge Sweet.pour_sugar when true;
18    edge Sweet.sugar when CoinPresence.CoinPresent and
   TakeCupWhenSugarDone.NotPoured;
19    edge Tea.done when true;
20    edge Tea.pour_tea when TeaPoured.NotPoured;
21    edge Tea.tea when CoinPresence.CoinPresent and (Coffee.
   NoChoice and TakeCupWhenCoffeeOrTeaDone.NotPoured);
22 end

```

4.8 Case study: Body Comfort System

We present a case study on the Body Comfort System (BCS) (Lity et al. 2013), which is a frequently used benchmark in (S)PLE-related literature (Lochau et al. 2014; Fragal et al. 2017; Lity et al. 2017; Lachmann et al. 2016). It is a product line originating from the automotive industry. It contains a number of standard and optional features, such as LED's in the human machine interface, manual or automatic windows, security options such as an alarm system, and more. The feature model of the BCS is given in Figure 4.6. This feature model allows for 11,616 valid configurations. Note that for this case study, only features without attributes are considered.

The feature model is modeled in CIF as outlined in Section 4.3, this results in Listing 4.17.

Listing 4.17: BCS feature model in CIF.

```

1 plant def FEATURE():
2   uncontrollable come,go;
3   disc bool present in any;
4   location: initial ; marked;
5     edge come when not present do present:=true;
6     edge go when present do present:=false;
7 end
8
9 // Feature declaration by level in FM
10 // Level 1
11 FBCS:FEATURE();
12 // Level 2
13 FHMI:FEATURE(); FDoor:FEATURE(); FSecu:FEATURE();
14 // Level 3
15 FPowerW:FEATURE(); FMir:FEATURE(); FAlarm:FEATURE();FRCKey:
    FEATURE();FCLS:FEATURE();
16 // Level 4
17 FLED:FEATURE(); FFingerP:FEATURE(); FAutoPW:FEATURE(); FManPW:
    FEATURE(); FMirE:FEATURE(); FMirHeat:FEATURE(); FInterMon:
    FEATURE(); FCtrAlarm:FEATURE(); FCtrAutoPW:FEATURE(); FSafe:
    FEATURE(); FAdjMir:FEATURE(); FAutoL:FEATURE();
18 // Level 5
19 FLEDMir:FEATURE(); FLEDFP:FEATURE(); FLEDPW:FEATURE(); FLEDCLS:
    FEATURE(); FLEDAlarm:FEATURE(); FLEDHeat:FEATURE();
20
21 // Feature relations
22 // Level 1
23 alg bool r11 = FBCS.present; //Root feature present
24 // Level 2
25 alg bool r21 = FBCS.present <=> FHMI.present; //HMI mandatory
26 alg bool r22 = FBCS.present <=> FDoor.present; //Door mandatory
27 alg bool r23 = FSecu.present => FBCS.present; //Security
    optional
28 // Level 3
29 alg bool r31 = FDoor.present <=> FPowerW.present; //PW
    mandatory

```

```

30 alg bool r32 = FDoor.present <=> FMir.present; //EM mandatory
31 alg bool r33 = FAlarm.present => FSecu.present; //AS optional
32 alg bool r34 = FCLS.present => FSecu.present; //CLS optional
33 alg bool r35 = FRCKey.present => FSecu.present; //RCK optional
34 // Level 4
35 alg bool r41 = FLED.present => FHMI.present; //LED optional
36 alg bool r42 = (FManPW.present <=> (not FAutoPW.present and
    FPowerW.present)) and (FAutoPW.present <=> (not FManPW.
    present and FPowerW.present)); //Manual or automatic PW
37 alg bool r43 = FPowerW.present <=> FFingerP.present; //Finger
    Protection mandatory
38 alg bool r44 = FMir.present <=> FMirE.present; //Electric
    exterior mirror mandatory
39 alg bool r45 = FMirHeat.present => FMir.present; //Mirror
    heating optional
40 alg bool r46 = FInterMon.present => FAlarm.present; //Interior
    monitoring optional
41 alg bool r47 = FCtrAlarm.present => FRCKey.present; //Control
    alarm optional
42 alg bool r48 = FCtrAutoPW.present => FRCKey.present; //Control
    automatic power window optional
43 alg bool r49 = FSafe.present => FRCKey.present; //Safety
    optional
44 alg bool r410 = FAdjMir.present => FRCKey.present; //Adjust
    exterior mirror optional
45 alg bool r411 = FAutoL.present => FCLS.present; //Automatic
    locking optional
46 // Level 5
47 alg bool r51 = FLED.present <=> (FLEDAlarm.present or FLEDFP.
    present or FLEDCLS.present or FLEDPW.present or FLEDMir.
    present or FLEDHeat.present);
48 //cross tree relations
49 alg bool rx1 = FLEDAlarm.present => FAlarm.present; //LED
    alarm requires Alarm
50 alg bool rx2 = FLEDCLS.present => FCLS.present; //LED central
    requires central locking
51 alg bool rx3 = FLEDHeat.present => FMirHeat.present; //LED
    heat mirror requires heated mirror
52 alg bool rx4 = not (FManPW.present and FCtrAutoPW.present);
    //Manual power windows excludes control autoPW
53 alg bool rx5 = FCtrAlarm.present => FAlarm.present; //Control
    alarm requires Alarm system
54 alg bool rx6 = FRCKey.present => FCLS.present; //Remote
    control key requires central locking system
55
56 alg bool sys_valid = r11 and r21 and r22 and r23 and r31 and
    r32 and r33 and r34 and r35 and r41 and r42 and r43 and r44
    and r45 and r46 and r47 and r48 and r49 and r410 and r411
    and r51 and rx1 and rx2 and rx3 and rx4 and rx5 and rx6;
57
58 plant automaton Validity:
59   location: initial sys_valid; marked;
60 end

```

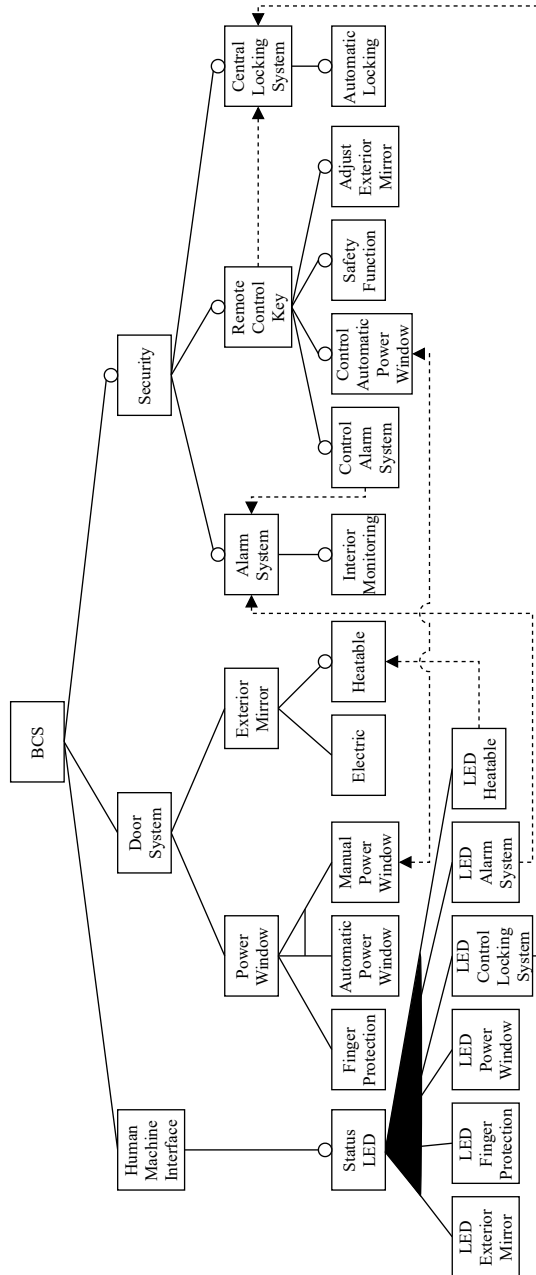


Figure 4.6: Feature model of the Body Comfort System (Lity et al. 2013).

In this model, the system is initially in a valid configuration, but can reconfigure to any configuration through the come and go events. I.e., all invalid configurations are allowed during reconfiguration. This model (without any component behavior added) has 134,217,728 reachable states. Of these states, 11,616 states are initial states representing the valid configurations.

In Lity et al. (2013) state machine test models are constructed for product instances of the BCS. In Tuitert (2017) a component wise behavioral model of the BCS is made, as an interpretation of the models in Lity et al. (2013). We will use the models from Tuitert (2017) here.

As an example, we consider the alarm system and interior monitoring features. The uncontrolled behavior of the relevant components is defined by the automata shown in Listing 4.18. In the event names, "c_" and "u_" are used as prefixes to respectively indicate controllable or uncontrollable events.

Listing 4.18: BCS uncontrolled behavior of the alarm system and interior monitoring.

```

1 plant automaton AlarmSystem:
2   controllable c_on, c_off, c_deactivated, c_activated,
   c_IM_detected;
3   uncontrollable u_detected, u_time_elapsed;
4   location Deactivated:
5     edge c_activated goto Activated;
6   location Activated:
7     initial; marked;
8     edge c_on goto On;
9     edge c_deactivated goto Deactivated;
10  location On:
11   edge c_off goto Activated;
12   edge u_detected goto Alarm_detected;
13   edge c_IM_detected goto Alarm_detected;
14  location Alarm_detected:
15   edge c_off goto Activated;
16   edge u_time_elapsed goto On;
17 end
18
19 plant automaton InteriorMonitoring:
20  uncontrollable u_detected, u_clear;
21  controllable c_on, c_off;
22  location Off:
23   initial; marked;
24   edge c_on goto On;
25  location On:
26   edge c_off goto Off;
27   edge u_detected goto Detected;
28  location Detected:
29   edge u_clear goto On;
30   edge c_off goto Off;
31 end

```

The events of the automata are linked to their presence in Listing 4.19. One can observe that the events of AlarmSystem and InteriorMonitoring are depen-

dent on the presence features `FAlarm` and `FInterMon` respectively. However, the event `AlarmSystem.c_IM_detected` requires both features to be present.

Listing 4.19: BCS presence check alarm system.

```

1 plant automaton PRESENCE_UNCONTROLLED_AS:
2   location: initial; marked;
3   edge AlarmSystem.u_detected when FAlarm.present;
4   edge AlarmSystem.u_time_elapsed when FAlarm.present;
5   edge AlarmSystem.c_on when FAlarm.present;
6   edge AlarmSystem.c_off when FAlarm.present;
7   edge AlarmSystem.c_deactivated when FAlarm.present;
8   edge AlarmSystem.c_IM_detected when FAlarm.present and
   FInterMon.present;
9   edge InteriorMonitoring.u_detected when FInterMon.present;
10  edge InteriorMonitoring.u_clear when FInterMon.present;
11  edge InteriorMonitoring.c_on when FInterMon.present;
12  edge InteriorMonitoring.c_off when FInterMon.present;
13 end

```

Requirements for these components are given in Listing 4.20. One can see how these refer to other relevant components in the system such as `Key_lock` and `RCK_CLS`, which are the physical key and the remote control key of the central locking system.

Listing 4.20: BCS requirements alarm system.

```

1 requirement AlarmSystem.c_on needs Key_lock.Locked or RCK_CLS.
   Locked;
2 requirement AlarmSystem.c_off needs Key_lock.Unlocked or
   RCK_CLS.Unlocked;
3 requirement AlarmSystem.c_deactivated needs Key_lock.Unlocked
   or RCK_CLS.Unlocked;
4 requirement AlarmSystem.c_IM_detected needs InteriorMonitoring.
   Detected;
5 requirement InteriorMonitoring.c_off needs Key_lock.Unlocked or
   RCK_CLS.Unlocked;

```

In the complete model³, there are in total 31 plant automata representing the behavior of the components. Additionally, there are 27 feature automata and 18 plant automata that link the component events to the presence of the features. 55 requirements are specified. Only event condition requirements are used. Using CIF, synthesis can successfully be applied to this system.

Some relevant state space sizes are mentioned in Table 4.2. The worst case state space is calculated by calculating the product of the number of states in each automaton. The other state space sizes all denote the reachable states from the initial states. For the uncontrolled systems, no synthesis or requirements are applied yet. For the controlled systems, the state space size is the number of reachable states in the system controlled by the synthesized supervisory controller. Using a standard personal computer and applying supervisory controller synthesis in CIF using the default settings,

³ Note that also all complete CIF models we use of the BCS are available here: https://github.com/sbthuijsman/TECS_PLE

the supervisor is obtained in roughly 0.3 seconds for each of the static and the dynamic case. For these computations, the CIF application requires no more than 0.5 GB of memory. Even though a state space in the order of 10^{20} can be considered large, CIF has been shown capable of performing supervisory controller synthesis for systems with much larger state spaces (Reijnen et al. 2020). It should be noted that, since CIF uses symbolic supervisory controller synthesis using Binary Decision Diagrams (BDDs), the computational effort of synthesis is dependent on more factors than state space size, see Chapter 5. Supervisory controller synthesis for the static case requires 19,614 peak used BDD nodes and 1,864,598 BDD operations. For the dynamic case this is 26,140 peak used BDD nodes and 2,039,318 operations. These BDD-based metrics of computational effort of supervisory controller synthesis are detailed in Chapter 5 and measurements for benchmark systems are provided in the same work.

Table 4.2: State space sizes BCS.

State space	Number of states
Worst case	$7.7 \cdot 10^{20}$
Uncontrolled static	$3.2 \cdot 10^{14}$
Uncontrolled dynamic	$6.2 \cdot 10^{20}$
Controlled static	$7.6 \cdot 10^{13}$
Controlled dynamic	$1.1 \cdot 10^{20}$

Even though the BCS is a frequently used benchmark in literature related to product line engineering (Lity et al. 2013; Lochau et al. 2014; Fragal et al. 2017; Lity et al. 2017; Lachmann et al. 2016), as far as we are aware there is no existing work to which we can compare these results. In fact, only in Tuitert (2017) the first models of the BCS were made that were suitable for application of supervisory control theory.

With the BCS use case, we have shown that CIF is capable of both modeling an industrial sized product line through the use of feature models, and synthesizing a supervisory controller for this system in which dynamic reconfiguration is allowed.

4.9 Concluding Remarks

We have presented a framework for engineering supervisory controllers for product lines of which the valid configurations are described by a feature model and where dynamic configuration of the features is allowed. The CIF language has shown to be adequate for modeling the involved concepts. It was shown how the presence and absence of features can be modeled, and how this presence can update through single or multi feature configuration. Component wise modeling of the system behavior has been demonstrated, where the presence of features influences the possible behavior. It was shown how requirements can be formulated that take the presence of the features into account, and how requirements can be strengthened when the system is in an invalid

configuration. Supervisory controller synthesis can be applied such that the maximal nonblocking, controllable, and safe behavior under control is obtained. The method was demonstrated using the coffee machine as a running example. Although the coffee machine system is small, feasibility of industrial application has been demonstrated with the much larger BCS use case.

By using this framework, supervisory controllers do not have to be computed one-by-one for each product instance in a product line, and also the reconfiguration is directly handled. Presumably, by applying this framework, supervisory controllers for product lines can be engineered more efficiently than doing so without using the product line engineering approach. A case study evaluation on the improvement of engineering efficiency remains future work.

Chapter 5

Reducing the computational effort of symbolic supervisor synthesis

Abstract Supervisor synthesis is a means to algorithmically derive a supervisory controller from a discrete-event model of a system and a requirements specification. For large systems, supervisor synthesis suffers from state space explosion. To handle large state spaces, and thereby large systems, the model can be symbolically represented using Binary Decision Diagrams (BDDs), and supervisor synthesis can be applied to this symbolic representation. Peak used BDD nodes and BDD operation count are introduced as deterministic and platform-independent metrics to express the computational effort of a symbolic supervisor synthesis. These BDD-based metrics can be used to analyze the efficiency of the synthesis algorithm. From this analysis, modifications can be made to the way the BDDs are handled during synthesis, improving the synthesis efficiency. In this paper we showcase this approach by: introducing and analyzing DCSH, a variable ordering heuristic; analyzing several edge ordering heuristics; and introducing and analyzing an approach to efficiently enforce state exclusion requirements in synthesis. These methods have recently been implemented in our open source supervisory control tool, Eclipse ESCET. The analysis is based on large scale experiments of performing synthesis on a variety of models from literature. It is shown that: (1) by using DCSH, performing synthesis with relatively high computational effort can be avoided, and generally relatively low computational effort is required; (2) applying reverse-model edge order realizes relatively low synthesis effort; and (3) state exclusion requirements can efficiently be enforced by restricting edge guards prior to synthesis.

5.1 Introduction

Supervisory control theory (Ramadge and Wonham 1987, 1989) is a model-based approach to control cyber-physical systems. Given a plant (a model that defines all possible system behavior) and a requirements specification (a model that defines what

This chapter is strongly based on: Thuijsman, Sander B.; Hendriks, Dennis; Reniers, Michel A.: Reducing the computational effort of symbolic supervisor synthesis. Submitted to: *Discrete Event Dynamic Systems*.

behavior is allowed), a supervisor can be computed algorithmically (synthesized) that restricts the plant's behavior so that it is in accordance with the requirements specification. Depending on the synthesis algorithm, the supervised system has some useful properties by construction, such as safety, nonblockingness, controllability, and maximal permissiveness. There are a number of formal modeling frameworks to which supervisory control theory can be applied. The framework of *extended finite automata* (EFAs) (Sköldstam et al. 2007) is an extension to *finite state automata* that augments them with variables, guard expressions and updates, which enables more convenient modeling of systems.

The power of supervisory control theory has been demonstrated in literature. There are many examples where it is applied to controller design. We refer to Table 5.1 further down this chapter for a selection. Despite the advantages of supervisory control theory, and demonstration thereof in case studies, industrial acceptance is scarce. Wonham et al. (2018) point to the *state space explosion* as one of the barriers to industrial acceptance. Technically, all possible combinations of states of components in the system must be taken into account. Therefore, adding a small component to the model might induce a large increase to the total system state space. A way to mitigate state space explosion, is by representing the system model using *binary decision diagrams* (BDDs) (Akers 1978; Lee 1959), and performing supervisor synthesis on this symbolic representation (Ma and Wonham 2006; Vahidi et al. 2006; Miremadi et al. 2012). This approach is considered state of the art to handle industrial-sized systems (Malik et al. 2017).

Symbolic supervisor synthesis has been shown to be able to deal with large-scale systems. For instance, monolithic synthesis was successfully performed for a system where the uncontrolled system and supervised system respectively had $2.3 \cdot 10^{57}$ and $4.5 \cdot 10^{34}$ states by Reijnen et al. (2020), which are much larger state spaces than non-symbolic monolithic synthesis could handle. However, as we will also show in this chapter, the amount of time and memory required for symbolic synthesis is majorly impacted by the settings the algorithm is initiated with, and different ways the algorithm can be applied (Vahidi et al. 2006; Thuijsman et al. 2019). It is of practical benefit to optimize the application of the algorithm to minimize time and memory required to perform synthesis (of large-scale systems). Such optimization is difficult, as what techniques are beneficial is often case dependent, and even frequently counter-intuitive, since a BDD representing a small amount of states may require many more BDD nodes than a BDD representing a much larger amount of states (Ciardo and Siminiceanu 2002). Sufficient experimentation and validation is required to judge the efficiency of a method.

A tool that can be used to perform symbolic supervisor synthesis is *CIF* (van Beek et al. 2014). CIF is part of the *Eclipse Supervisory Control Engineering Toolkit* (Eclipse ESCET™)¹ since 2020 (Fokkink et al. 2023). As a result of this open source project, the intensity of the development of the CIF tool has recently greatly increased. Among the many developments that have been made, are implementations of recently proposed

¹ The ESCET toolkit and documentation is open source and freely available at <https://eclipse.org/escet/>. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

methods that aim to improve the computational efficiency of symbolic supervisor synthesis, such as the BDD variable ordering heuristic algorithm from Lousberg et al. (2020) and efficient enforcement of state exclusion requirements from Thuijsman et al. (2021). These methods were previously only available in local proof-of-concept implementations.

This chapter is an extension to Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021). These papers contain many results of elaborate experiments. Because, as mentioned, many developments have taken place for the CIF tool, we re-perform and re-evaluate the results from Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021). We use the BDD-based metrics of Thuijsman et al. (2019) to measure the computational effort of performing symbolic supervisor synthesis. We re-evaluate the impact of the variable order heuristic of Lousberg et al. (2020) and the requirement enforcement of Thuijsman et al. (2021) on the computational effort, now that they are implemented and available to all users. For further validation, the experiments are performed on a larger set of models. Additionally, we make all models publicly available, so that our experiments are repeatable². Note that the experiments are performed on a large scale: running all experiments in a single sequence would require several years of computation time. Instead, we have performed many experiments in parallel on a high-performance computing cluster. In further extension to Thuijsman et al. (2019); Lousberg et al. (2020); and Thuijsman et al. (2021), we investigate various (simple) heuristics for edge ordering to improve synthesis efficiency. We also evaluate how these methods perform together, e.g., the edge ordering heuristics are evaluated using the variable ordering heuristic we introduce. Furthermore, this chapter contains a proof of correctness of the efficient requirement enforcement method, that was not given in Thuijsman et al. (2021). Finally, in this chapter the methods are presented in a more unified way.

There are many ways in which improvements can be made on computational efficiency of symbolic supervisor synthesis. Evidently, we restrict ourselves to a few options in this chapter in order to keep this study contained. The methods that we evaluate have the following in common, they are:

- monolithic approaches: supervisor synthesis is not divided into multiple sub-problems;
- non-restrictive to the input model: the method can be used for synthesis of any plant and requirement specification that CIF supports synthesis of;
- “under the hood”: the user does not have to supply additional parameters or modify their model;
- static: in the sense that optimization is performed only at a single stage, and not on-the-fly (dynamically) during/throughout synthesis.

² All experiments in this chapter are performed using ESCET release v0.9, available here: <https://eclipse.org/escet/v0.9/>. The models are available bundled in ESCET under “CIF Benchmarks”, see <https://eclipse.org/escet/cif/examples.html>. The files to run the same experiments as presented in this chapter are available here: https://github.com/sbthuijsman/reduce_effort.

Related work

The foundations of symbolic supervisor synthesis are discussed in Ma and Wonham (2006). Symbolic supervisor synthesis for (sets of) EFAs is discussed in Ouedraogo et al. (2011) and Fei et al. (2014). In Ziller and Schneider (2003) a supervisor synthesis algorithm is constructed that is based on μ -calculus, and a BDD-based implementation is made. In Vahidi et al. (2006) partitioning and ordering of the transition relation to efficiently perform BDD-based synthesis is investigated. This partitioning, as well as how supervisor guards can efficiently be generated for such a partitioning, is inspected in Fei et al. (2013). A symbolic synthesis approach using hierarchical decomposition is presented in Song and Leduc (2006). In Miremedi and Lennartson (2016) an efficient synthesis algorithm is introduced that is based on forward reachability rather than backward reachability to avoid unnecessary exploration of states, of which also a BDD-based implementation is evaluated.

Efficient symbolic state space exploration is also a well-studied topic in the field of model checking. An overview of concepts and techniques for BDD-based model checking is provided in Chaki and Gurfinkel (2018). A BDD-based algorithm for computation tree logic model checking is introduced in Burch et al. (1994). Several variable ordering heuristics for state space exploration of interacting finite state machines are evaluated in Aziz et al. (1994). In Cabodi et al. (1999) the efficiency of BDD-based operators is improved by partitioning the BDDs. Zero-suppressed BDDs are used in Minato (2001) to reduce computational effort of symbolic model checking in some applications.

We are not aware of existing works that study variable ordering heuristics specifically for supervisor synthesis, static edge ordering heuristics for supervisor synthesis (although dynamic edge ordering is investigated in Vahidi et al. (2006) and Fei et al. (2014)), or efficient enforcement of state-based requirements.

Structure

We first discuss preliminaries on symbolic supervisor synthesis in Section 5.2. BDD-based metrics to measure computational effort are introduced in Section 5.3. In the same section, these metrics are used to show the impact of the variable order and edge order on the computational effort of synthesis. In Sections 5.4 and 5.5 respectively a variable and an edge ordering heuristic algorithm are introduced, that aim to reduce the computational effort in synthesis. In Section 5.6 a method is introduced to efficiently enforce requirements in synthesis. Conclusions are provided in Section 5.7.

5.2 Symbolic supervisor synthesis

In this section we first introduce EFAs and their linearized version that can be symbolically encoded. Next, we discuss symbolic supervisor synthesis. Following, the encoding of a system in BDDs is considered. Finally, we introduce the relevant parts of the tool CIF, which we use for symbolic supervisor synthesis.

5.2.1 Automata

We consider an EFA A defined as 8 -tuple:

$$A = (L, V, \Sigma, T, L_0, V_0, L_m, V_m),$$

where L is a finite set of locations, V is a finite set of discrete variables (each with a finite domain), and Σ is a finite set of events, usually called alphabet. The alphabet is split into two disjoint subsets: Σ_c and Σ_u , representing controllable and uncontrollable events respectively. $L_0 \subseteq L$ is a set of possible initial locations, V_0 is an expression indicating possible initial values of all variables in V , $L_m \subseteq L$ is the set of marked locations, and V_m is an expression indicating marked values for variables V . T is a set of transitions where a transition t is defined as 5 -tuple:

$$t = (l_o, l_t, \sigma, \gamma, v),$$

where l_o and l_t are the origin and target location in L , σ is an event in Σ , γ is a guard expression, indicating for which variable values the transition can take place, and v is an update expression that indicates new values for the variables after the transition has occurred. In case in a transition the update is not specified for a variable, then its value remains the same when taking the transition. In case in a transition the guard is not specified, then the guard is assumed ‘true’, i.e., the transition can occur for any valuation of variables.

Essentially, locations can be modeled as variables, which we call *location pointer variables*, and transitions between locations can be modeled as guards and updates. Furthermore, expressions stated in an EFA can be encoded in (Boolean) predicates, that return true or false for a particular evaluation of variable values. Therefore, to simplify our explanations, and also stay consistent with the implementation of symbolic supervisor synthesis in CIF, we consider *Linearized Finite Automata (LFAs)*, where an LFA is defined as a 5 -tuple:

$$A_L = (X, \Sigma, E, X_0(X), X_m(X)),$$

in which X is a finite set of variables (which may contain a location pointer variable). A *state* is defined by a valuation over these variables. Σ is the alphabet. X_0 and X_m are predicates over variables from X that respectively represent the initial and marked

states. Note that for a predicate $P(X)$ we may simply write P when it is clear from context it is a predicate over variables X . E is the set of *edges*, with edge e defined as *triple*:

$$e = (\sigma, g(X), u(X, X^+)),$$

where σ is an event, g is a guard predicate, expressing from what states the event may occur, and u is an update predicate over current state variables X and *new state variables* $X^+ = \{x^+ | x \in X\}$, representing what state will be reached when the edge is taken from a particular current state. We assume $X \cap X^+ = \emptyset$.

Example 5.1 EFA and LFA

We consider the EFA of Figure 5.1. This EFA consists of two locations $L = \{l_0, l_1\}$ of which l_1 is marked: $L_m = \{l_1\}$, as indicated in Figure 5.1 by a double circle. The initial location $L_0 = \{l_0\}$ is indicated by the dangling incoming arrow. We have two Boolean variables named a and b . Both variables are initially set to *false*, as indicated by the expression next to the dangling incoming arrow. Events a_on and b_on can occur at l_0 , the value of a and b will then respectively update to *true*. These updates are denoted in Figure 5.1 by the keyword ‘do’. An edge with event label *continue* can be taken from origin location l_0 to target location l_1 . This can only happen if the guard $a = b$ evaluates to true. The guard is denoted by the keyword ‘when’. In case no update expression is given, the variables keep the same value after taking the transition. In case no guard expression is given, it is assumed true, i.e., the transition can occur for any values of the variables. All variable values are considered to be marked.

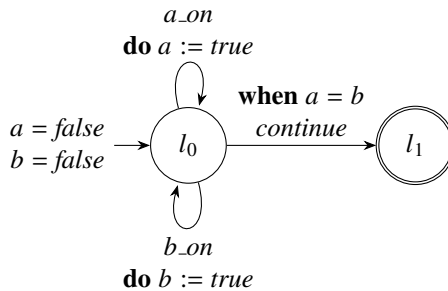


Figure 5.1: Example EFA.

The same model can be expressed as an LFA as follows: The set of variables is $\{l_s, a, b\}$, where l_s is the location pointer variable with domain $\{l_0, l_1\}$, used to encode the current location. The initial state predicate is $(l_s = l_0) \wedge \neg a \wedge \neg b$. The marked state predicate is $(l_s = l_1)$. There are three edges:

$$\begin{aligned}
& (a_on \quad , l_s = l_0 \quad , (l_s^+ = l_0) \wedge a^+ \wedge (b^+ = b)); \\
& (b_on \quad , l_s = l_0 \quad , (l_s^+ = l_0) \wedge (a^+ = a) \wedge b^+); \text{ and} \\
& (continue, (l_s = l_0) \wedge (a = b), (l_s^+ = l_1) \wedge (a^+ = a) \wedge (b^+ = b)).
\end{aligned}$$

To explain the notation, the first edge is labeled by event a_on , has guard $l_s = l_0$, so can only occur when the system is currently in location l_0 , and has update $(l_s^+ = l_0) \wedge a^+ \wedge (b^+ = b)$, indicating that after taking the edge, the location becomes l_0 , a becomes true, and b does not change value.

5.2.2 Symbolic supervisor synthesis

The purpose of applying supervisor synthesis is to generate a supervisor automaton such that the parallel composition between the plant automaton and supervisor is *safe*, *nonblocking*, *controllable*, and *maximally permissive* (Cassandras and Lafortune 2021). Safe means that the requirements are always satisfied. How requirements are specified, and what it exactly means to satisfy them, is discussed in more detail in Section 5.6. Nonblocking indicates that from every reachable state in the controlled system, a marked state can be reached. Controllable means that from every reachable state in the controlled system, when the plant can execute an uncontrollable event, this event can also be executed in the parallel composition between supervisor and plant. In other words, the supervisor does not stop any uncontrollable events from occurring. Maximally permissive says that these properties are ensured without disabling any events that do not strictly need to be disallowed.

In Algorithm 5.1 a supervisor synthesis algorithm is presented. This synthesis algorithm is based on the algorithm introduced by Ouedraogo et al. (2011), simplified by using an LFA instead of an EFA. In line 1 the requirements are applied by using algorithm `applyRequirements`. Again, application of requirements is discussed Section 5.6. For this preliminary section, it is only relevant to know that `applyRequirements` returns a predicate N that defines all states where the requirements are satisfied, a set of edges E_S which is the set of plant edges E with restricted guards of the controllable events such that the requirements are satisfied, and a predicate $X_{0,S}$ defining the initial states that satisfy the requirements, i.e., the initial states that are safe³.

So, `applyRequirements` outputs a predicate N for which the requirements are satisfied. However, this predicate might still allow blocking states (states that cannot reach a marked state). Algorithm 5.1 iteratively calculates nonblocking states N , followed by bad states B . The calculation to obtain N and B is done by means of a backward reachability search, given in Algorithm 5.2. This search is performed on the predicates by using the existential quantification operator (Bryant 1992; Lousberg et al. 2020). The bad states are removed from N . The removal of these states can induce

³ In case of multiple initial states, it is assumed that the supervisor can restrict in which of those states the system starts.

Algorithm 5.1 SS (Supervisor Synthesis)

Input: Plant LFA $A_L = (X, \Sigma, E, X_0, X_m)$, state exclusion predicates SX , state-event exclusion predicates EX **Output:** Supervisor LFA S

```

1:  $(N, E_S, X_{0,S}) = \text{applyRequirements}(SX, EX, E, X_0)$ 
2: repeat
3:    $N' = N$ 
4:    $N = \text{BRS}(N, E_S, X_m)$ 
5:    $B = \text{BRS}(\text{true}, \{(\sigma, g, u) \in E \mid \sigma \in \Sigma_u\}, \neg N)$ 
6:    $N = N \wedge \neg B$ 
7: until  $N = N'$ 
8: for all  $(\sigma, g, u) \in E_S$  with  $\sigma \in \Sigma_c$ 
9:    $g(X) = g(X) \wedge \exists_{X^+} [N(X^+) \wedge u(X, X^+)]$ 
10: end
11:  $S = (X, \Sigma, E_S, X_{0,S} \wedge N, X_m \wedge N)$ 

```

Algorithm 5.2 BRS (Backward Reachability Search)

Input: Restriction predicate $P_R(X)$, edges E , start predicate $P(X)$ **Output:** Coreachable predicate $P'(X)$

```

1: repeat
2:    $P'(X) = P(X)$ 
3:    $P(X) = P_R(X) \wedge (P(X) \vee \bigvee_{(\sigma, g, u) \in E} \exists_{X^+} [P(X^+) \wedge g(X) \wedge u(X, X^+)])$ 
4: until  $P(X) = P'(X)$ 

```

other states to become blocking. Therefore, the algorithm repeats these steps until a fixpoint is reached, i.e., no further bad states get removed. Next, the guards of the edges labeled by controllable events are strengthened such that the guard can only be true when the nonblocking predicate is true for the state that is reached after taking the edge. Here, notation $N(X^+)$ denotes predicate $N(X)$ in which each current state variable $x \in X$ is substituted by its new state counterpart x^+ . Finally, the supervisor LFA is constructed. The conjunction is taken between the initial state predicate and the nonblocking predicate, so that the supervised system is only initialized in nonblocking states. The supervised system will remain in nonblocking states, as the strengthened guards prevent transitions from nonblocking to bad states.

5.2.3 Binary Decision Diagrams

A Binary Decision Diagram (BDD) (Akers 1978) is a data structure that is used to represent Boolean functions and predicates, and can be used to represent and perform calculations on an LFA. BDDs are directed acyclic graphs that consist out of two types of nodes: decision- and terminal nodes. Each decision node is labeled by a Boolean variable b and has two edges leading to child nodes, one edge labeled true and the other false. When evaluating b to true or false we take the respective edge. At the leaves of the BDD are terminal nodes, that are labeled by true or false, indicating the final result of evaluating the BDD.

When referring to BDDs in this chapter, we implicitly always mean *reduced ordered* BDDs (Bryant 1992). This type of BDD imposes some additional restrictions such that the BDD is minimal in the number of decision nodes and canonical for a given order of the variables. This order is strictly imposed over all the variables in the BDD and is called the *variable order*. A variable order is denoted as $<$, where $b_1 < b_2$ indicates that decision node b_1 is placed closer to the root node than b_2 .

The variable order can influence the number of decision nodes required to encode a Boolean expression, see Figure 5.2 for an example. Visually, we represent true edges by solid lines and false edges by dashed lines. The size of a BDD is defined by the number of decision nodes and in worst case this size can be exponential in the number of Boolean variables (Bryant 1992).

In our work, when we mention variable order, we refer to an order of the LFA variables. The variable order corresponds to an order of Boolean variables by which the BDD is ordered. We explain how the LFA variables are encoded as Boolean variables, and how an LFA variable order corresponds to a variable order of Boolean variables in Example 5.2.

Example 5.2 BDD encoding

We consider an LFA with variables y and z . y is an integer that can take values $\{0, 1, 2\}$ and z is an integer that can take values $\{0, 1\}$. y can be encoded using two Boolean variables: $b_{y,0}$ and $b_{y,1}$; z can be encoded using a single Boolean variable $b_{z,0}$. Note that for every current state variable, there is also a new state variable. So, there is also an integer y^+ corresponding to Boolean variables $b_{y^+,0}$ and $b_{y^+,1}$, and integer z^+ corresponding to Boolean variable $b_{z^+,0}$. Let us assume the LFA variables are ordered by $y < z$. Then, this corresponds to the following variable order of the Boolean variables: $b_{y,0} < b_{y^+,0} < b_{y,1} < b_{y^+,1} < b_{z,0} < b_{z^+,0}$. So, in the order of Boolean variables, a Boolean variable corresponding to a current state variable is always immediately succeeded by the respective Boolean variable corresponding to the new state variable (using default settings in CIF).

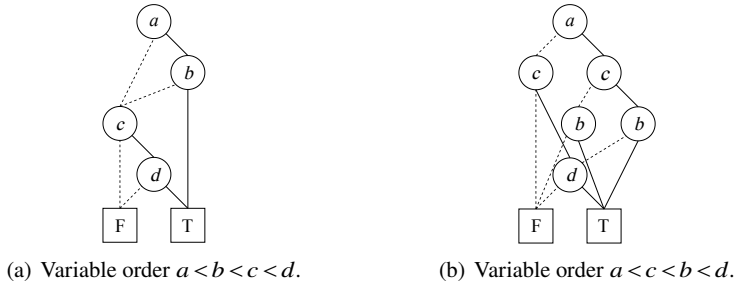


Figure 5.2: Two BDDs representing $(a \wedge b) \vee (c \wedge d)$ for different variable orders.

In this work we study how to effectively use BDDs in symbolic supervisor synthesis to reduce computational effort.

5.2.4 CIF

There are several tools that allow modeling of plants and requirements with the ability to synthesize a supervisor. Of the tools considered in Reniers and van de Mortel-Fronczak (2018), the tools Supremica (Malik et al. 2017) and CIF (van Beek et al. 2014) allow for the use of EFAs and base their EFAs supervisor synthesis algorithm on the use of BDDs. The syntheses in this chapter are performed using the EFAs supervisor synthesis tool of CIF. CIF has been used to synthesize supervisors for industrial sized systems (Theunissen et al. 2014; Reijnen et al. 2017, 2018a; Loose et al. 2018; Korssen et al. 2018; Reijnen et al. 2020).

CIF is part of the ESCET project, an Eclipse open-source project since 2020 (Fokkink et al. 2023). ESCET provides a model-based approach and toolkit for developing supervisory controllers. It supports synthesis-based engineering of supervisory controllers for discrete-event systems, combining model-based engineering with computer-aided design to automatically generate correct-by-construction controllers through supervisor synthesis.

5.3 Evaluating computational effort in symbolic supervisor synthesis

The performance of algorithms is typically judged by their space- and time complexity (Meinel and Theobald 1998). We use the metrics *peak used BDD nodes* and *BDD operation count* introduced in Thuijsman et al. (2019) to quantitatively express the

space- and time effort required for supervisor synthesis. These metrics have some advantages over conventional metrics such as peak random access memory and wall clock time. First, they are deterministic: performing a supervisor synthesis twice with the same input and algorithm configuration will give the exact same result. This determinism also holds when doing the synthesis on two different platforms, even if one is a supercomputer and the other is a personal computer. As a result, it becomes easier to compare results from different measurements or publications. Second, there is no overhead in the measurement, loaded-in Java classes and other computer processes will not influence the measurement. After introducing the BDD-based metrics, we relate them to the wall-clock time and peak memory usage of a synthesis and worst case state space size of the model in Section 5.3.3.

We distinguish *complexity* from *effort*. Complexity regards classes of problems, and defines the generic trend of the (space/time) resources a computation requires for inputs of different sizes, often expressed using ‘Big O’ notation (Knuth 1976). Effort specifies the amount of resources used for one particular computation, where the complete input is considered rather than only its size. This input includes algorithm configuration settings and, in our case, variable and edge order.

5.3.1 Peak used BDD nodes

During symbolic supervisor synthesis, the number of BDD nodes used to describe the predicates generally fluctuates. Since *reduced* ordered BDDs are used, which are minimal representations, the used BDD nodes is the minimal amount of BDD nodes required to represent the predicates at that point during the computation. The space effort can be measured by the peak (maximal) number of BDD nodes used during synthesis (Meinel and Theobald 1998; Vahidi et al. 2006).

In CIF, BDD nodes are stored in a hash table. Each new node is allocated to an entry in the hash table. Once the hash table reaches a certain fill rate, garbage collection is employed to free no longer used entries. We only count the *used* BDD nodes, i.e., hash table entries that still contain relevant information for the BDDs that are still in use. Garbage collection is performed by means of a standard mark-and-sweep algorithm. Functions from the implementation of this algorithm in the JavaBDD library⁴ are reused to count the BDD nodes that are in use. This measurement is performed each time just before a BDD reference is deleted, which causes used nodes to become unused. Thereby ensuring the exact peak value of used nodes is found. For a more detailed explanation on BDD node references, used nodes, and unused (dead) nodes we refer to Somenzi (1999).

Peak used BDD nodes is a reproducible metric: Performing a supervisor synthesis twice with the same input yields exactly the same peak used BDD nodes.

⁴ The JavaBDD library is available at <https://javabdd.sourceforge.net>.

5.3.2 BDD operation count

The time effort can be expressed in the number of steps/operations during a computation (Meinel and Theobald 1998). As supervisor synthesis is done by performing operations on BDDs, we use BDD operation count to express the time effort of performing supervisor synthesis. Since BDD operations (such as ‘and’ and ‘or’) are implemented as functions that employ structural recursion on BDD nodes, the number of invocations of such functions can be used to express time effort. Since the functions are deterministic, the results are reproducible.

Generally, these functions consist of three parts. First, a few checks are performed to see whether the requested calculation is a terminal case. Second, if it is a non-terminal case, it is checked whether the calculation has already been performed, and is still in the cache. Note that we do not mean hardware cache here, but a table actively storing results of previous calculations. If both previous cases do not apply, the function performs recursive expansion over the child nodes. Each time this recursive expansion is performed, i.e., when operations are applied on the BDD, we increment the BDD operation count. For more details about terminal cases, cache lookup and recursive expansion over child nodes we refer to Somenzi (1999).

5.3.3 Relevance of metrics

In order to compare the BDD-based metrics to conventional metrics, e.g., wall clock time, memory usage, and state space sizes, we perform a number of supervisor syntheses and extract these metrics. The data presented in this chapter is acquired by performing supervisor syntheses to the models shown in Table 5.1. The models are selected to have a wide range of sizes. Table 5.1 shows, and is sorted by, the worst case state space size of the uncontrolled plant for each model, which is the product of the number of the locations and variable domain sizes of each EFA in the model. E.g., the EFA given in Example 5.1 has two locations and two variables with each a domain of two, so its worst case state space is $2 \cdot 2 \cdot 2 = 8$. The first two models have a worst case state space of a single state because their plant models only contain EFAs with a single state. The requirement specifications of these models contain automata with more than one state.

For a supervisor synthesis of the Waterway lock model, Figure 5.3 shows how the number of used BDD nodes evolves, as BDD operations are performed during synthesis. Intuitively, the horizontal axis represents the ever-increasing number of operations performed as synthesis progresses, and the vertical axis represents the fluctuating memory usage. The metrics presented in this chapter are the maxima along both axes in this plot: the peak used BDD nodes and the final BDD operation count.

Figures 5.4(a) and 5.4(b) show how peak random access memory and wall clock time respectively relate to peak used BDD nodes and BDD operation count. A supervisor was synthesized for each model of Table 5.1 for 100 pairs of random variable- and edge

Table 5.1: Benchmark models.

Name		Worst case state space size
Robotic swarm aggregation	(Lopes et al. 2016)	$1.0 \cdot 10^0$
Robotic swarm clustering	(Lopes et al. 2016)	$1.0 \cdot 10^0$
Robotic swarm segregation	(Lopes et al. 2016)	$6.4 \cdot 10^1$
Robotic swarm formation	(Lopes et al. 2016)	$8.0 \cdot 10^1$
Multi agent formation	(Cai and Wonham 2014)	$1.0 \cdot 10^3$
Automatic guided vehicles	(Wonham and Cai 2019)	$3.1 \cdot 10^3$
Ball sorting system	(Čengić and Åkesson 2008)	$7.4 \cdot 10^4$
Theme park vehicles	(Forschelen et al. 2012)	$2.9 \cdot 10^5$
Cluster tool	(Su et al. 2010)	$2.7 \cdot 10^8$
Production cell	(Feng et al. 2008)	$7.5 \cdot 10^8$
Modified cat and mouse tower ($n=3, k=1$)	(Thuijsman et al. 2021)	$1.1 \cdot 10^9$
Advanced driver assistance system (Korssen et al. 2018)		$3.4 \cdot 10^9$
Cat and mouse tower ($n=3, k=2$)	*	$2.1 \cdot 10^{14}$
Lithography machine initialization	(Vos 2020)	$1.8 \cdot 10^{16}$
Bridge	(Reijnen et al. 2018b)	$2.8 \cdot 10^{27}$
FESTO production line	(Reijnen et al. 2018a)	$1.3 \cdot 10^{28}$
Waterway lock	(Reijnen et al. 2017)	$6.0 \cdot 10^{32}$

* This model first appears in literature in Ma and Wonham (2008), Miremadi et al. (2008), and Moor et al. (2008). See Section 3.6 for more details.

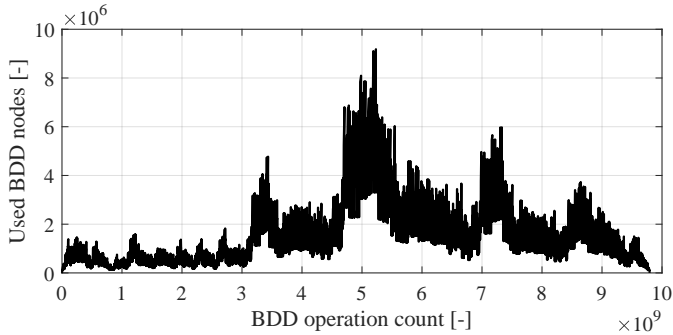


Figure 5.3: Evolution of used BDD nodes during synthesis.

orders. Note that these variable orders have been re-ordered by heuristic algorithms FORCE and Sliding Window (SW), which we discuss later, to obtain the variable order that synthesis actually uses. The measurements were performed in sequence using two Intel Xeon Gold 6226 processors clocked at 2.70 GHz, operating on Linux. The measurements for random access memory and wall clock time were done separately from the measurements of the BDD-based metrics to avoid them from interfering.

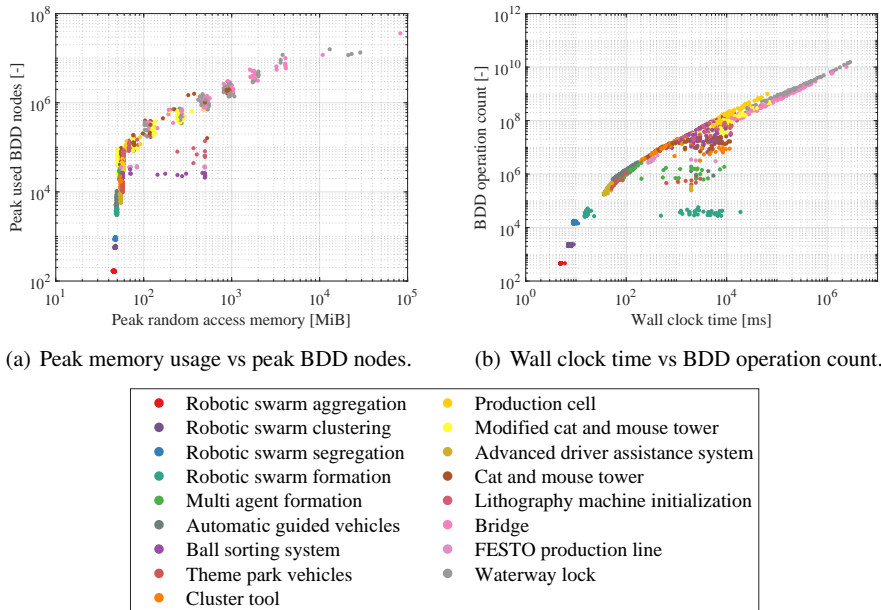


Figure 5.4: BDD-based metrics against conventional metrics.

For small models, peak random access memory cannot indicate a difference in synthesis effort, as all results of the small models are grouped around 60 MiB. For some of the smaller and medium-sized models there is a significant amount of noise in the measured wall clock time and random access memory. For wall clock time, this noise typically originates from delays in IO procedures (writing the output file). For memory, the noise originates from loaded in classes and the practically random intervals at which Java performs garbage collection.

For larger computations, a linear relation is visible between wall clock time and BDD operation count. The threshold at which this relation starts, and its slope, are dependent on the used hardware. The scattering that is seen for larger computations in Figure 5.4(a) is a result of the manner in which the BDD space allocation takes place: when the current table is full, it gets doubled in size, the new free entries in this table will have an influence on the memory, but are not measured when counting the used BDD nodes. Also, when performing computations that require more memory, the Java Virtual Machine will perform garbage collection in the background to free memory. For separate measurements this will happen at different times, which impacts the peak random access memory, not the amount of used BDD nodes. Note that for the measurements in Figure 5.4 additional garbage collection is performed before every measurement to achieve a consistent situation between measurements.

An advantage of wall clock time and peak random access memory is that a user performing supervisor synthesis is more likely to be familiar with these metrics. It gives

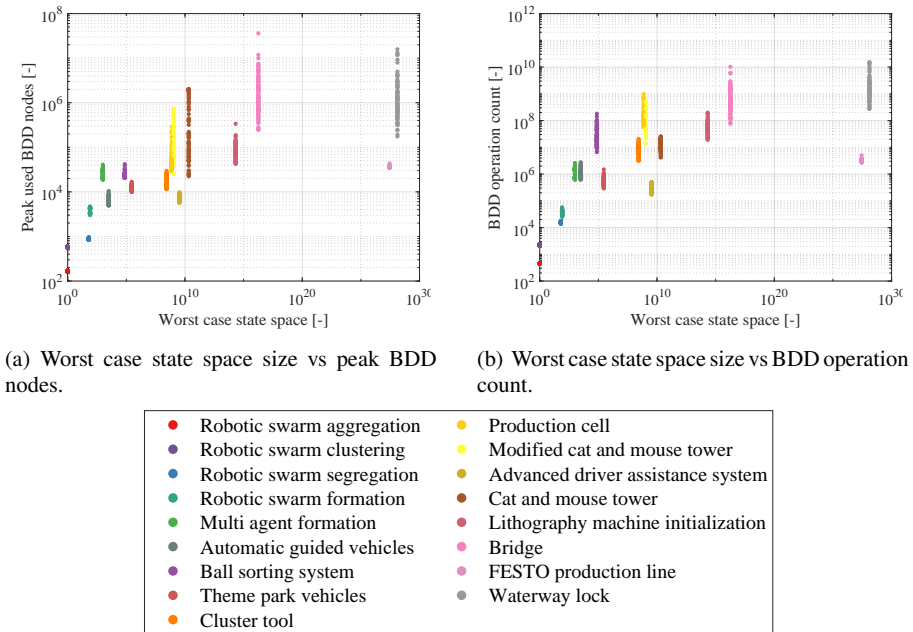


Figure 5.5: BDD-based metrics against conventional metrics.

a better idea whether their computer is able to perform the synthesis in an acceptable amount of time given the available memory. However, opposed to the BDD-based metrics, wall clock time and peak random access memory are not deterministic, so they will yield different results for every synthesis run. Their results are influenced by aspects including loaded classes, garbage collection, and IO operations. The BDD-based metrics enable a distinction in effort for the actual synthesis portion of the computation. Also, the BDD-based metrics are platform dependent. Particularly the wall clock time of some synthesis will be influenced by the used hardware, making it difficult to compare results.

Worst case state space of the uncontrolled system is also frequently used to indicate (expected) synthesis effort. The advantage of using worst case state space size of the uncontrolled system over BDD-based metrics to indicate the synthesis effort, is that no supervisor synthesis or reachability computations are required to calculate this number. Figures 5.5(a) and 5.5(b) show how this state space size relates to the BDD-based metrics. There is a general trend of a larger state space size suggesting more supervisor synthesis effort, but it is not a very accurate indicator. Note that there are multiple ways to indicate state space sizes, some also taking the product with requirement automata, but we found that also in those cases the state space size cannot accurately indicate the expected supervisor synthesis effort.

5.3.4 Impact of variable- and edge order on computational effort

We presented two metrics that indicate the computational effort of a supervisor synthesis. However, we have to be careful when making conclusions based on this synthesis effort. The variable- and edge order have an influence on the results. This can also be seen in Figures 5.4 and 5.5, where the results per model are scattered due to using different variable- and edge orders. Recall that since the BDD-based metrics are deterministic, re-performing a synthesis with the same variable- and edge orders would provide the exact same result.

It is well known that the variable order has a notable impact on the BDD size, and consequently on the computational effort. Therefore, the variable ordering heuristics FORCE and SW (Aloul et al. 2003) are implemented in CIF (already prior to this work). FORCE is supplied with a variable order and reorders it to group variables together that have high interaction, meaning they often appear together in guards and updates. Note that this algorithm finds a local optimum: initializing it with different orders, might give different resulting variable orders. SW starts from a variable order, and “slides a window” across the variables to locally optimize that part of the order. In this work, always a (default) window size of 4 is used. These heuristics can be sequenced. We denote *FORCE+SW* to indicate that first FORCE is applied to some initial variable order, and SW is performed on the variable order computed by FORCE, to produce the variable order used in synthesis.

In previous work (Thuijsman et al. 2019; Lousberg et al. 2020; Thuijsman et al. 2021) the conditions when FORCE and SW considered two variables to be related was different from the current implementation. For further details we refer to the CIF documentation. The current implementation uses the variable relations as discussed in Section 5.4. Hence, we repeat the experiments of Thuijsman et al. (2019); Lousberg et al. (2020) and Thuijsman et al. (2021) using the same variable dependencies for all variable ordering heuristics, that we discuss below, such that we can accurately compare their efficiency.

We investigate to what extent the edge order and initial variable order influence the supervisor synthesis effort. For each model in Table 5.1, a supervisor has been synthesized for all combinations of 100 random edge orders and 100 random initial variable orders. These initial variable orders are re-ordered by *FORCE+SW* to produce the variable order used in synthesis. The effort of performing each synthesis is shown in Figure 5.6.

It can be seen that there are major differences in computational effort by using different orders. For the Waterway lock model, the highest peak used BDD nodes is 658 times larger than the lowest peak used BDD nodes. For BDD operations this factor is 338. This is purely a result of changing the edge orders and initial variable orders: all other algorithm configurations were the same for all measurements.

Figure 5.6 also shows that measuring both peak used BDD nodes and BDD operation count is relevant. It would be difficult to distinguish the computational effort between some of the syntheses if only one of the metrics was used. For example, if we only measured the BDD operation count, we would not see much difference for the efforts

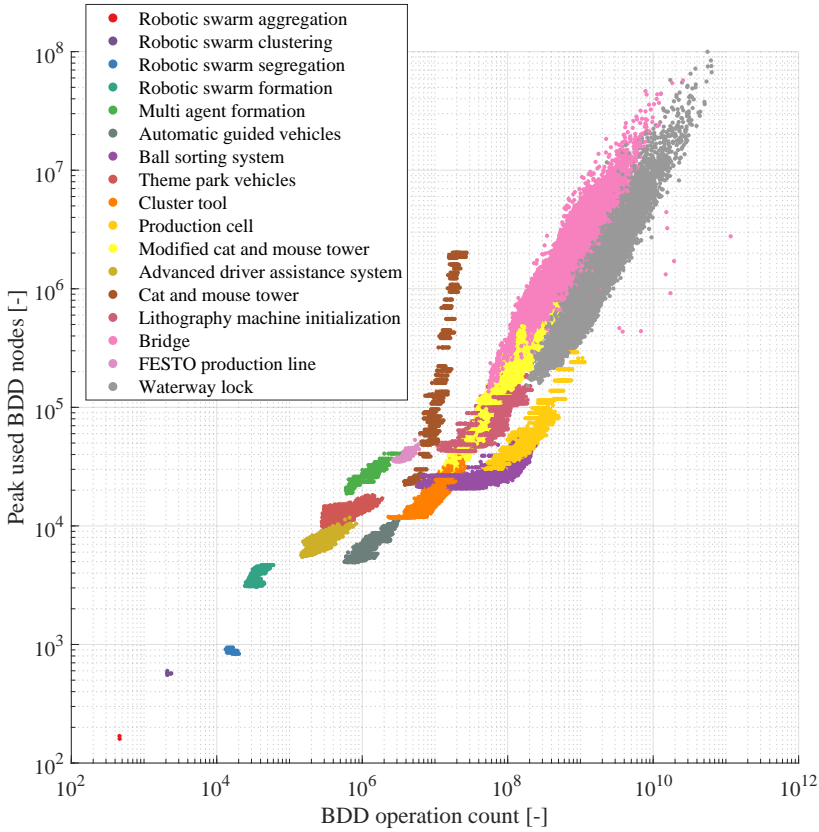


Figure 5.6: Supervisor synthesis effort for all combinations of 100 edge- and 100 variable orders for each model.

of synthesis for Cat and mouse tower. If we only measured peak used BDD nodes, we would not see much difference for the effort of synthesis for Ball sorting system. Measuring both metrics enables us to differentiate between the efforts of synthesis for each model based on the various initial variable orders and edge orders.

Figure 5.7 shows the peak used BDD nodes for all syntheses of the Theme park vehicles model. Darker squares indicate a higher amount of peak used BDD nodes. All variable- and edge orders were given an index. A row shows the peak used BDD nodes of all syntheses that were performed with the same edge order and varying variable orders, and a column shows the same for a fixed variable order with varying edge orders. In Figure 5.7, we see rows and columns where the elements are similarly colored, indicating that variable order and edge order both have a reasonable impact on the peak used BDD nodes for this particular model. There are other models where only the elements in columns are similarly colored, indicating that the variable order mainly

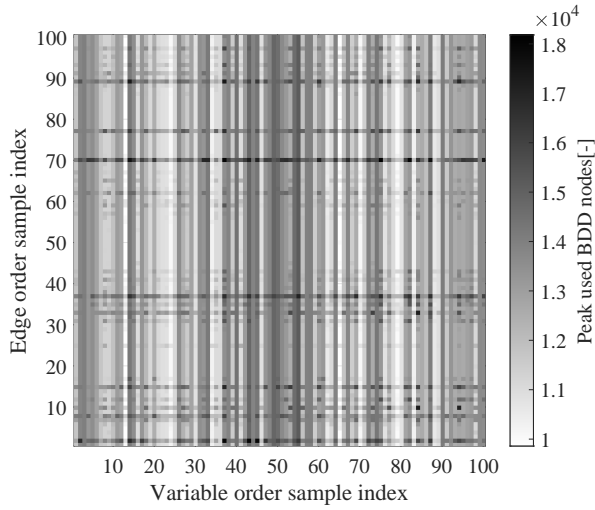


Figure 5.7: Peak used BDD nodes for all supervisor syntheses of the Theme park vehicles model.

influences their synthesis effort. We observe similar results for the BDD operation count.

Figure 5.7, along with analyzing the same plot for other models, shows that a relatively poor/good variable order generally performs relatively poor/good for all edge orders and vice versa. This means the variable order and edge order can be improved individually, which is what we respectively focus on in Sections 5.4 and 5.5.

If we define the peak used BDD nodes for a certain model as a deterministic function $f(o_{v,i}, o_{e,j})$, where $o_{v,i}$ is the i 'th sample random variable order and $o_{e,j}$ the j 'th sample random edge order, the global sample mean (Montgomery and Runger 2018) of the peak used BDD nodes $\mu_G(f)$ is given by Equation 5.1:

$$\mu_G(f) = \frac{1}{N \cdot M} \sum_{i=1}^N \sum_{j=1}^M f(o_{v,i}, o_{e,j}), \quad (5.1)$$

where N and M respectively are the total number of sampled variable- and edge orders. For our experiment, $N = M = 100$ for each model.

The global (unbiased) sample variance (Montgomery and Runger 2018) of the peak used BDD nodes $\sigma_G^2(f)$ is given by Equation 5.2:

$$\sigma_G^2(f) = \frac{1}{N \cdot M - 1} \sum_{i=1}^N \sum_{j=1}^M (f(o_{v,i}, o_{e,j}) - \mu_G(f))^2. \quad (5.2)$$

The sample variance $\sigma_{v,i}^2(f)$ of the peak used BDD nodes for the edge orders tested with a particular variable order $o_{v,i}$, is given by Equation 5.3:

$$\sigma_{v,i}^2(f) = \frac{1}{M-1} \sum_{j=1}^M (f(o_{v,i}, o_{e,j}) - \mu_{v,i}(f))^2, \quad (5.3)$$

where $\mu_{v,i}(f) = \frac{1}{M} \sum_{j=1}^M f(o_{v,i}, o_{e,j})$ is the mean peak used BDD nodes of the edge orders tested with variable order $o_{v,i}$. The mean sample variance for fixed variable orders $\overline{\sigma_v^2}(f)$ is computed by Equation 5.4:

$$\overline{\sigma_v^2}(f) = \frac{1}{N} \sum_{i=1}^N \sigma_{v,i}^2(f). \quad (5.4)$$

Equations 5.3 and 5.4 can analogously be applied to compute the sample variance of peak used BDD nodes for variable orders tested with particular edge orders $\sigma_{e,j}^2(f)$, and the mean sample variance for fixed edge orders $\overline{\sigma_e^2}(f)$. Likewise, we can define a function $g(o_{v,i}, o_{e,j})$ for the BDD operation count of a model and apply above computations to this.

When relating these characteristics to what we see in Figure 5.7, a low mean sample variance for fixed variable orders $\overline{\sigma_v^2}(f)$ would indicate a similar amount of peak used BDD nodes for a given variable order. This would be visible in Figure 5.7, as elements located in the same column would be similarly colored. This would indicate that the variable order mainly influences the peak used BDD nodes, and the edge order has little influence.

For each model, the global sample mean μ_G , global sample variance σ_G^2 , mean sample variance for fixed variable orders $\overline{\sigma_v^2}$ and mean sample variance for fixed edge orders $\overline{\sigma_e^2}$ are given for peak used BDD nodes (f) and BDD operation count (g) in Table 5.2. For most of the models, the mean sample variance for fixed variable orders is smaller than the mean sample variance for fixed edge orders. This indicates that the variable order has a larger influence on the supervisor synthesis effort than the edge order. The effect of the variable order is particularly notable, when considering that for these experiments FORCE+SW is being applied to the variable order (to compute the variable order that synthesis is performed with). The variance is even higher when FORCE+SW is not applied (Lousberg et al. 2020). However, the mean variance for fixed variable orders is large enough that the edge order is still of considerable influence to the supervisor synthesis effort.

Models that require a relatively large amount of supervisor synthesis effort, also have a relatively large variance in effort. This would also be observed if we were to normalize to the mean values of the models, i.e., σ^2/μ or σ/μ . This indicates that applying a good variable- and edge order becomes more beneficial when considering models that require more supervisor synthesis effort.

Table 5.2: Sample means and variances of all models.

Name	$\mu_G(f)$	$\sigma_G^2(f)$	$\overline{\sigma_v^2(f)}$	$\overline{\sigma_e^2(f)}$	$\mu_G(g)$	$\sigma_G^2(g)$	$\overline{\sigma_v^2(g)}$	$\overline{\sigma_e^2(g)}$
Swarm aggr.	$1.6 \cdot 10^2$	$2.0 \cdot 10^1$	0.0	$2.0 \cdot 10^1$	$4.6 \cdot 10^2$	1.0	0.0	1.0
Swarm clust.	$5.7 \cdot 10^2$	$1.6 \cdot 10^2$	0.0	$1.6 \cdot 10^2$	$2.2 \cdot 10^3$	$1.7 \cdot 10^4$	0.0	$1.7 \cdot 10^4$
Swarm segr.	$8.8 \cdot 10^2$	$9.7 \cdot 10^2$	0.0	$9.8 \cdot 10^2$	$1.6 \cdot 10^4$	$8.9 \cdot 10^5$	$5.6 \cdot 10^5$	$4.7 \cdot 10^5$
Swarm form.	$3.9 \cdot 10^3$	$2.6 \cdot 10^5$	0.0	$2.6 \cdot 10^5$	$3.6 \cdot 10^4$	$4.5 \cdot 10^7$	$4.6 \cdot 10^6$	$4.2 \cdot 10^7$
Multi agent form.	$2.6 \cdot 10^4$	$2.3 \cdot 10^7$	0.0	$2.3 \cdot 10^7$	$1.2 \cdot 10^6$	$2.0 \cdot 10^{11}$	$5.8 \cdot 10^9$	$2.0 \cdot 10^{11}$
Autom. guid. vehic.	$6.0 \cdot 10^3$	$1.5 \cdot 10^6$	$7.1 \cdot 10^3$	$1.5 \cdot 10^6$	$1.1 \cdot 10^6$	$1.7 \cdot 10^{11}$	$1.4 \cdot 10^{10}$	$1.6 \cdot 10^{11}$
Ball sort. sys.	$2.4 \cdot 10^4$	$9.3 \cdot 10^6$	$8.4 \cdot 10^5$	$9.3 \cdot 10^6$	$3.4 \cdot 10^7$	$7.6 \cdot 10^{14}$	$1.5 \cdot 10^{14}$	$6.7 \cdot 10^{14}$
Theme park veh.	$1.3 \cdot 10^4$	$2.0 \cdot 10^6$	$3.2 \cdot 10^5$	$1.8 \cdot 10^6$	$5.7 \cdot 10^5$	$3.8 \cdot 10^{10}$	$2.6 \cdot 10^{10}$	$1.7 \cdot 10^{10}$
Cluster tool	$1.7 \cdot 10^4$	$2.1 \cdot 10^7$	$2.2 \cdot 10^6$	$2.0 \cdot 10^7$	$8.5 \cdot 10^6$	$1.3 \cdot 10^{13}$	$1.2 \cdot 10^{12}$	$1.2 \cdot 10^{13}$
Prod. cell	$5.9 \cdot 10^4$	$1.9 \cdot 10^9$	$3.3 \cdot 10^6$	$1.9 \cdot 10^9$	$1.9 \cdot 10^8$	$2.4 \cdot 10^{16}$	$4.9 \cdot 10^{14}$	$2.4 \cdot 10^{16}$
Modif. CMT	$2.9 \cdot 10^5$	$4.5 \cdot 10^{10}$	$4.3 \cdot 10^8$	$4.5 \cdot 10^{10}$	$2.0 \cdot 10^8$	$2.5 \cdot 10^{16}$	$2.8 \cdot 10^{14}$	$2.5 \cdot 10^{16}$
Adv. driv. asst.	$7.4 \cdot 10^3$	$1.1 \cdot 10^6$	$2.6 \cdot 10^5$	$8.7 \cdot 10^5$	$2.8 \cdot 10^5$	$6.7 \cdot 10^9$	$4.6 \cdot 10^9$	$2.9 \cdot 10^9$
CMT	$7.3 \cdot 10^5$	$6.3 \cdot 10^{11}$	$4.1 \cdot 10^3$	$6.3 \cdot 10^{11}$	$1.4 \cdot 10^7$	$3.1 \cdot 10^{13}$	$4.8 \cdot 10^{11}$	$3.1 \cdot 10^{13}$
Lith. mach. init.	$8.5 \cdot 10^4$	$1.6 \cdot 10^9$	$2.1 \cdot 10^6$	$1.6 \cdot 10^9$	$6.6 \cdot 10^7$	$1.2 \cdot 10^{15}$	$8.4 \cdot 10^{13}$	$1.1 \cdot 10^{15}$
Bridge	$2.3 \cdot 10^6$	$8.7 \cdot 10^{12}$	$5.2 \cdot 10^{12}$	$6.5 \cdot 10^{12}$	$8.9 \cdot 10^8$	$2.5 \cdot 10^{18}$	$2.1 \cdot 10^{18}$	$2.2 \cdot 10^{18}$
FESTO	$3.6 \cdot 10^4$	$1.1 \cdot 10^6$	$5.1 \cdot 10^5$	$1.0 \cdot 10^6$	$3.2 \cdot 10^6$	$9.5 \cdot 10^{10}$	$3.7 \cdot 10^{10}$	$7.9 \cdot 10^{10}$
Lock	$2.3 \cdot 10^6$	$1.9 \cdot 10^{13}$	$1.0 \cdot 10^{13}$	$1.8 \cdot 10^{13}$	$2.5 \cdot 10^9$	$1.5 \cdot 10^{19}$	$5.7 \cdot 10^{18}$	$1.3 \cdot 10^{19}$

5.4 DCSH variable ordering heuristic

In Section 5.3.4 we have touched on the extent in which edge and variable order influence the computational effort. There is a large variance in the computational effort of synthesis as a result of the initial variable order, even if FORCE+SW is applied. In this section, we introduce a heuristic algorithm named *DSM-based Cuthill-McKee-Sloan variable ordering Heuristic* (DCSH) to find a variable order that reduces the computational effort required for symbolic supervisor synthesis compared to current implementation (FORCE+SW). This heuristic is based on two matrix ordering heuristics from Cuthill and McKee (1969) and Sloan (1989), that are used to minimize the *Weighted Event Span* (WES) (Siminiceanu and Ciardo 2006). These matrix reordering heuristics are applied to a Dependency Structure Matrix (DSM) that stores the number of times BDD-variables appear together in transition relations, to find a new variable order. It is shown in Meijer and van de Pol (2016) that these heuristics are able to reduce the WES, and thereby the computational effort for symbolic model checking. Heuristics are used, since directly minimizing the WES is an NP-complete problem (Siminiceanu and Ciardo 2006).

5.4.1 Transition relation, variable order, and computational effort

When studying the evolution of BDDs during synthesis, most computational effort is performed during the reachability searches (Algorithm 5.2). Specifically, during the existential quantification operation in line 3 of Algorithm 5.2. Because this operation is applied many times, the guard and update predicates are placed in a single predicate, which is the *transition relation*: $T_e(X, X^+) = g_e(X) \wedge u_e(X, X^+)$, where there is a transition relation T_e for each edge $e = (\sigma, g_e(X), u_e(X, X^+))$.

Existential quantification over the transition relations is frequently applied during synthesis. This operation can be executed by first computing $P(X^+) \wedge T_e(X, X^+)$ and then quantifying over X^+ . However, this results in a large intermediate result of $P(X^+) \wedge T_e(X, X^+)$. Therefore, both the conjunction and existential quantification are computed in a single recursive pass over $P(X^+)$ and $T_e(X, X^+)$ by utilizing the relational product operation (Burch et al. 1994). This operation prevents computing the entire BDD $P(X^+) \wedge T_e(X, X^+)$ and quantifies early over X^+ , thereby reducing memory usage and number of required operations. Nevertheless, computing the relational product is known to be an expensive computation (Burch et al. 1994).

So, we frequently apply computationally expensive operations to BDDs that represent predicate $T_e(X, X^+)$. We say sets of variables are *strongly related* if they appear together in many transitions. BDDs are overall small if strongly related BDD-variables are placed near each other in the variable order (Minato 1996; Somenzi 1999). If we keep variables of each transition relation near each other in the variable order, it is likely that the resulting BDDs representing the (nonblocking and bad-state) predicates are kept small during synthesis, leading to reduced computational effort.

5.4.2 Dependency Structure Matrix reordering

A DSM is a square $n \times n$ matrix representing dependencies between n aspects of a system or model (Browning 2016). We capture variables of the *LFA* along the rows and columns where each index represents a single variable. In our use, the variables are always ordered the same along the row and column axis. In this chapter we utilize static Numerical DSMs (NDSMs). The off-diagonal elements can be non-negative integers, where the value indicates the number of times the respective variables appear together in a predicate expression T_e of an edge in the *LFA*. In our use, the diagonal elements are always zero. Furthermore, all dependencies in the NDSM are regarded as undirected, thus providing a symmetric matrix. Subsequently, the NDSM is manipulated by two matrix ordering heuristic algorithms that reorder the row and column indices such that non-zero values are placed towards the diagonal. The order in which the variables appear along the rows/columns is used as variable order for synthesis. Essentially, we are creating a variable order such that variables that often appear together in T_e , are placed near each other in the variable order.

Before synthesis we extract the variables that appear in predicate expression T_e for each edge $e \in E$. For all occurrences of pairs of variables per T_e we increment the element in the NDSM by one, thus a higher value indicates a stronger dependency between the variables. The increment is executed for both combinations of the pair such that the resulting NDSM is symmetric.

Figure 5.8 shows an example of an NDSM before and after reordering. The before image has the variables ordered alphabetically, which is the default initial variable order in CIF. The variables are reordered, by applying the method we discuss next. In the after image we observe that variables that are closely related are clustered together.

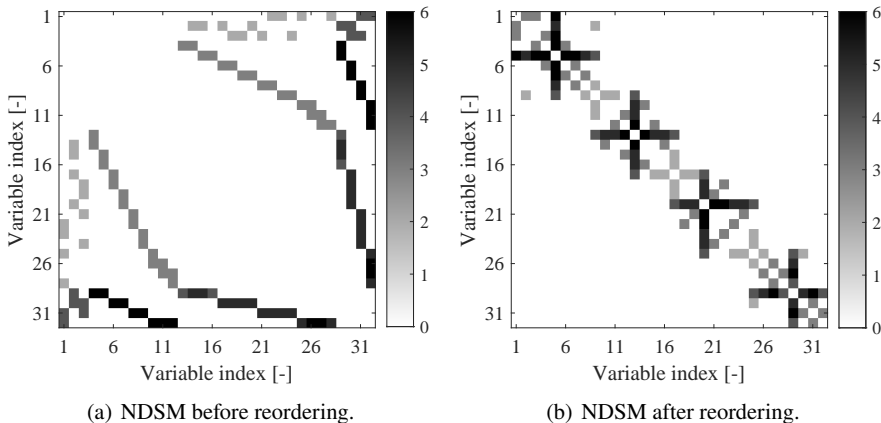


Figure 5.8: NDSM before and after reordering of the Cluster tool model.

The use of DSMs in supervisory control theory is not new. In Goorden et al. (2020) DSMs are used to find clusters of highly interactive components for the purpose of applying multilevel synthesis. However, we are not looking for clusters, but interested in reordering the row and column indices such that higher valued elements are placed as close as possible towards the diagonal relative to lower valued elements. By finding such an order, we also find an order where variables that often appear together in transition relations are placed near each other.

In practice, the NDSMs constructed in our approach are sparse. We utilize existing variable ordering heuristics, that have been designed for bandwidth, profile, and/or wavefront reduction of symmetric sparse matrices. For an elaboration on these metrics we refer to Cuthill and McKee (1969) for bandwidth and Sloan (1989) for profile and wavefront. By minimizing any of these metrics, an order is achieved for which relatively low computational effort is expected. The effective use of these heuristics for static variable order optimization for BDDs is shown in Meijer and van de Pol (2016), where several bandwidth, profile and wavefront reducing node ordering heuristics have been compared. These heuristics apply a reordering to the adjacency graph that can directly be extracted from an NDSM. As we utilize an NDSM we append the graph's edges by weights resulting in a *weighted* adjacency graph. For an NDSM with row index i and column index j , we denote elements by $\eta_{i,j}$. For each row i we generate a node labeled by i . Subsequently, each non-zero element $\eta_{i,j}$ results in an undirected edge with weight $\eta_{i,j}$ between nodes i and j . This results in a weighted adjacency graph where the node labels are reordered using the following two heuristics.

5.4.2.1 Weighted Cuthill-McKee ordering

The Cuthill-McKee (CM) ordering is a bandwidth reducing node ordering heuristic introduced by Cuthill and McKee (1969). The standard algorithm places non-zero elements near the diagonal to result in a matrix with a lower bandwidth. We introduce an adjustment to the standard algorithm, such that it is able to differentiate between non-zero elements. Higher valued elements are prioritized in being placed close to the diagonal over lower valued elements. We will refer to this algorithm as the *weighted* CM ordering, which is shown in Algorithm 5.3. Lines 7 and 8 are an adjustment of the standard algorithm. As a convention, the for-loop at line 2 selects sub-graphs in descending size.

5.4.2.2 Sloan's ordering

Sloan's ordering is a profile and wavefront reducing node ordering heuristic introduced by Sloan (1989). It places non-zero elements near the diagonal to result in a lower profile of the matrix. In this chapter the standard algorithm is not adjusted to be able to differentiate between non-zero elements, although this is of interest for future work.

Algorithm 5.3 Weighted Cuthill-McKee ordering.

Input: NDSM M **Output:** Variable order list R

- 1: Initialize empty list R , compute weighted adjacency graph A of M
 - 2: **for each** Connected subgraph A' of A not connected to another node in A
 - 3: Compute pseudo-peripheral node p of A'
 - 4: Mark p and append p to R
 - 5: **while** Unmarked nodes exist in A'
 - 6: Find list C of unmarked neighbors of p
 - 7: Sort list C such that the nodes are in descending weight
 - 8: Sort list C such that nodes with equal weight are in ascending degree
 - 9: Append C to R and mark all nodes in C
 - 10: Set the next node in R as p
 - 11: **end**
 - 12: **end**
-

5.4.2.3 Weighted Event Span

We apply both ordering heuristics to the NDSM indicating related pairs of variables. This results in two orders. Furthermore, we notice that reversing the order can sometimes lead to significant differences in synthesis effort. Siminiceanu and Ciardo (2006) noticed that placing variables that result in more costly operations towards the bottom of the BDD resulted in less effort required in a similar application of BDDs. This resulted in the Weighted Event Span (WES) metric. Furthermore, the WES has been extensively tested by Meijer and van de Pol (2016), where a correlation is shown between peak BDD nodes, computation time, and the WES for several types of decision diagrams applied to symbolic model checking. Given a variable order, the WES is found by

$$\text{WES} = \sum_{e \in E} \frac{2x_l(e)}{|X|} \cdot \frac{x_l(e) - x_h(e) + 1}{|X||E|}, \quad (5.5)$$

where $|X|$ and $|E|$ respectively indicate the total number of variables and edges. $x_l(e)$ and $x_h(e)$ are respectively the lowest- and highest variable index from the variables in $T_e(X)$. The first term in Equation 5.5 increases as $x_l(e)$ is placed later in the variable order. The second term increases the WES when $x_l(e) - x_h(e)$ is large.

To estimate which of the four orders (two orders resulting from two different ordering heuristics and two reverse orders) should be used in synthesis, the WES is computed for each of the orders. The order that has the lowest WES is used in synthesis. This results in the proposed variable ordering heuristic, named DSM-based Cuthill-McKee-Sloan variable ordering Heuristic, abbreviated to DCSH for ease of reference.

5.4.3 Experiments

In Lousberg et al. (2020), the efficiency of variable orders computed by DCSH was compared to that of FORCE+SW. As mentioned in Section 5.3.4, in Lousberg et al. (2020) DCSH used different variable relations than FORCE and SW. We repeat the same experiments of Lousberg et al. (2020) here, using the same variable relations as discussed above for all heuristic algorithms. Additionally, the experiments are now performed for all models in Table 5.1, which is a larger set of models than was used in Lousberg et al. (2020).

For each model in Table 5.1, 10,000 random initial variable orders are generated. Each of those orders, is ordered by FORCE+SW and DCSH (separately) and then synthesis is performed using the computed order. We noticed it may be beneficial to apply FORCE+SW on the order computed by DCSH, so essentially performing

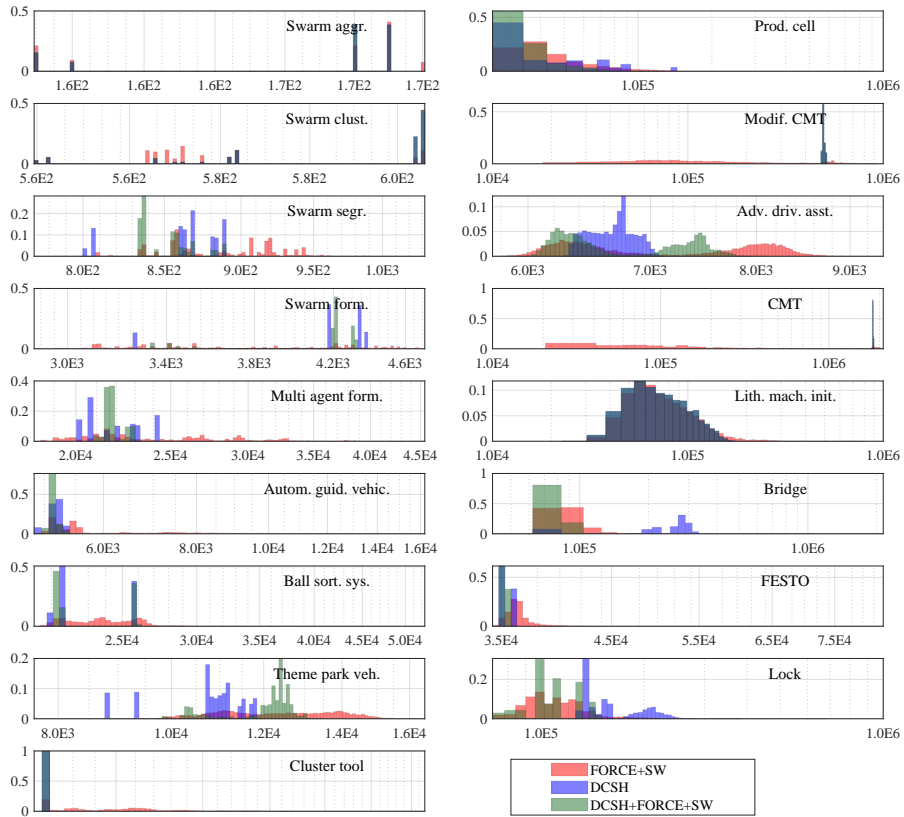


Figure 5.9: Distribution of peak used BDD nodes for 10,000 random initial variable orders, ordered by FORCE+SW, DCSH, and DCSH+FORCE+SW

sequence DCSH+FORCE+SW. We perform synthesis using each resulting variable order and measure the computational effort. Besides the initial variable order and turning on the BDD measurements, other settings in CIF were kept default in these experiments.

For each model, the computational effort using FORCE+SW, DCSH, and DCSH+FORCE+SW is plotted in a histogram, respectively for peak used BDD nodes in Figure 5.9 and BDD operation count in Figure 5.10. The histograms are created by dividing the space between the lowest and highest measurement in 100 bins, and setting the height of each bar to the fraction of measurements that are in that bin. Note that while each bin has equal width, the bars in Figures 5.9 and 5.10 visually have varying widths due to logarithmic scaling along the horizontal axis.

For each heuristic method, the minimal, mean, and maximal value are shown in Figure 5.11, for both peak used BDD nodes and BDD operation count. These values

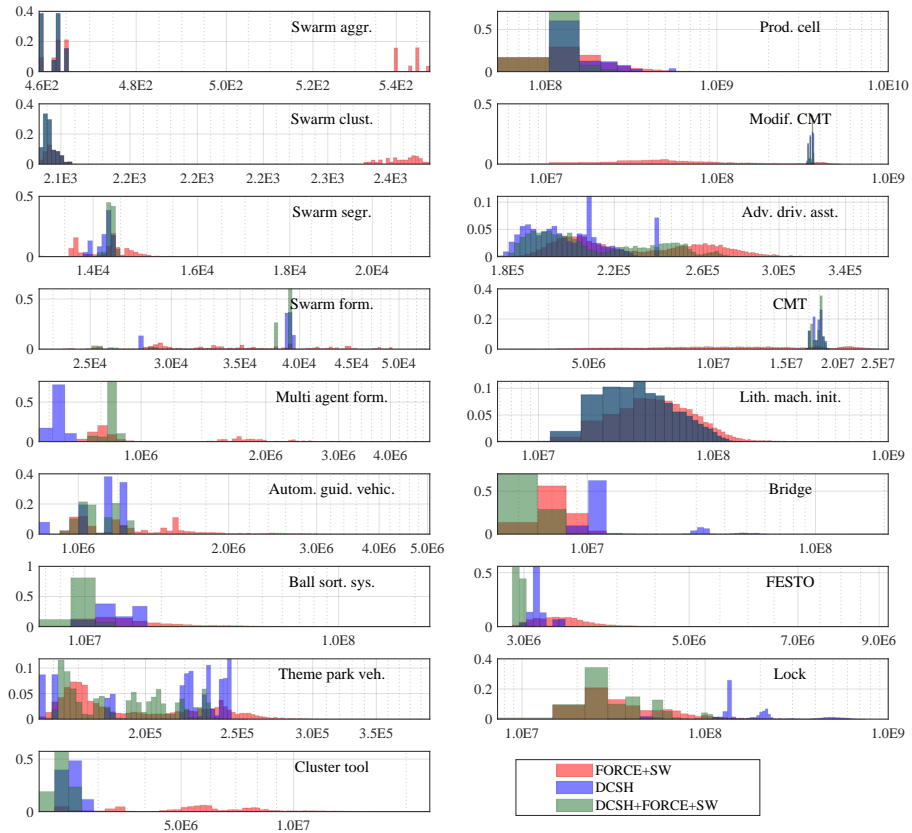


Figure 5.10: Distribution of BDD operation count for 10,000 random initial variable orders, ordered by FORCE+SW, DCSH, and DCSH+FORCE+SW

are normalized to the mean value for FORCE+SW for that model. E.g., a mean value of 0.8 ($8 \cdot 10^{-1}$) for DCSH+FORCE+SW for some model, indicates that for that model the mean computational effort of DCSH+FORCE+SW was 20% lower than the mean computational effort of FORCE+SW.

Summarizing the results, for both peak used BDD nodes and BDD operation count, for 16 out of 17 models, the maximal computational effort was found when using FORCE+SW. So, using FORCE+SW there is the likelihood to perform synthesis with relatively a really high computational effort. Generally, using DCSH or DCSH+FORCE+SW, removes most high-effort outliers: the maximal values are much lower. The mean of peak used BDD nodes over the measurements is less for DCSH+FORCE+SW compared to DCSH in 12 out of 17 models. For BDD operation count this is the case for 10 out of 17 models. Also, the maximal peak used BDD nodes using DCSH+FORCE+SW is less than the maximal peak used BDD nodes using just

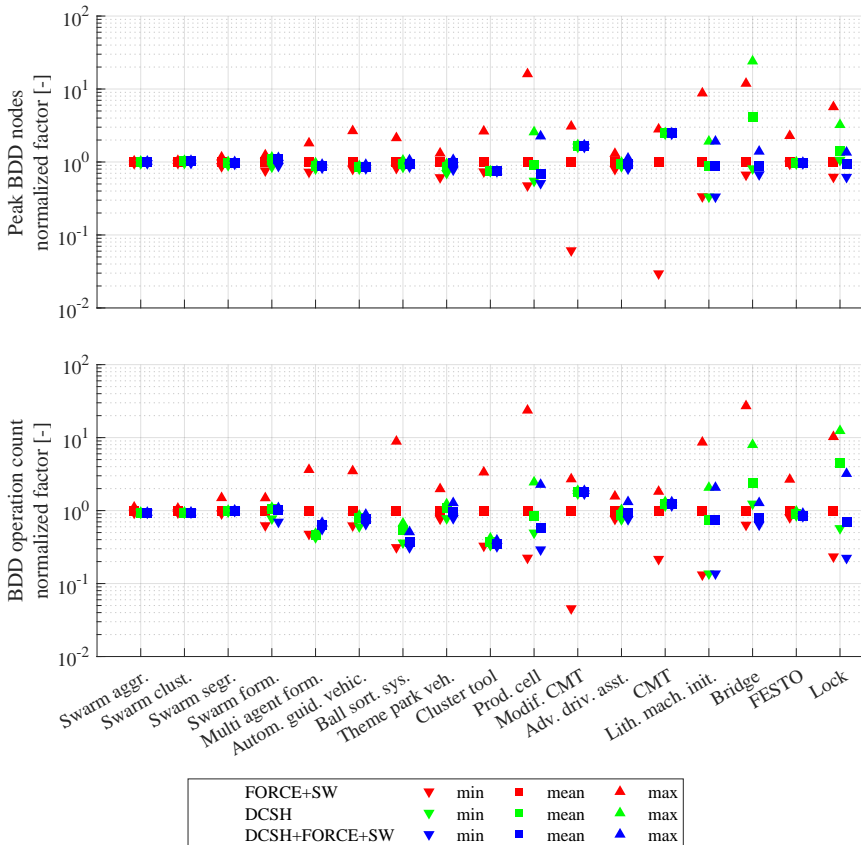


Figure 5.11: Min, mean, and max computational effort for variable ordering heuristics, normalized to mean computational effort to FORCE+SW

DCSH for 13 out of 17 models. For BDD operation count this is the case for 12 out of 17 models.

On average over all models, when using DCSH+FORCE+SW the peak used BDD nodes increase by 5%, and the BDD operation count lowers by 14%, relative to FORCE+SW. Relative to DCSH, it realizes 8% less peak used BDD nodes and a 12% lower BDD operation count. Even though DCSH+FORCE+SW realizes a slightly higher average amount of peak used BDD nodes than FORCE+SW, it is very effective at avoiding the high effort computations. In conclusion, using DCSH+FORCE+SW generally the least computational effort is required, and computations with relatively high computational effort are avoided. Therefore, using DCSH+FORCE+SW is advisable over just applying FORCE+SW or DCSH.

The time to run these variable ordering heuristic algorithms is negligible relative to the time to perform synthesis.

5.5 Edge order

As shown in Section 5.3.4, next to the variable order also the edge order has a significant impact on the computational effort of synthesis. In the current implementation of CIF, there are six options to set the edge order:

1. model: the order in which each edge appears when reading the model top-to-bottom;
2. reverse-model: the reverse of ‘model’;
3. sorted: alphabetical sorting of the edges by their event label;
4. reverse-sorted: the reverse of ‘sorted’;
5. random: a random ordering (optionally with a seed);
6. a manually specified order.

In this section, we compare the efficiency of the first five options. There are perhaps more interesting approaches to ordering the edges, e.g., such as presented in Vahidi et al. (2006) and Fei et al. (2014). These approaches however do not satisfy our self-imposed restriction mentioned in Section 5.1 to investigate static (not on-the-fly) optimization. Yet, however simple the static edge ordering heuristics may be, we will see in the following they can still produce good results, and investigating which option to use is worthwhile, also in order to come up with more meticulous heuristics in the future.

For each model in Table 5.1, synthesis is performed using model, reverse-model, sorted, and reverse-sorted edge order. Also for 100 random edge orders synthesis is performed. Following Section 5.4, DCSH+FORCE+SW is applied as variable ordering heuristic in these experiments, using the variable relations as introduced in Section 5.4.2. Besides applying DCSH+FORCE+SW, setting the edge order, and turning on BDD measurements, other settings in CIF are kept default in these experiments.

The results of these experiments are shown in Figure 5.12. For model, reverse-model, sorted, and reverse-sorted edge order, computational effort is displayed, normalized to the mean of using the 100 random edge orders. E.g., a value of $2 \cdot 10^{-1}$ indicates an

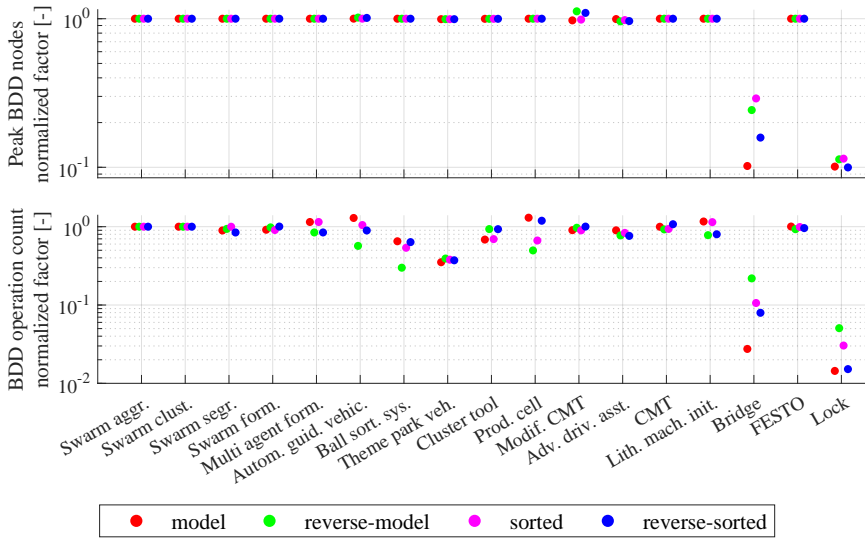


Figure 5.12: Computational effort reduction factor for edge order heuristics normalized to random.

80% reduction in effort compared to the random orders on average. These values are shown for the peak used BDD nodes, and the BDD operations count.

As was concluded in Section 5.3.4, the edge order has a smaller influence on the peak used BDD nodes compared to BDD operation count. For most models the peak used BDD nodes are similar for the different edge orders. However, for the Bridge and Waterway lock models, using any edge order option other than random considerably reduces the peak memory usage, with reductions up to 90%, compared to the mean of using random edge orders, for both models. Overall, for peak used BDD nodes the average reduction by using any edge order option other than random is 10%.

The edge order has a larger influence on the BDD operation count. Again, for the Bridge and Waterway lock models the highest impacts are found, with a reduction up to 99% for the Waterway lock model compared to the mean of using random edge orders. Overall, for BDD operation count the average reduction by using any edge order option other than random is 22%.

Over all models, the best average reduction for BDD operation count is found using reverse-model, with a average reduction of 29%. For model, sorted, and reverse-sorted these average reductions are 16%, 22%, and 21% respectively. Also, reverse-model is the only heuristic that performed the same or better than the averaged random edge orders for every model. Therefore, usage of the reverse-model edge order is advisable over the other edge ordering heuristics available in CIF.

We give a possible reason why model, reverse-model, sorted, and reverse-sorted generally perform better than the random edge orders. In all models, the model is

specified by a network of automata. If an event is locally specified within an automaton, its name gets prefixed with the automaton name (to avoid duplicate names). As a result for these ordering heuristics, edges that are used in the same automata are placed next to each other in the edge order. It is likely that edges specified in the same automaton, address similar variables in their guard and update expressions. Therefore, calculations on the parts of the BDD representing these variables are performed in close succession to each other. This means there is a higher likelihood of comparable calculations still being in the cache if we perform calculations on the same variables back-to-back. We refer back to Section 5.3.2: when a previous calculation is found in the cache, that result is used, no computations on the BDD need to be performed, and the BDD operation count is not incremented. So, iterating over edges in an order such that similar calculations are performed back-to-back improves the efficiency of the cache mechanism, therefore speeding up synthesis.

Also, we can reason why reverse-model generally has the best results. People generally write down the automata in a way that logically follows the behavior from top-to-bottom. For synthesis, the main computations are calculating the nonblocking and bad state predicate through backward reachability searches, i.e., the behavior is followed backwards (from the nonblocking or bad states). Evaluating an edge that does not lead to a currently found nonblocking/bad state costs computational effort, but does not aid in further construction of the nonblocking/bad state predicate. As a result, states are more efficiently found by evaluating edges in the reverse order of how they occur in the behavior, for which practically reverse-model is a good guess.

5.6 Efficiently enforcing requirements

Central to supervisory control theory, is the specification of behavioral requirements, and enforcing those through supervisor synthesis. In CIF, requirements can be specified in three ways:

1. *requirement EFAs*: prescribing allowed behavior w.r.t. a subset of the plant's events;
2. *state exclusion expressions*: an expression defining a condition that needs to hold in every state of the controlled system;
3. *state-event exclusion expressions*: an expression defining a condition that needs to hold for a particular event to occur.

Similarly to how an EFA was converted to an LFA prior to synthesis in Section 5.2.1, these requirements are also converted to a format based on predicates. Respectively:

1. *requirement LFAs*: prescribing allowed behavior w.r.t. a subset of the plant's events, which are direct linearizations from the requirement EFAs;
2. *state exclusion predicates*: a predicate using variables X that needs to hold in every state of the controlled system. Logically, this predicate directly relates to the condition specified in a state exclusion expression;

3. *state-event exclusion predicates*: defined by $\sigma \Rightarrow J$, where J is an expression defining a predicate over X that needs to hold for event σ to occur in the controlled system.

The latter two requirement types are discussed in Markovski et al. (2010). These are also the requirement types that we consider in this chapter. Note that requirement automata can be converted to plant automata to enforce them in synthesis (Flordal et al. 2007).

Even though we define the safe states or states from which events are allowed to occur, we call it ‘exclusion’ requirements because it is a restriction on the plant behavior.

Conversion of the requirements to their predicate-based counterparts is relatively straightforward, and specification of requirements, along with this conversion, is exemplified in Example 5.3.

Example 5.3 Traffic lights

We consider two traffic lights, each regulating traffic for their road at a two-way intersection. The plant behavior can be modeled by two automata, given in Figure 5.13.



Figure 5.13: Traffic lights plant automata.

The informal requirement is that the traffic lights should not be green at the same time, as this may result in a collision. This requirement can be formalized by a requirement EFA, given in Figure 5.14.

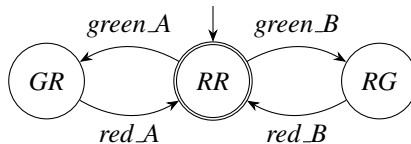


Figure 5.14: Traffic lights requirement automaton.

Alternatively, the requirement specification can be given by a state exclusion expression (i.e., this is the syntax a modeler would use in CIF):

$$\text{not}(\text{LightA.Green and LightB.Green}),$$

which directly relates to a state exclusion predicate:

$$\neg(l_A = \text{Green} \wedge l_B = \text{Green}),$$

where, e.g., l_A is the location pointer variable for *LightA* that can take values *Green* and *Red*.

As another option, the modeler may give two state-event exclusion expressions, specifying that one light can only be turned green if the other light is red:

$$\begin{aligned} \text{green}_A \text{ needs } \text{LightB.Red}, \\ \text{green}_B \text{ needs } \text{LightA.Red}. \end{aligned}$$

These expressions, written in CIF syntax, can be directly converted to state-event exclusion predicates:

$$\begin{aligned} \text{green}_A \Rightarrow l_B = \text{Red}, \\ \text{green}_B \Rightarrow l_A = \text{Red}. \end{aligned}$$

We will refer to the set of all state exclusion predicates as *SX*, and to the set of all state-event exclusion predicates as *EX*. For simplicity we will consider specifications that do not contain any requirement EFA. Enforcing the requirements expressed by automata in (symbolic) supervisor synthesis is well known (Ramadge and Wonham 1987; Flordal et al. 2007; Ouedraogo et al. 2011; Cassandras and Lafortune 2021).

Through general usage of CIF, it has been noticed empirically that the manner in which the requirements are modeled can impact the efficiency of performing supervisor synthesis, even if they represent the same informal requirement specification and the same controlled behavior is achieved. Notably the usage of state exclusion expressions would lead to computations that require a lot of time and memory. Consequently, this type of requirement specification was sometimes avoided when modeling larger systems. This can be observed in, e.g., the Waterway lock model from Reijnen et al. (2017).

For the purpose of modeling ease and model clarity, in a number of cases it might be useful to use state exclusion expressions. For the traffic lights in Example 5.3, the state exclusion expression is arguably the most straightforward formalization of the informal requirement specification. Ideally, the usage of state exclusion expressions would not be penalized by a higher computational effort in synthesis. This introduces the problem statement discussed in this section: How can state exclusion requirements be enforced (more) efficiently in symbolic supervisor synthesis?

We discuss the current way requirements are enforced in CIF in Section 5.6.1. Then we present our modified approach in Section 5.6.2. Experiments comparing both approaches are presented in Section 5.6.3.

5.6.1 Current application of requirements

We first introduce the current way requirements are enforced during synthesis in CIF (before the new algorithm we discuss in Section 5.6.2 was implemented).

In the synthesis algorithm (Algorithm 5.1), first requirements are applied by Algorithm 5.4. In this algorithm, a safe state predicate P is computed by first taking the conjunction of all state exclusion predicates. This predicate returns true only for states for which all state exclusion predicates hold. Note that the empty conjunction is assumed true. Next, the state-event exclusion predicates are enforced by computing a safe state predicate and safe edges in Algorithm 5.5. When these requirements consider controllable events, the guard of each edge labeled by that event can simply be strengthened by taking the conjunction with the predicate, so that the event only occurs when the predicate holds. This is not possible for uncontrollable events, because the supervisor is not able to disallow uncontrollable events from occurring when they can occur in the plant. In that case, the safe state predicate is modified to exclude states from which the event can take place (i.e., for some edge (σ, g, u) , labeled by the same event σ that the state-event exclusion predicates addresses, g evaluates to true), but the state-event exclusion predicate does not hold (i.e., J evaluates to false). The predicate $g \implies J$ specifies the states where the state-event exclusion requirement is adhered to. Finally, $X_0 \wedge P$ restricts the initial state predicate to the safe part⁵.

Algorithm 5.4 `applyRequirements`

Input: Mutual state exclusion predicates SX , state-event exclusion predicates EX , edges E , initial state predicate X_0

Output: Safe state predicate P , safe edges E , safe initial state predicate X_0

- 1: $P = \bigwedge_{I \in SX} I$
 - 2: $(P, E) = \text{applyEventRequirements}(P, E, EX)$
 - 3: $X_0 = X_0 \wedge P$
-

Algorithm 5.5 `applyEventRequirements`

Input: State predicate P , edges E , state-event exclusion predicates EX

Output: Safe state predicate P , safe edges E

- 1: **for all** $(\sigma, g, u) \in E, (\sigma \implies J) \in EX$
 - 2: **if** $\sigma \in \Sigma_c$
 - 3: $g = g \wedge J$
 - 4: **else**
 - 5: $P = P \wedge (g \implies J)$
 - 6: **end if**
 - 7: **end for**
-

⁵ Strictly speaking, calculating the initial state predicate here is not necessary, and only needs to be performed at the end of synthesis (line 11 Algorithm 5.1). We already calculate $X_{0,S}$ here to simplify our proofs.

In the following we show that, when supervisor synthesis is performed using Algorithm 5.4 to apply the requirements, supervisor synthesis (Algorithm 5.1) computes the correct result, i.e., the maximal, safe, controllable, and nonblocking supervisor. Our explanation is structured as follows: We first define a *safe state* and *safe LFA* in Definition 5.1. We then define when an LFA is *minimally restricted* in Definition 5.3, i.e., no more behavior is removed from the LFA than strictly necessary so that the requirements are satisfied. We show that after performing `applyRequirements`, we can induce an LFA that is both safe (Lemma 5.1) and minimally restricted (Lemma 5.2) with respect to the requirements. Since Algorithm 5.1 follows the same structure as the synthesis algorithm in Ouedraogo et al. (2011), it is known that after performing `applyRequirements` in line 1 of `SS`, the remaining lines (2-11) compute the maximal, controllable, and nonblocking supervisor (Theorem 3 in Ouedraogo et al. (2011)), of the minimally restricted safe LFA. Therefore, we show in Theorem 5.1 that Algorithm 5.1 computes the maximal, safe, controllable, and nonblocking supervisor.

Given a set of symbols X , let $\Phi(X)$ be the set of functions $\phi(X)$, where a function $\phi(X)$ assigns a value to each variable $x \in X$, in the domain of x . We write a function $\phi(X)$ as a predicate $\bigwedge_{x \in X} x = \phi(x)$. E.g., say we have variables $X = \{s, t\}$, where the domain of s is $\{1, 2\}$ and the domain of t is $\{3, 4\}$, then: $\Phi(X) = \{(s=1 \wedge t=3), (s=1 \wedge t=4), (s=2 \wedge t=3), (s=2 \wedge t=4)\}$. Given an LFA A_L with symbols X , we refer to a function $\phi(X)$ as a *state* of A_L . We may write ϕ and Φ to refer to $\phi(X)$ and $\Phi(X)$ respectively. For a predicate $P(X)$ we may write $P(\phi)$ to denote the valuation of P for state ϕ .

We say there is a transition from $\phi \in \Phi$ to $\phi' \in \Phi$, if there is some edge (σ, g, u) for which $g(\phi) \wedge u(\phi, \phi')$. We say this transition is controllable or uncontrollable, when respectively $\sigma \in \Sigma_c$ or $\sigma \in \Sigma_u$.

Definition 5.1 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then a state $\phi \in \Phi$ is *safe* when:

- $\forall I \in SX : I(\phi)$, and
- $\forall (\sigma, g, u) \in E, (\sigma \Rightarrow J) \in EX : g(\phi) \implies J(\phi)$.

We call LFA A_L *safe* if all its reachable states are safe. Non-safe states or automata are called *unsafe*.

Definition 5.2 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, then *restricted* LFA $A_{L,R}$, with respect to predicate N , edges E_S , and safe initial states $X_{0,S}$, is defined as $A_{L,R} = (X, \Sigma, E_{S \triangleleft N}, X_{0,S}, X_m)$, where $E_{S \triangleleft N} = \{(\sigma, g(X) \wedge \exists X^+ [N(X^+) \wedge u(X, X^+)], u) \mid (\sigma, g, u) \in E_S\}$.

Note that in Definition 5.2 the guards are restricted in the same manner as in line 9 of Algorithm 5.1.

Definition 5.3 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then after performing $(N, E_S, X_{0,S}) = \text{applyRequirements}(SX, EX, E, X_0)$, restricted automaton $A_{L,R}$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is *minimally restricted* w.r.t. A_L, SX , and EX when:

- all initial states in $\{\phi \in \Phi \mid X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ are unsafe, and
- all restricted transitions lead to an unsafe state, or they are disallowed by some state-event exclusion predicate: $\forall \phi, \phi' \in \Phi : (\exists (\sigma, g, u) \in E : g(\phi) \wedge u(\phi, \phi') \wedge \nexists (\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi')) \implies (\text{state } \phi' \text{ is unsafe or } \exists (\sigma, J) \in EX : \neg J(\phi))$.

I.e. only unsafe initial states are removed, and all transitions that are removed directly lead to an unsafe state or are disallowed by some state-event exclusion predicate.

Lemma 5.1 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then after performing $(N, E_S, X_{0,S}) = \text{applyRequirements}(SX, EX, E, X_0)$, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \rightarrow N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is safe w.r.t. A_L, SX , and EX .

Proof For all controllable transitions, it directly follows from line 3 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \implies J) \in EX$, with $\sigma \in \Sigma_c : g(\phi) \implies J(\phi)$.

It directly follows from line 1 in Algorithm 5.4 that $\forall \phi \in \Phi, I \in SX : N(\phi) \implies I(\phi)$.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \implies J) \in EX$, with $\sigma \in \Sigma_c : N(\phi) \implies (g(\phi) \implies J(\phi))$. Therefore, all states in $\{\phi \in \Phi \mid N(\phi)\}$ are safe.

From line 3 in Algorithm 5.4 we can conclude that all initial states $\{\phi \in \Phi \mid X_{0,S}(\phi)\}$ are safe. From the definition of $A_{L,R}$, and specifically $E_{S \rightarrow N}$, we can conclude that only transitions to safe states are possible. Therefore, $A_{L,R}$ is safe. \square

Lemma 5.2 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then after performing $(N, E_S, X_{0,S}) = \text{applyRequirements}(SX, EX, E, X_0)$, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \rightarrow N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is minimally restricted w.r.t. A_L, SX , and EX .

Proof For all controllable transitions, it follows from line 3 in Algorithm 5.5 that $\forall \phi, \phi' \in \Phi : (\exists (\sigma, g, u) \in E : \sigma \in \Sigma_c \wedge g(\phi) \wedge u(\phi, \phi') \wedge (\nexists (\sigma \implies J) \in EX : \neg J(\phi)) \implies \exists (\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi'))$. I.e., guards of controllable transitions in E_S are not restricted from E when there is no state-event exclusion predicate that does not disallow them.

It directly follows from line 1 in Algorithm 5.4 that $\forall \phi \in \Phi, I \in SX : \neg I(\phi) \implies \neg N(\phi)$.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S \mid \sigma \in \Sigma_u\}, (\sigma \implies J) \in EX : \neg(g(\phi) \implies J(\phi)) \implies \neg N(\phi)$. Therefore, all states in $\{\phi \in \Phi \mid \neg N(\phi)\}$ are unsafe.

From the definition of $A_{L,R}$, and specifically $E_{S \rightarrow N}$, we can conclude that only transitions in E_S to unsafe states are restricted.

Also, since all states $\{\phi \in \Phi \mid \neg N(\phi)\}$ are unsafe, it follows that all states in $\{\phi \in \Phi \mid X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ are unsafe. \square

Theorem 5.1 The supervisor obtained by Algorithm 5.1 is a maximal, safe, controllable, and nonblocking supervisor for automaton A_L and requirements SX and EX .

Proof Note that during the repeat-until loop in Algorithm 5.1, for all states $\phi \in \Phi$, if at the start of the loop $\neg N(\phi)$, then because BRS is restricted by N , after line 4 $\neg N(\phi)$ still holds. Also after lines 5 and 6 $\neg N(\phi)$ still holds, because BRS has $\neg N(\phi)$ as a start predicate, and only the disjunction with other predicates is taken when computing B , except for conjunction with *true*. So after line 5, we know that $B(\phi)$ holds, so after line 6 $\neg N(\phi)$ still holds. In other words, as shown in Lemma 5.2, all states $\phi \in \Phi$ for which $\neg N(\phi)$ after line 1 of Algorithm 5.1 are unsafe, and these will remain unsafe (/bad/blocking) states during the fixpoint computation. Note also that in the restriction of Definition 5.2, the edges are restricted in the same manner as in in line 9 of Algorithm 5.1.

It is shown in Theorem 3 in Ouedraogo et al. (2011) that lines 2-11 compute a maximal, controllable, and nonblocking supervisor. This is a maximal, controllable, and nonblocking supervisor, for the LFA that is safe (Lemma 5.1), and minimally restricted (Lemma 5.2). It follows that Algorithm 5.1 computes a maximal, safe, controllable, and nonblocking supervisor for automaton A_L and requirements SX and EX . \square

5.6.2 Efficient application of requirements

As stated above, it has been found empirically that synthesis on models containing state exclusion expressions was inefficient. The problem is that, when there are many state exclusion expressions, the BDD describing the safe state predicate can become quite large. This predicate is the starting point for the nonblocking predicate, which is continuously updated during synthesis. It is beneficial to keep the BDD representing this predicate as small as possible, to have low computational effort for synthesis.

Because synthesis on models containing state exclusion expressions was inefficient, they are sometimes manually converted to state-event exclusion expressions. From practice it has been found that this can solve the inefficiency problem. This is because the requirements are encoded into the guards of the edges, rather than into the non-blocking predicate. Therefore, we seek our solution in the same direction: we enforce state exclusion requirements in the same manner as state-event exclusion requirements. This is done in Algorithm 5.6.

In Algorithm 5.6, for each state exclusion predicate I and each edge (σ, g, u) , a predicate J is constructed that expresses the states from which the edge can be performed such that I holds after executing the edge. In the controlled behavior, the edge can only be performed from the states indicated by $g \wedge I$, because states where I does not hold are not reached by a safe supervisor. In all cases that the edge can be performed, J must hold so that a safe state is reached. In line 5 it is checked whether there are any states for which this does not hold. If that is the case, the respective edge is restricted in the same way as for state-event exclusion predicates, note that lines 6-10 in Algorithm 5.6 are the same as lines 2-6 in Algorithm 5.5. So, if the edge is labeled by a controllable event, the guard is restricted by J . If the edge is labeled by an uncontrollable event, the safe state predicate is restricted so that it only describes

Algorithm 5.6 `applyRequirementsEfficient`

Input: Mutual state exclusion predicates SX , state-event exclusion predicates EX , edges E , initial states X_0

Output: Safe state predicate P , safe edges E , safe initial states X_0

```

1:  $P = true$ 
2: for all  $I \in SX$ 
3:   for all  $(\sigma, g, u) \in E$ 
4:      $J(X) = \exists_{X^+} [g(X) \wedge u(X, X^+) \wedge I(X^+)]$ 
5:     if  $(g \wedge I \implies J) \neq true$ 
6:       if  $\sigma \in \Sigma_c$ 
7:          $g = g \wedge J$ 
8:       else
9:          $P = P \wedge (g \implies J)$ 
10:      end if
11:    end if
12:  end for
13: end for
14:  $(P, E) = applyEventRequirements(P, E, EX)$ 
15:  $X_0 = X_0 \wedge P \wedge \bigwedge_{I \in SX} I$ 

```

states where if the edge can occur, a state is reached where I holds. After enforcing all state exclusion requirements for all edges, the state-event exclusion requirements are applied in the same way as in Algorithm 5.4, by using Algorithm 5.5. Finally, the initial state predicate is modified so that all state exclusion predicates hold in the initial state. Thus, the system starts in a safe state, and does not leave the safe states as a result of the restricted guards.

We show in Example 5.4 how the state exclusion predicate given in Example 5.3 is enforced by Algorithm 5.6.

Example 5.4 Traffic lights continued

This is a continuation of Example 5.3, where a traffic light system was modeled, and requirements were specified for that model. We consider the state exclusion predicate:

$$\neg(l_A = Green \wedge l_B = Green).$$

We consider LightA turning green. In this case there is only one edge labeled with this event, which is: $e_{gA} = (green_A, l_A = Red, l_A^+ = Green \wedge l_B^+ = l_B)$. Computing J as in line 4 of Algorithm 5.6, provides the following predicate:

$$\begin{aligned}
J &= \exists_{X^+} [(l_A = Red \wedge l_A^+ = Green \wedge l_B^+ = l_B \wedge \neg(l_A^+ = Green \wedge l_B^+ = Green))] \\
&\equiv l_A = Red \wedge \neg(l_B = Green) \\
&\equiv l_A = Red \wedge l_B = Red
\end{aligned}$$

Next, we compute the $(g \wedge I \implies J)$ as in line 5:

$$\begin{aligned} l_A = Red \wedge \neg(l_A = Green \wedge l_B = Green) &\implies l_A = Red \wedge l_B = Red \\ \equiv l_A = Red &\implies l_A = Red \wedge l_B = Red \\ \equiv l_A = Red &\implies l_B = Red. \end{aligned}$$

We can see that this does not equal *true*, i.e., there are states from which edge e_{g_A} can be taken such that the state exclusion predicate does not hold afterwards.

Because *green_A* is a controllable event, its guard g_A is restricted as follows:

$$\begin{aligned} g_A = l_A = Red \wedge l_A = Red \wedge l_B = Red \\ \equiv l_A = Red \wedge l_B = Red. \end{aligned}$$

Similarly, repeating the same steps for edge e_{g_B} that models LightB turning green, would result in the following restricted guard g_B :

$$g_B = l_A = Red \wedge l_B = Red.$$

One can verify that these guards computed by Algorithm 5.6 restrict the behavior in the same way as the state-event exclusion predicates provided in Example 5.3.

In Algorithm 5.1, we can substitute line 1 with the following line, to apply the introduced efficient enforcement of the requirements:

$$1: (N, E_S, X_{0,S}) = \text{applyRequirementsEfficient}(SX, EX, E, X_0)$$

We will refer to Algorithm 5.1 with line 1 substituted as above as SS' .

Same as in Section 5.6.1, we show that SS' computes the maximal, safe, controllable, and nonblocking supervisor in Theorem 5.2. We do so by first showing that the induced restricted LFA (by Definition 5.2) after performing `applyRequirementsEfficient` is both safe (Lemma 5.3) and minimally restricted (Lemma 5.4) with respect to the requirements.

Lemma 5.3 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then after performing $(N, E_S, X_{0,S}) = \text{applyRequirementsEfficient}(SX, EX, E, X_0)$, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \wedge N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is safe w.r.t. A_L, SX , and EX .

Proof For all controllable transitions, it directly follows from line 3 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \implies J) \in EX$, with $\sigma \in \Sigma_c : g(\phi) \implies J(\phi)$. For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in E_S, (\sigma \implies J) \in EX$, with $\sigma \in \Sigma_u : N(\phi) \implies (g(\phi) \implies J(\phi))$. So, in the restricted automaton the state-event exclusion predicates are satisfied for all transitions originating from a state $\phi \in \Phi$ where $N(\phi)$ holds.

For all controllable transitions, it directly follows from line 7 in Algorithm 5.6 that $\forall \phi, \phi' \in \Phi, (\sigma, g, u) \in E_S$, with $\sigma \in \Sigma_c, I \in SX : g(\phi) \wedge u(\phi, \phi') \wedge I(\phi) \implies$

$g(\phi) \wedge u(\phi, \phi') \wedge I(\phi')$. For all uncontrollable transitions, it directly follows from line 9 in Algorithm 5.6 that $\forall \phi, \phi' \in \Phi, (\sigma, g, u) \in E_S$, with $\sigma \in \Sigma_u, I \in SX : N(\phi) \implies (g(\phi) \wedge u(\phi, \phi') \implies I(\phi'))$. So, in the restricted automaton there are no transitions to a state that does not satisfy some state exclusion predicate.

Since for all states in $\{\phi \in \Phi | N(\phi)\}$ it is implied that no uncontrollable transition can be performed that does not satisfy a state-event exclusion predicate, all states in $\{\phi \in \Phi | N(\phi) \wedge_{I \in SX} I(\phi)\}$ are safe. From line 15 in Algorithm 5.6 we can conclude that all initial states in $\{\phi \in \Phi | X_{0,S}(\phi)\}$ are safe. Since $A_{L,R}$ only has safe initial states, and can only transition to safe states, $A_{L,R}$ is safe. \square

Lemma 5.4 Given LFA $A_L = (X, \Sigma, E, X_0, X_m)$, and requirements SX and EX , then after performing $(N, E_S, X_{0,S}) = \text{applyRequirementsEfficient}(SX, EX, E, X_0)$, restricted automaton $A_{L,R} = (X, \Sigma, E_{S \leftarrow N}, X_{0,S}, X_m)$ (w.r.t. $A_L, N, E_S, X_{0,S}$) is minimally restricted w.r.t. A_L, SX , and EX .

Proof For all controllable transitions, it follows from line 3 in Algorithm 5.5 and line 7 in Algorithm 5.6 that $\forall \phi, \phi' \in \Phi : (\exists (\sigma, g, u) \in E : \sigma \in \Sigma_c \wedge g(\phi) \wedge u(\phi, \phi') \wedge (\nexists (\sigma \implies J) \in EX : \neg J(\phi)) \wedge (\nexists I \in SX : \neg I(\phi')))) \implies \exists (\sigma, g_S, u_S) \in E_S : g_S(\phi) \wedge u_S(\phi, \phi')$. I.e., guards of controllable transitions in E_S are not restricted from E when there is no state-event exclusion predicate that does not disallow them, and there is no state exclusion predicate that is not satisfied in the target state.

For all uncontrollable transitions, it directly follows from line 5 in Algorithm 5.5 that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S | \sigma \in \Sigma_u\}, (\sigma \implies J) \in EX : \neg(g(\phi) \implies J(\phi)) \implies \neg N(\phi)$. For all uncontrollable transitions, it directly follows from line 9 in Algorithm 5.6 that $\forall \phi \in \Phi, (\sigma, g, u) \in \{(\sigma, g, u) \in E_S | \sigma \in \Sigma_u\}, I \in SX : \neg(g(\phi) \wedge I(\phi) \implies g(\phi) \wedge u(\phi, \phi') \wedge I(\phi')) \implies \neg N(\phi')$. Therefore, all states in $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe. From the definition of $A_{L,R}$, and specifically $E_{S \leftarrow N}$, we can conclude that only transitions in E_S to unsafe states are restricted.

Also, since all states $\{\phi \in \Phi | \neg N(\phi)\}$ are unsafe, it follows that all states in $\{\phi \in \Phi | X_0(\phi) \wedge \neg X_{0,S}(\phi)\}$ are unsafe. \square

Theorem 5.2 The supervisor obtained by Algorithm 5.1 is a maximal, safe, controllable, and nonblocking supervisor for automaton A_L and requirements SX and EX .

Proof The same proof as in Theorem 5.1 applies here. \square

Not necessarily the supervisor LFAs computed by SS and SS' are the same. When applyRequirements is used, the guards of all edges are restricted such that they can never reach an unsafe state, also if the edge originates from another unsafe state. Since it is assumed in $\text{applyRequirementsEfficient}$ that unsafe states are never reached, the guards of the edges from unsafe states are not necessarily restricted such that they can never reach another unsafe state. Regardless, the behavior under control of the supervisor, that only reaches safe states, is the same.

5.6.3 Experiments

Here we compare the computational effort of SS and SS' . In the set of benchmark models (Table 5.1), there are three models that use state exclusion expressions, which are: Lithography machine initialization, Cat and mouse tower (with 3 floors, 2 cats, and 2 mice), and Modified cat and mouse tower (with 3 floors, and at most 1 cat or 1 mouse per room). As the suggested approach only influences synthesis of these models, experiments are only performed for these models. The lithography machine initialization model contains 51 state exclusion expressions. Both cat and mouse tower models contain 15 state exclusion expressions. More details on these models can be found in Thuijsman et al. (2021).

For each model, synthesis is performed using default CIF settings, other than: DCSH+FORCE+SW is applied (as a result of Section 5.4) using the variable relations as introduced in Section 5.4.2, edge order is set to reverse-model (as a result of Section 5.5), and BDD measurements are turned on. The results of these experiments are shown in Table 5.3. For all models the efficient approach (Algorithm 5.6) requires less computational effort than the current approach (Algorithm 5.4). Generally, there is a small decrease in peak used BDD nodes, which is reduced by 8% on average. The computational benefit for BDD operation count is more significant. For these models, the BDD operation counts were decreased by 64% on average. Note that the suggested method is not necessarily always more efficient than the current method. Nevertheless, these experiments suggest that using `applyRequirementsEfficient` rather than `applyRequirements` is beneficial.

Table 5.3: Experimental results efficiently enforcing requirements.

Name	Peak used BDD nodes		BDD operation count	
	SS	SS'	SS	SS'
Lith. mach. init.	$4.63 \cdot 10^4$	$4.17 \cdot 10^4$	$1.68 \cdot 10^7$	$5.88 \cdot 10^6$
CMT	$1.82 \cdot 10^6$	$1.81 \cdot 10^6$	$1.67 \cdot 10^7$	$1.09 \cdot 10^7$
Modif. CMT	$5.68 \cdot 10^5$	$4.92 \cdot 10^5$	$3.92 \cdot 10^8$	$2.83 \cdot 10^7$

To provide further discussion on the influence of `applyRequirements` and `applyRequirementsEfficient` on the computational effort, we study the evolution of the BDD during synthesis for the two methods. For Lithography machine initialization, the BDD evolution during synthesis is displayed in Figure 5.15. One can validate that the peak used BDD nodes and final BDD operation count are indeed lower when SS' is applied instead of SS (and match the values in Table 5.3). Performing `applyRequirementsEfficient` actually requires more BDD operations than performing `applyRequirements` (i.e., only performing line 1 of SS' and SS , not yet the remainder). Respectively, these algorithms are finished after $9.5 \cdot 10^5$ and $1.8 \cdot 10^5$ BDD operations. So, SS' starts later on its fixpoint computation (lines 2-7 in Algorithm 5.1) than SS . However, this computation is less costly in SS' , because the

state exclusion predicates do not appear directly in the nonblocking predicate, which is the case for SS . At this point, the additional computational effort that was invested when applying the predicates is “won back” (and more) by SS' , leading to a lower computational effort overall. The peak that is observed at the end of both syntheses in Figure 5.15 is a result of restricting the guards (lines 8-10 in Algorithm 5.1).

The efficiency of `applyRequirementsEfficient` likely depends on the number of edges labeled by controllable/uncontrollable events in the system. When there are many edges labeled by an uncontrollable event, the state exclusion requirements are still encoded in the nonblocking predicate. Unfortunately, at the moment we do not have any more models containing state exclusion expressions to use for further experimentation. In part, this is because they were previously avoided because of their inefficient application in synthesis.

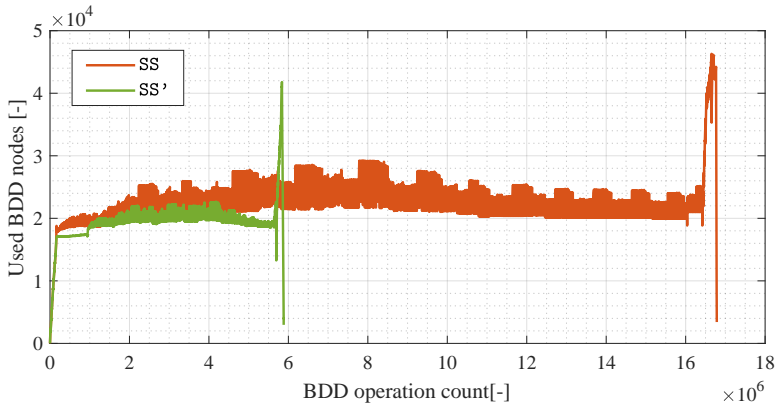


Figure 5.15: BDD evolution for SS and SS' of Lithography machine initialization.

5.7 Conclusion

The computational effort of symbolic supervisor synthesis can be expressed using peak used BDD nodes and BDD operation count. Unlike wall clock time and peak random access memory, these BDD-based metrics are platform-independent, deterministic, and include no overhead in their measurement. BDD-based metrics can be used to analyze, and improve, the efficiency of the synthesis algorithm. In this chapter we showcase this approach by: introducing and analyzing DCSH, a variable ordering heuristic; analyzing several edge ordering heuristics; and introducing and analyzing an approach to efficiently enforce state exclusion requirements. It is shown that:

1. Even though using DCSH+FORCE+SW on average requires 5% more peak used BDD nodes than FORCE+SW, it on average realizes a 14% lower BDD operation

count and for 16 out of 17 models it resulted in both a lower maximal measured peak used BDD nodes and BDD operation count. Therefore, by applying DCSH+FORCE+SW as variable ordering heuristic, performing synthesis with relatively high computational effort can be avoided, and generally relatively low computational effort is required.

2. Using reverse-model edge order realizes relatively low synthesis effort, averaging 10% lower peak used BDD nodes and a 29% lower BDD operation count than using random edge orders.
3. State exclusion requirements can efficiently be enforced by restricting edge guards prior to synthesis. On average, this method reduces the peak used BDD nodes by 8% and BDD operation count by 64%, relative to the conventional method.

These methods are implemented in the ESCET toolkit, and therefore available to all those who wish to use them. Experiments like presented in this chapter help in selecting what methods or settings should be used by default. For instance, in the newest ESCET release DCSH+FORCE+SW is now applied by default.

From the experimental results it becomes clear that generally there are no one-size-fits-all solutions. What works for one model, does not necessarily work for another model. Unfortunately it is hard to predict when this is the case. A small change in the synthesis input, e.g., the variable order, can have a huge influence on the synthesis effort. This means that methods need to be thoroughly validated, which we do in this work. Nevertheless, these huge variances in effort also indicate how much improvement can be made. Scalability is a major factor in the industrial acceptance of supervisory control theory. This makes it worthwhile to investigate techniques like the ones discussed in this chapter, to be able to keep tackling the engineering of supervisory controllers for larger and more complex systems with supervisor synthesis.

Future work

As (industrial) systems generally become more and more complex, the computational efficiency of symbolic supervisor synthesis should continuously be improved in the future. With respect to static ordering of variables or edges, more heuristics can be investigated, and their efficiency together with, or compared to, the heuristics discussed in this chapter can be analyzed. Furthermore, dynamic reordering during synthesis could bring additional benefits, e.g., as considered in Panda et al. (1994) or Ranjan et al. (1995) for dynamic variable reordering and in Vahidi et al. (2006) for dynamic selection of edges. Also, other synthesis settings that directly influence the computational efficiency, such as the size of the BDD operation cache, can be evaluated and improved using BDD-based metrics. Finally, since certain methods perform well for some models, but poorly for others, it can be investigated whether these cases can be recognized prior to performing synthesis, to make a selection of methods that are likely to perform well.

Chapter 6

Conclusions

This chapter contains the concluding remarks of this thesis. First, answers are provided to the research questions that are posed in Chapter 1. Next, it is discussed how the contributions presented in this thesis aid ease of use, applicability, and efficiency in synthesis-based engineering. Possible directions for future work are suggested.

6.1 Answers to the research questions

Three research questions are formulated in Section 1.2. The questions are answered below.

Research question 1

How can supervisory control theory be efficiently applied to systems that evolve over time?

In this thesis, it is investigated for supervisor synthesis and supervisor localization how to efficiently apply them to a system that evolves over time, respectively in Chapter 2 and 3. For these approaches, transformational algorithms are introduced that not only consider the current system instance, but also use the knowledge from a previous instance. So, when a system evolves to a new instance, one may take the algorithmic results computed based on a previous instance, and transform those into the according results for the new instance, instead of performing the algorithmic computations once again completely from scratch. For supervisor synthesis, the transformational method is shown to be less efficient than using the basic (nontransformational) method. Conversely, for supervisor localization, the transformational method is shown to be more efficient than the basic method. Therefore, for a system that evolves over time, supervisor synthesis can be efficiently applied by using the basic method. Supervisor localization can be efficiently applied by using the transformational method.

Research question 2

How can supervisory control theory be efficiently applied to product families with dynamic reconfiguration?

Supervisory control theory can be efficiently applied to product families by incorporating feature models in automata models. Presence of features can be modeled, and restricted to the constraints expressed by the feature models. To allow dynamic reconfiguration, feature presence can change during runtime. Behavior of each feature is modeled, and made dependent on its presence. Requirements are formulated that take the presence of features into account. The resulting model is shown to be suitable for supervisor synthesis. The supervisor obtained by synthesis can control each valid configuration, as well as the transition phase during reconfiguration. The method is explained in Chapter 4. By a case study, the method is shown to be applicable for industrial product lines.

Research question 3

How can BDDs be efficiently applied in symbolic supervisor synthesis in order to reduce the computational effort?

The computational effort of symbolic supervisor synthesis can generally be reduced by: (1) applying DCSH+FORCE+SW variable ordering heuristic, (2) applying reverse-model edge order, and (3) enforcing state exclusion requirement by restricting edge guards prior to synthesis. In Chapter 5 these methods are presented, and supported by elaborate experimental evaluation. The evaluation shows that when these methods are applied, generally peak BDD sizes are smaller and the number of BDD operations are lower compared to when the methods are not used. The first method generally avoids high effort outliers, the second and third reduce memory usage and reduce the computation time even more significantly.

6.2 Contribution to synthesis-based engineering and future work

For each part of this thesis we discuss how it contributes to synthesis-based engineering, and mention some possible directions for future work.

6.2.1 Transformational approaches in supervisory control

In Chapters 2 and 3, transformational approaches in supervisory control are discussed. This part contributes to the efficiency and ease of use of supervisory control theory, by investigating ways to reduce the computational effort of algorithmic calculations.

Although it was not investigated in this work, possibly the “transformational way of thinking” may lead to more benefits in synthesis-based engineering in the future. For example, the concept of model deltas is now only applied to input models, but it might be interesting to also evaluate model deltas for results. E.g., say a supervisor has been synthesized for some model, afterward a new requirement is added to the model, and a new supervisor is (transformationally) synthesized. Then, it may be interesting to compute the model delta between both supervisors to compare in exactly what situations the new requirement influences the behavior permitted by supervisor.

6.2.2 Supervisory control for product lines

Chapter 4 contributes to the applicability and ease of use of supervisory control theory by presenting a framework for engineering of supervisory controllers for product lines with dynamic feature configuration. By using this framework, supervisory controllers do not have to be produced one-by-one for each product instance, and also the reconfiguration is directly handled. Presumably, by applying this framework, supervisory controllers for product lines can be engineered more efficiently than doing so without using the product line engineering approach. A case study evaluation on the improvement of engineering efficiency remains future work.

Even though in Section 4.8, it is demonstrated that the method can handle an industrial-sized system, eventually scalability issues may arise when considering larger systems. This is expected, because a single supervisor is generated that can control all unique configurations, and the number of unique configuration rapidly grows when features are added to the system. Since the models that are considered already have a structured layout based on the feature models, it would make sense to investigate modular synthesis techniques that exploit this structure. Some initial work in this direction is presented in Wetzels (2021).

6.2.3 Efficient symbolic supervisor synthesis

Chapter 5 directly contributes to the efficiency of symbolic supervisor synthesis. Therefore, this chapter also improves the ease of use, and makes synthesis an available option for more complex systems. The experimental evaluations indicate that much improvement can still be made on the efficiency of symbolic supervisor synthesis.

There are several directions that this research can be extended. In this thesis, only static ordering of variables and edges is considered. In the future, also dynamic ordering algorithms may be investigated. Also, other synthesis settings that directly influence the computational efficiency, such as the size of the BDD operation cache, may be improved. Furthermore, it may be interesting to investigate some meta-heuristic that selects good heuristic (ordering) algorithms to be applied, depending on characteristics of the model.

References

- Akers 1978. Akers, Sheldon B.: Binary decision diagrams. In: *Transactions on Computers* 27 (1978), Nr. 6, p. 509–516. IEEE. – URL <https://doi.org/10.1109/tc.1978.1675141>
- Åkesson et al. 2006. Åkesson, Knut ; Fabian, Martin ; Flordal, Hugo ; Malik, Robi: Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. In: *Proceedings of the 8th International Workshop on Discrete Event Systems*, IEEE, 2006, p. 384–385. – URL <https://doi.org/10.1109/wodes.2006.382401>
- Aloul et al. 2003. Aloul, Fadi A. ; Markov, Igor L. ; Sakallah, Karem A.: FORCE: a fast and easy-to-implement variable-ordering heuristic. In: *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, ACM Press, 2003, p. 116–119. – URL <https://doi.org/10.1145/764808.764839>
- Aziz et al. 1994. Aziz, Adnan ; Taşiran, Serdar ; Brayton, Robert K.: BDD variable ordering for interacting finite state machines. In: *Proceedings of the 31st Annual Conference on Design Automation*, ACM Press, 1994, p. 283–288. – URL <https://doi.org/10.1145/196244.196379>
- Baier and Katoen 2008. Baier, Christel ; Katoen, Joost-Pieter: *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008. – URL <https://mitpress.mit.edu/9780262026499/principles-of-model-checking/>. – ISBN 9780262026499
- Basile 2019. Basile, Davide: Applying supervisory control synthesis to priced featured automata and energy problems. In: *International Journal on Software Tools for Technology Transfer* 21 (2019), Nr. 6, p. 679–689. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10009-019-00533-3>
- Basile et al. 2020. Basile, Davide ; Beek, Maurice H. ter ; Degano, Pierpaolo ; Legay, Axel ; Ferrari, Gian-Luigi ; Gnesi, Stefania ; Giandomenico, Felicita D.: Controller synthesis of service contracts with variability. In: *Science of Computer Programming* 187 (2020), p. 102344. Elsevier BV. – URL <https://doi.org/10.1016/j.scico.2019.102344>
- Basile et al. 2019. Basile, Francesco ; Chiacchio, Pasquale ; Di Marino, Emiliano: An auction-based approach to control automated warehouses using smart vehicles. In: *Control Engineering Practice* 90 (2019), sep, p. 285–300. Elsevier BV. – URL <https://doi.org/10.1016/j.conengprac.2019.06.005>
- van Beek et al. 2014. Beek, Dirk A. van ; Fokkink, Wan J. ; Hendriks, Dennis ; Hofkamp, Albert T. ; Markovski, Jasen ; Mortel-Fronczak, Joanna M. van de ; Reniers, Michel A.: CIF 3: model-based engineering of supervisory controllers. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2014, p. 575–580. – URL https://doi.org/10.1007/978-3-642-54862-8_48
- ter Beek and de Vink 2014. Beek, Maurice H. ter ; Vink, Erik P. de: Towards modular verification of software product lines with mCRL2. In: *Leveraging Applications of Formal Methods, Verification*

- and Validation. Technologies for Mastering Change.* Springer Berlin Heidelberg, 2014, p. 368–385. – URL https://doi.org/10.1007/978-3-662-45234-9_26
- ter Beek et al. 2016. Beek, Maurice H. ter ; Reniers, Michel A. ; Vink, Erik P. de: Supervisory controller synthesis for product lines using CIF 3. In: *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques.* Springer International Publishing, 2016, p. 856–873. – URL https://doi.org/10.1007/978-3-319-47166-2_59
- Benavides et al. 2005. Benavides, David ; Trinidad, Pablo ; Ruiz-Cortés, Antonio: Automated reasoning on feature models. In: *Notes on Numerical Fluid Mechanics and Multidisciplinary Design.* Springer International Publishing, 2005, p. 491–503. – URL https://doi.org/10.1007/11431855_34
- Benavides et al. 2010. Benavides, David ; Segura, Sergio ; Ruiz-Cortés, Antonio: Automated analysis of feature models 20 years later: a literature review. In: *Information Systems* 35 (2010), Nr. 6, p. 615–636. Elsevier BV. – URL <https://doi.org/10.1016/j.is.2010.01.001>
- Bosch 2000. Bosch, Jan: *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.* ACM Press, 2000. – URL <https://dl.acm.org/doi/10.5555/339362>. – ISBN 978-0-201-67494-1
- Broadfoot and Hopcroft 2003. Broadfoot, Guy H. ; Hopcroft, Philippa J.: *Analytical Software Design.* Verum Consultants B.V., 2003. – Research Report. – URL https://researchgate.net/publication/228752794_Analytical_Software_Design
- Browning 2016. Browning, Tyson: Design structure matrix extensions and innovations: a survey and new opportunities. In: *Transactions on Engineering Management* 63 (2016), Nr. 1, p. 27–52. IEEE. – URL <https://doi.org/10.1109/tem.2015.2491283>
- Bryant 1992. Bryant, Randal E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. In: *ACM Computing Surveys* 24 (1992), Nr. 3, p. 293–318. ACM Press. – URL <https://doi.org/10.1145/136035.136043>
- Burch et al. 1994. Burch, Jerry R. ; Clarke, Edmund M. ; Long, David E. ; McMillan, Kenneth L. ; Dill, David L.: Symbolic model checking for sequential circuit verification. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13 (1994), Nr. 4, p. 401–424. IEEE. – URL <https://doi.org/10.1109/43.275352>
- Cabodi et al. 1999. Cabodi, Gianpiero ; Camurati, Paolo E. ; Quer, Stefano: Improving the efficiency of BDD-based operators by means of partitioning. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18 (1999), Nr. 5, p. 545–556. IEEE. – URL <https://doi.org/10.1109/43.759068>
- Cai and Wonham 2010. Cai, Kai ; Wonham, W. Murray: Supervisor localization: a top-down approach to distributed control of discrete-event systems. In: *Transactions on Automatic Control* 55 (2010), Nr. 3, p. 605–618. IEEE. – URL <https://doi.org/10.1109/tac.2009.2039237>
- Cai and Wonham 2014. Cai, Kai ; Wonham, W. Murray: New results on supervisor localization, with case studies. In: *Discrete Event Dynamic Systems* 25 (2014), p. 203–226. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-014-0194-6>
- Cao et al. 2002. Cao, Xiren ; Cohen, Guy ; Giua, Alessandro ; Wonham, W. Murray ; Schuppen, Jan H. van: Unity in diversity, diversity in unity: retrospective and prospective views on control of discrete event systems. In: *Discrete Event Dynamic Systems* 12 (2002), p. 253–264. Springer Science and Business Media LLC. – URL <https://doi.org/10.1023/A:1015617431563>
- Cassandras and Lafortune 2021. Cassandras, Christos G. ; Lafortune, Stéphane: *Introduction to Discrete Event Systems.* 3rd. Springer Nature Switzerland, 2021. – URL <https://doi.org/10.1007/978-3-030-72274-6>. – ISBN 978-3-030-72272-2
- Čengić and Åkesson 2008. Čengić, Goran ; Åkesson, Knut: A control software development method using IEC 61499 function blocks, simulation and formal verification. In: *Proceedings of the 20th IFAC World Congress.* Elsevier BV, 2008, p. 22–27. – URL <https://doi.org/10.3182/20080706-5-kr-1001.00003>

- Chaki and Gurfinkel 2018. Chaki, Sagar ; Gurfinkel, Arie: BDD-based symbolic model checking. In: *Handbook of Model Checking*. Springer International Publishing, 2018, p. 219–245. – URL https://doi.org/10.1007/978-3-319-10575-8_8
- Ciardo and Siminiceanu 2002. Ciardo, Gianfranco ; Siminiceanu, Radu I.: Using edge-valued decision diagrams for symbolic generation of shortest paths. In: *Formal Methods in Computer-Aided Design*. Springer Berlin Heidelberg, 2002, p. 256–273. – URL https://doi.org/10.1007/3-540-36126-x_16
- Classen et al. 2010. Classen, Andreas ; Heymans, Patrick ; Schobbens, Pierre-Yves ; Legay, Axel ; Raskin, Jean-Francois: Model checking lots of systems. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ACM Press, 2010, p. 335–344. – URL <https://doi.org/10.1145/1806799.1806850>
- Classen et al. 2013. Classen, Andreas ; Cordy, Maxime ; Schobbens, Pierre-Yves ; Heymans, Patrick ; Legay, Axel ; Raskin, Jean-Francois: Featured transition systems: foundations for verifying variability-intensive systems and their application to LTL model checking. In: *Transactions on Software Engineering* 39 (2013), Nr. 8, p. 1069–1089. IEEE. – URL <https://doi.org/10.1109/tse.2012.86>
- Correll et al. 2009. Correll, Nikolaus ; Arechiga, Nikos ; Bolger, Adrienne ; Bollini, Mario ; Charrow, Ben ; Clayton, Adam ; Dominguez, Felipe ; Donahue, Kenneth ; Dyar, Samuel ; Johnson, Luke ; Liu, Huan ; Patrikalakis, Alexander ; Robertson, Timothy ; Smith, Jeremy ; Soltero, Daniel ; Tanner, Melissa ; White, Lauren ; Rus, Daniela: Building a distributed robot garden. In: *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2009, p. 1509–1516. – URL <https://doi.org/10.1109/iros.2009.5354261>
- Cuthill and McKee 1969. Cuthill, Elizabeth H. ; McKee, James: Reducing the bandwidth of sparse symmetric matrices. In: *Proceedings of the 1969 24th national conference*, ACM Press, 1969, p. 157–172. – URL <https://doi.org/10.1145/800195.805928>
- Czarnecki and Eisenecker 2000. Czarnecki, Krzysztof ; Eisenecker, Ulrich: *Generative Programming: Methods, Tools, and Applications*. ACM Press, 2000. – URL <https://dl.acm.org/doi/10.5555/345203>. – ISBN 978-0-201-30977-5
- D’Andrea and Wurman 2008. D’Andrea, Raffaello ; Wurman, Peter: Future challenges of coordinating hundreds of autonomous vehicles in distribution facilities. In: *Proceedings of the 2008 IEEE International Conference on Technologies for Practical Robot Applications*, IEEE, 2008, p. 80–83. – URL <https://doi.org/10.1109/tepra.2008.4686677>
- Demirci 2021. Demirci, Seref: The requirements for automation systems based on Boeing 737 MAX crashes. In: *Aircraft Engineering and Aerospace Technology* 94 (2021), Nr. 2, p. 140–153. Emerald. – URL <https://doi.org/10.1108/aeat-03-2021-0069>
- Fabian and Hellgren 1998. Fabian, Martin ; Hellgren, Anders: PLC-based implementation of supervisory control for discrete event systems. In: *Proceedings of the 37th IEEE Conference on Decision and Control* Volume 4, IEEE, 1998, p. 3305–3310. – URL <https://doi.org/10.1109/cdc.1998.758209>
- Fei et al. 2013. Fei, Zhennan ; Miremadi, Sajed ; Åkesson, Knut ; Lennartson, Bengt: Symbolic state-space exploration and guard generation in supervisory control theory. In: *Communications in Computer and Information Science* Volume 271. Springer Berlin Heidelberg, 2013, p. 161–175. – URL https://doi.org/10.1007/978-3-642-29966-7_11
- Fei et al. 2014. Fei, Zhennan ; Miremadi, Sajed ; Åkesson, Knut ; Lennartson, Bengt: Efficient symbolic supervisor synthesis for extended finite automata. In: *Transactions on Control Systems Technology* 22 (2014), Nr. 6, p. 2368–2375. IEEE. – URL <https://doi.org/10.1109/tcst.2014.2303134>
- Feng et al. 2008. Feng, Lei ; Cai, Kai ; Wonham, W. Murray: A structural approach to the non-blocking supervisory control of discrete-event systems. In: *The International Journal of Advanced Manufacturing Technology* 41 (2008), Nr. 11-12, p. 1152–1168. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s00170-008-1555-9>
- Flordal et al. 2007. Flordal, Hugo ; Malik, Robi ; Fabian, Martin ; Åkesson, Knut: Compositional synthesis of maximally permissive supervisors using supervision equivalence. In: *Discrete Event*

- Dynamic Systems* 17 (2007), Nr. 4, p. 475–504. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-007-0018-z>
- Fokkink et al. 2023. Fokkink, Wan J. ; Goorden, Martijn A. ; Hendriks, Dennis ; Beek, Dirk A. van ; Hofkamp, Albert T. ; Reijnen, Ferdie F. H. ; Etman, Pascal F. P. ; Moormann, Lars ; Mortel-Fronczak, Joanna M. van de ; Reniers, Michel A. ; Rooda, Jacobus E. ; Sanden, Bram J. van der ; Schiffelers, Ramon R. H. ; Thuijsman, Sander B. ; Verbakel, Jeroen J. ; Vogel, Han A.: Eclipse ESCET™: the Eclipse supervisory control engineering toolkit. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2023, p. 44–52. – URL https://doi.org/10.1007/978-3-031-30820-8_6
- Forschelen et al. 2012. Forschelen, Stefan T. J. ; Mortel-Fronczak, Joanna M. van de ; Su, Rong ; Rooda, Jacobus E.: Application of supervisory control theory to theme park vehicles. In: *Discrete Event Dynamic Systems* 22 (2012), Nr. 4, p. 511–540. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-012-0130-6>
- Fragal et al. 2017. Fragal, Vanderson H. ; Simao, Adenildo ; Mousavi, Mohammad R.: Validated test models for software product lines: featured finite state machines. In: *Formal Aspects of Component Software*. Springer International Publishing, 2017, p. 210–227. – URL https://doi.org/10.1007/978-3-319-57666-4_13
- Ghallab et al. 2016. Ghallab, Malik ; Nau, Dana ; Traverso, Paolo: *Automated Planning and Acting*. Cambridge University Press, 2016. – URL <https://doi.org/10.1017/cbo9781139583923>. – ISBN 978-1-107-03727-4
- Gharsellaoui et al. 2021. Gharsellaoui, Hamza ; Maazoun, Jihen ; Bouassida, Nadia ; Ahmed, Samir B. ; Ben-Abdallah, Hanene: A software product line design based approach for real-time scheduling of reconfigurable embedded systems. In: *Computers in Human Behavior* 115 (2021), p. 104925. Elsevier BV. – URL <https://doi.org/10.1016/j.chb.2017.04.026>
- Goorden et al. 2020. Goorden, Martijn A. ; Mortel-Fronczak, Joanna van de ; Reniers, Michel A. ; Fokkink, Wan J. ; Rooda, Jacobus E.: Structuring multilevel discrete-event systems with dependence structure matrices. In: *Transactions on Automatic Control* 65 (2020), Nr. 4, p. 1625–1639. IEEE. – URL <https://doi.org/10.1109/tac.2019.2928119>
- Guerin et al. 2012. Guerin, François ; Lefebvre, Dimitri ; Loisel, Vincent: Supervisory control design for systems of multiple sources of energy. In: *Control Engineering Practice* 20 (2012), Nr. 12, p. 1310–1324. Elsevier BV. – URL <https://doi.org/10.1016/j.conengprac.2012.07.006>
- Heymans et al. 2008. Heymans, Patrick ; Schobbens, Pierre-Yves ; Trigaux, Jean-Cristophe ; Bontemp, Yves ; Matulevicius, Raimundas ; Classen, Andreas: Evaluating formal properties of feature diagram languages. In: *IET Software* 2 (2008), Nr. 3, p. 281–302. Institution of Engineering and Technology. – URL <https://doi.org/10.1049/iet-sen:20070055>
- Kahraman and Cleophas 2021. Kahraman, Gökhan ; Cleophas, Loek: Automated derivation of variants in manufacturing systems design. In: *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B*, ACM Press, 2021, p. 45–50. – URL <https://doi.org/10.1145/3461002.3473942>
- Khan 2013. Khan, Yasir I.: Optimizing verification of structurally evolving algebraic Petri nets. In: *Software Engineering for Resilient Systems*. Springer Berlin Heidelberg, 2013, p. 64–78. – URL https://doi.org/10.1007/978-3-642-40894-6_6
- Kleinberg and Tardos 2005. Kleinberg, Jon ; Tardos, Eva: *Algorithm Design*. ACM Press, 2005. – URL <https://dl.acm.org/doi/10.5555/1051910>. – ISBN 978-0-321-29535-4
- Knuth 1976. Knuth, Donald E.: Big omicron and big omega and big theta. In: *ACM SIGACT News* 8 (1976), apr, Nr. 2, p. 18–24. ACM Press. – URL <https://doi.org/10.1145/1008328.1008329>
- Kogekar et al. 2004. Kogekar, Sachin ; Neema, Sandeep ; Eames, Brandon ; Koutsoukos, Xenofon ; Ledeczi, Akos ; Maroti, Miklos: Constraint-guided dynamic reconfiguration in sensor networks. In: *Proceedings of the 3rd international symposium on Information processing in sensor networks*, ACM Press, 2004, p. 379–387. – URL <https://doi.org/10.1145/984622.984677>

- Korssen et al. 2018. Korssen, Tim ; Dolk, Victor ; Mortel-Fronczak, Joanna M. van de ; Reniers, Michel A. ; Heemels, Maurice P. M. H.: Systematic model-based design and implementation of supervisors for advanced driver assistance systems. In: *Transactions on Intelligent Transportation Systems* 19 (2018), Nr. 2, p. 533–544. IEEE. – URL <https://doi.org/10.1109/tits.2017.2776354>
- Krook et al. 2020. Krook, Jonas ; Kianfar, Roozbeh ; Fabian, Martin: Formal synthesis of safe stop tactical planners for an automated vehicle. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2020, p. 445–452. – URL <https://doi.org/10.1016/j.ifacol.2021.04.059>
- Lachmann et al. 2016. Lachmann, Remo ; Schulze, Sandro ; Nieke, Manuel ; Seidl, Christoph ; Schaefer, Ina: System-level test case prioritization using machine learning. In: *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications*, IEEE, 2016, p. 361–368. – URL <https://doi.org/10.1109/icmla.2016.0065>
- Larman and Basili 2003. Larman, Craig ; Basili, Victor R.: Iterative and incremental developments. a brief history. In: *Computer* 36 (2003), Nr. 6, p. 47–56. IEEE. – URL <https://doi.org/10.1109/mc.2003.1204375>
- Lee 1959. Lee, C. Y.: Representation of switching circuits by binary-decision programs. In: *The Bell System Technical Journal* 38 (1959), Nr. 4, p. 985–999. IEEE. – URL <https://doi.org/10.1002/j.1538-7305.1959.tb01585.x>
- Lehman 1996. Lehman, Meir M.: Laws of software evolution revisited. In: *Software Process Technology*. Springer Berlin Heidelberg, 1996, p. 108–124. – URL <https://doi.org/10.1007/bfb0017737>
- Li et al. 2019. Li, Wei ; Lin, Zhiyun ; Cai, Kai ; Zhou, Hanyun ; Yan, Gangfeng: Multi-objective optimal charging control of plug-in hybrid electric vehicles in power distribution systems. In: *Energies* 12 (2019), Nr. 13, p. 2563. MDPI AG. – URL <https://doi.org/10.3390/en12132563>
- Lin et al. 2019. Lin, Liyong ; Thuijsman, Sander B. ; Zhu, Yuting ; Ware, Simon ; Su, Rong ; Reniers, Michel A.: Synthesis of supremal successful normal actuator attackers on normal supervisors. In: *Proceedings of the 2019 American Control Conference*, IEEE, 2019, p. 5614–5619. – URL <https://doi.org/10.23919/acc.2019.8814712>
- Lity et al. 2013. Lity, Sascha ; Lachmann, Remo ; Lochau, Malte ; Schaefer, Ina: Delta-oriented software product line test models - the body comfort system case study. Technische Universität Braunschweig, 2013. – Research Report. – 302 p. – URL https://www.isf.cs.tu-bs.de/cms/team/lity/bcs_tubs_tech_rep_V1_4.pdf
- Lity et al. 2017. Lity, Sascha ; Al-Hajjaji, Mustafa ; Thüm, Thomas ; Schaefer, Ina: Optimizing product orders using graph algorithms for improving incremental product-line analysis. In: *Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems*, ACM Press, 2017, p. 60–67. – URL <https://doi.org/10.1145/3023956.3023961>
- Liu et al. 2021. Liu, Yangtao ; Zhang, Ruihan ; Wang, Jun ; Wang, Yan: Current and future lithium-ion battery manufacturing. In: *iScience* 24 (2021), Nr. 4, p. 102332. Elsevier BV. – URL <https://doi.org/10.1016/j.isci.2021.102332>
- Lochau et al. 2014. Lochau, Malte ; Lity, Sascha ; Lachmann, Remo ; Schaefer, Ina ; Goltz, Ursula: Delta-oriented model-based integration testing of large-scale systems. In: *Journal of Systems and Software* 91 (2014), p. 63–84. Elsevier BV. – URL <https://doi.org/10.1016/j.jss.2013.11.1096>
- Loose et al. 2018. Loose, Robin ; Sanden, Bram J. van der ; Reniers, Michel A. ; Schiffelers, Ramon R. H.: Component-wise supervisory controller synthesis in a client/server architecture. In: *Proceedings of the 14th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2018, p. 381–387. – URL <https://doi.org/10.1016/j.ifacol.2018.06.329>
- Lopes et al. 2016. Lopes, Yuri K. ; Trenkwalder, Stefan M. ; Leal, André B. ; Dodd, Tony J. ; Groß, Roderich: Supervisory control theory applied to swarm robotics. In: *Swarm Intelligence* 10 (2016), Nr. 1, p. 65–97. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s11721-016-0119-0>

- Lousberg et al. 2020. Lousberg, Sam A. J. ; Thuijsman, Sander B. ; Reniers, Michel A.: DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2020, p. 429–436. – URL <https://doi.org/10.1016/j.ifacol.2021.04.058>
- Ma and Wonham 2006. Ma, Chuan ; Wonham, W. Murray: Nonblocking supervisory control of state tree structures. In: *Transactions on Automatic Control* 51 (2006), Nr. 5, p. 782–793. IEEE. – URL <https://doi.org/10.1109/tac.2006.875030>
- Ma and Wonham 2008. Ma, Chuan ; Wonham, W. Murray: STSLib and its application to two benchmarks. In: *Proceedings of the 9th International Workshop on Discrete Event Systems*, IEEE, 2008, p. 119–124. – URL <https://doi.org/10.1109/wodes.2008.4605932>
- Malik et al. 2017. Malik, Robi ; Åkesson, Knut ; Flordal, Hugo ; Fabian, Martin: Supremica—an efficient tool for large-scale discrete event systems. In: *Proceedings of the 20th IFAC World Congress*, Elsevier BV, 2017, p. 5794–5799. – URL <https://doi.org/10.1016/j.ifacol.2017.08.427>
- Markovski et al. 2010. Markovski, Jasen ; Beek, Dirk A. van ; Theunissen, Rolf J. M. ; Jacobs, Koen G. M. ; Rooda, Jacobus E.: A state-based framework for supervisory control synthesis and verification. In: *Proceedings of the 49th IEEE Conference on Decision and Control*, IEEE, 2010, p. 3481–3486. – URL <https://doi.org/10.1109/cdc.2010.5717095>
- Meijer and van de Pol 2016. Meijer, Jeroen ; Pol, Jan C. van de: Bandwidth and wavefront reduction for static variable ordering in symbolic reachability analysis. In: *Proceedings of the 8th NASA Formal Methods Symposium*. Springer International Publishing, 2016, p. 255–271. – URL https://doi.org/10.1007/978-3-319-40648-0_20
- Meinel and Theobald 1998. Meinel, Christoph ; Theobald, Thorsten: *Algorithms and Data Structures in VLSI Design*. Springer Berlin Heidelberg, 1998. – URL <https://doi.org/10.1007/978-3-642-58940-9>. – ISBN 978-3-540-64486-6
- Minato 1996. Minato, Shin-ichi: *Binary Decision Diagrams and Applications for VLSI CAD*. Springer US, 1996. – URL <https://doi.org/10.1007/978-1-4613-1303-8>. – ISBN 978-1-4612-8558-8
- Minato 2001. Minato, Shin-ichi: Zero-suppressed BDDs and their applications. In: *International Journal on Software Tools for Technology Transfer* 3 (2001), Nr. 2, p. 156–170. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s100090100038>
- Miremadi et al. 2008. Miremadi, Sajed ; Åkesson, Knut ; Fabian, Martin ; Vahidi, Arash ; Lennartson, Bengt: Solving two supervisory control benchmark problems using Supremica. In: *Proceedings of the 9th International Workshop on Discrete Event Systems*, IEEE, 2008, p. 131–136. – URL <https://doi.org/10.1109/wodes.2008.4605934>
- Miremadi et al. 2012. Miremadi, Sajed ; Lennartson, Bengt ; Åkesson, Knut: A BDD-based approach for modeling plant and supervisor by extended finite automata. In: *Transactions on Control Systems Technology* 20 (2012), Nr. 6, p. 1421–1435. IEEE. – URL <https://doi.org/10.1109/tcst.2011.2167150>
- Miremadi and Lennartson 2016. Miremadi, Sajed ; Lennartson, Bengt: Symbolic on-the-fly synthesis in supervisory control theory. In: *Transactions on Control Systems Technology* 24 (2016), Nr. 5, p. 1705–1716. IEEE. – URL <https://doi.org/10.1109/tcst.2015.2508978>
- Montgomery and Runger 2018. Montgomery, Douglas C. ; Runger, George C.: *Applied Statistics and Probability for Engineers*. 7th. John Wiley & Sons, Inc., 2018. – URL <https://wiley.com/en-us/Applied+Statistics+and+Probability+for+Engineers,+7th+Edition-p-9781119400363>. – ISBN 978-1-119-40036-3
- Moor et al. 2008. Moor, Thomas ; Schmidt, Klaus ; Perk, Sebastian: libFAUDES - an open source C++ library for discrete event systems. In: *Proceedings of the 9th International Workshop on Discrete Event Systems*, IEEE, 2008, p. 125–130. – URL <https://doi.org/10.1109/wodes.2008.4605933>
- Moormann et al. 2020. Moormann, Lars ; Maessen, Patrick ; Goorden, Martijn A. ; Mortel-Fronczak, Joanna M. van de ; Rooda, Jacobus E.: Design of a tunnel supervisory controller using synthesis-based engineering. In: *ITA-AITES World Tunnel Congress*,

- WTC2020 and 46th General Assembly, Proceedings*, ITA Library, 2020, p. 573–578. – URL <https://library.ita-aites.org/wtc/1822-design-of-a-tunnel-supervisory-controller-using-synthesis-based-engineering.html>
- Moormann et al. 2021. Moormann, Lars ; Schouten, Jaap H. J. ; Mortel-Fronczak, Joanna M. van de ; Fokink, Wan J. ; Rooda, Jacobus E.: Synthesis and implementation of distributed supervisory controllers with communication delays. In: *Proceedings of the IEEE 17th International Conference on Automation Science and Engineering*, IEEE, 2021, p. 1268–1275. – URL <https://doi.org/10.1109/case49439.2021.9551519>
- Ouedraogo et al. 2011. Ouedraogo, Lucien ; Kumar, Ratnesh ; Malik, Robi ; Åkesson, Knut: Nonblocking and safe control of discrete-event systems modeled as extended finite automata. In: *Transactions on Automation Science and Engineering* 8 (2011), Nr. 3, p. 560–569. IEEE. – URL <https://doi.org/10.1109/tase.2011.2124457>
- Panda et al. 1994. Panda, Shipra ; Somenzi, Fabio ; Plessier, Bernard F.: Symmetry detection and dynamic variable ordering of decision diagrams. In: *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, IEEE, 1994, p. 628–631. – URL <https://dl.acm.org/doi/10.5555/191326.191598>
- Pohl et al. 2005. Pohl, Klaus ; Böckle, Günter ; Linden, Frank van der: *Software Product Line Engineering*. Springer Berlin Heidelberg, 2005. – URL <https://doi.org/10.1007/3-540-28901-1>. – ISBN 978-3-540-24372-4
- van Putten et al. 2020. Putten, Berend Jan C. van ; Sanden, Bram J. van der ; Reniers, Michel A. ; Voeten, Jeroen P. M. ; Schiffelers, Ramon R. H.: Supervisor synthesis and throughput optimization of partially-controllable manufacturing systems. In: *Discrete Event Dynamic Systems* 31 (2020), Nr. 1, p. 103–135. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-020-00325-x>
- de Queiroz and Cury 2002. Queiroz, Max H. de ; Cury, José E. R.: Synthesis and implementation of local modular supervisory control for a manufacturing cell. In: *Proceedings of the 6th International Workshop on Discrete Event Systems*, IEEE, 2002, p. 377–382. – URL <https://doi.org/10.1109/wodes.2002.1167714>
- Ramadge and Wonham 1987. Ramadge, Peter J. ; Wonham, W. Murray: Supervisory control of a class of discrete event processes. In: *SIAM Journal on Control and Optimization* 25 (1987), Nr. 1, p. 206–230. Society for Industrial & Applied Mathematics. – URL <https://doi.org/10.1137/0325013>
- Ramadge and Wonham 1989. Ramadge, Peter J. ; Wonham, W. Murray: The control of discrete event systems. In: *Proceedings of the IEEE* 77 (1989), Nr. 1, p. 81–98. IEEE. – URL <https://doi.org/10.1109/5.21072>
- Ranjan et al. 1995. Ranjan, Rajeev K. ; Aziz, Adnan ; Brayton, Robert K. ; Plessier, Bernard ; Pixley, Carl: Efficient BDD algorithms for FSM synthesis and verification. In: *International Workshop on Logic and Synthesis*, IEEE/ACM, 1995, p. 1–8. – URL <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=7d6478f6e5ffe6cb40e445008ccd50efff59fdad>
- Rawlings et al. 2014. Rawlings, Blake C. ; Christenson, Ben ; Wassick, John M. ; Ydstie, B. Erik: Supervisor synthesis to satisfy safety and reachability requirements in chemical process control. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2014, p. 195–200. – URL <https://doi.org/10.3182/20140514-3-fr-4046.00127>
- Reijnen et al. 2017. Reijnen, Ferdie F. H. ; Goorden, Martijn A. ; Mortel-Fronczak, Joanna M. van de ; Rooda, Jacobus E.: Supervisory control synthesis for a waterway lock. In: *Proceedings of the 2017 IEEE Conference on Control Technology and Applications*, IEEE, 2017, p. 1562–1568. – URL <https://doi.org/10.1109/ccta.2017.8062679>
- Reijnen et al. 2018a. Reijnen, Ferdie F. H. ; Goorden, Martijn A. ; Mortel-Fronczak, Joanna M. van de ; Reniers, Michel A. ; Rooda, Jacobus E.: Application of dependency structure matrices and multilevel synthesis to a production line. In: *Proceedings of the 2018 IEEE Conference on Control Technology and Applications*, IEEE, 2018, p. 458–464. – URL <https://doi.org/10.1109/ccta.2018.8511449>

- Reijnen et al. 2018b. Reijnen, Ferdie F. H. ; Reniers, Michel A. ; Mortel-Fronczak, Joanna M. van de ; Rooda, Jacobus E.: Structured synthesis of fault-tolerant supervisory controllers. In: *Proceedings 10th IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes*, Elsevier BV, 2018, p. 894–901. – URL <https://doi.org/10.1016/j.ifacol.2018.09.681>
- Reijnen et al. 2020. Reijnen, Ferdie F. H. ; Goorden, Martijn A. ; Mortel-Fronczak, Joanna M. van de ; Rooda, Jacobus E.: Modeling for supervisor synthesis – a lock-bridge combination case study. In: *Discrete Event Dynamic Systems* 30 (2020), Nr. 3, p. 499–532. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-020-00314-0>
- Reijnen et al. 2021. Reijnen, Ferdie F. H. ; Leliveld, Eva-Britt M. L. ; Mortel-Fronczak, Joanna M. van de ; Dinther, John van ; Rooda, Jacobus E. ; Fokkink, Wan J.: Synthesized fault-tolerant supervisory controllers, with an application to a rotating bridge. In: *Computers in Industry* 130 (2021), p. 103473. Elsevier BV. – URL <https://doi.org/10.1016/j.compind.2021.103473>
- Reniers and van de Mortel-Fronczak 2018. Reniers, Michel A. ; Mortel-Fronczak, Joanna M. van de: An engineering perspective on model-based design of supervisors. In: *Proceedings of the 14th IFAC Workshop on Discrete Event Systems* 51 (2018), Nr. 7, p. 257–264. Elsevier BV. – URL <https://doi.org/10.1016/j.ifacol.2018.06.310>
- Reniers and Thuijsman 2020. Reniers, Michel A. ; Thuijsman, Sander B.: Supervisory control for dynamic feature configuration in product lines. In: *Forum for Specification and Design Languages*, IEEE, 2020, p. 1–8. – URL <https://doi.org/10.1109/fd150818.2020.9232937>
- Rosa et al. 2020. Rosa, Marcelo ; Cury, José E. R. ; Baldissera, Fabio L.: Supervisory control in construction robotics: in the quest for scalability and permissiveness. In: *Proceedings of the 15th IFAC Workshop on Discrete Event Systems*, Elsevier BV, 2020, p. 117–122. – URL <https://doi.org/10.1016/j.ifacol.2021.04.012>
- Rosenmüller et al. 2011. Rosenmüller, Marko ; Siegmund, Norbert ; Pukall, Mario ; Apel, Sven: Tailoring dynamic software product lines. In: *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, ACM Press, 2011, p. 3–12. – URL <https://doi.org/10.1145/2047862.2047866>
- van der Sanden et al. 2015. Sanden, Bram J. van der ; Reniers, Michel A. ; Geilen, Marc C. W. ; Basten, Twan ; Jacobs, Johan H. ; Voeten, Jeroen P. M. ; Schiffelers, Ramon R. H.: Modular model-based supervisory controller design for wafer logistics in lithography machines. In: *Proceedings of the ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems*, IEEE, 2015, p. 416–425. – URL <https://doi.org/10.1109/models.2015.7338273>
- van der Sanden et al. 2021. Sanden, Bram J. van der ; Blankenstein, Yuri ; Schiffelers, Ramon R. H. ; Voeten, Jeroen P. M.: LSAT: specification and analysis of product logistics in flexible manufacturing systems. In: *Proceedings of the IEEE 17th International Conference on Automation Science and Engineering*, IEEE, 2021, p. 1–8. – URL <https://doi.org/10.1109/case49439.2021.9551412>
- Schaefer et al. 2012. Schaefer, Ina ; Rabiser, Rick ; Clarke, Dave ; Bettini, Lorenzo ; Benavides, David ; Botterweck, Goetz ; Pathak, Animesh ; Trujillo, Salvador ; Vilella, Karina: Software diversity: state of the art and perspectives. In: *International Journal on Software Tools for Technology Transfer* 14 (2012), Nr. 5, p. 477–495. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10009-012-0253-y>
- van der Schriek 2018. Schriek, Yorick I. C. van der: *Evaluation of supervisory control theory based on requirement evolution of LOPW*, Eindhoven University of Technology, Master thesis, 2018. – URL <https://research.tue.nl/en/studentTheses/evaluation-of-supervisory-control-theory-based-on-requirement-evo>
- Selvaraj et al. 2022. Selvaraj, Yuvaraj ; Farooqui, Ashfaq ; Panahandeh, Ghazaleh ; Ahrendt, Wolfgang ; Fabian, Martin: Automatically learning formal models from autonomous driving software. In: *Electronics* 11 (2022), Nr. 4, p. 643. MDPI AG. – URL <https://doi.org/10.3390/electronics11040643>

- Sharifloo et al. 2016. Sharifloo, Amir M. ; Metzger, Andreas ; Quinton, Clément ; Baresi, Luciano ; Pohl, Klaus: Learning and evolution in dynamic software product lines. In: *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ACM, 2016. – URL <https://doi.org/10.1145/2897053.2897058>
- Shen et al. 2011. Shen, Liwei ; Peng, Xin ; Liu, Jindu ; Zhao, Wenyun: Towards feature-oriented variability reconfiguration in dynamic software product lines. In: *Top Productivity through Software Reuse*. Springer Berlin Heidelberg, 2011, p. 52–68. – URL https://doi.org/10.1007/978-3-642-21347-2_5
- Shokry and Babar 2008. Shokry, Hesham ; Babar, Muhammad Ali: Dynamic software product line architectures using service-based computing for automotive systems. In: *Proceedings of the 12th Conference on Software Product Lines*, University of Limerick, 2008, p. 1–6. – URL <https://hdl.handle.net/10344/1897>
- Siminiceanu and Ciardo 2006. Siminiceanu, Radu I. ; Ciardo, Gianfranco: New metrics for static variable ordering in decision diagrams. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2006, p. 90–104. – URL https://doi.org/10.1007/11691372_6
- Sköldstam et al. 2007. Sköldstam, Markus ; Åkesson, Knut ; Fabian, Martin: Modeling of discrete event systems using finite automata with variables. In: *Proceedings of the 46th IEEE Conference on Decision and Control*, IEEE, 2007, p. 3387–3392. – URL <https://doi.org/10.1109/cdc.2007.4434894>
- Sloan 1989. Sloan, Scott W.: A FORTRAN program for profile and waveform reduction. In: *International Journal for Numerical Methods in Engineering* 28 (1989), Nr. 11, p. 2651–2679. John Wiley & Sons, Inc.. – URL <https://doi.org/10.1002/nme.1620281111>
- de Smet et al. 2018. Smet, Marc D. de ; Naus, Gerrit J. L. ; Faridpooya, Koorosh ; Mura, Marco: Robotic-assisted surgery in ophthalmology. In: *Current Opinion in Ophthalmology* 29 (2018), Nr. 3, p. 248–253. Ovid Technologies (Wolters Kluwer Health). – URL <https://doi.org/10.1097/icu.0000000000000476>
- Somenzi 1999. Somenzi, Fabio: Binary decision diagrams. In: *The VLSI Handbook*. CRC Press, 1999, p. 680–694. – URL <https://doi.org/10.1201/9781420049671-29>
- Song and Leduc 2006. Song, Raoguang ; Leduc, Ryan J.: Symbolic synthesis and verification of hierarchical interface-based supervisory control. In: *Proceedings of the 8th IFAC Workshop on Discrete Event Systems*, IEEE, 2006, p. 419–426. – URL <https://doi.org/10.1109/wodes.2006.382510>
- Su and Wonham 2004. Su, Rong ; Wonham, W. Murray: Supervisor reduction for discrete-event systems. In: *Discrete Event Dynamic Systems* 14 (2004), Nr. 1, p. 31–53. Springer Science and Business Media LLC. – URL <https://doi.org/10.1023/b:disc.0000005009.40749.b6>
- Su et al. 2010. Su, Rong ; Schuppen, Jan H. van ; Rooda, Jacobus E.: Aggregative synthesis of distributed supervisors based on automaton abstraction. In: *Transactions on Automatic Control* 55 (2010), Nr. 7, p. 1627–1640. IEEE. – URL <https://doi.org/10.1109/tac.2010.2042342>
- Theunissen et al. 2014. Theunissen, Rolf J. M. ; Petreczky, Mihaly ; Schiffelers, Ramon R. H. ; Beek, Dirk A. van ; Rooda, Jacobus E.: Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner. In: *Transactions on Automation Science and Engineering* 11 (2014), Nr. 1, p. 20–32. IEEE. – URL <https://doi.org/10.1109/tase.2013.2279692>
- Thuijsman et al. 2019. Thuijsman, Sander B. ; Hendriks, Dennis ; Theunissen, Rolf J. M. ; Reniers, Michel A. ; Schiffelers, Ramon R. H.: Computational effort of BDD-based supervisor synthesis of extended finite automata. In: *Proceedings of the IEEE 15th International Conference on Automation Science and Engineering*, IEEE, 2019, p. 486–493. – URL <https://doi.org/10.1109/coase.2019.8843327>
- Thuijsman and Reniers 2020. Thuijsman, Sander B. ; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Proceedings of the 15th IFAC Workshop on Discrete*

- Event Systems*, Elsevier BV, 2020, p. 309–316. – URL <https://doi.org/10.1016/j.ifacol.2021.04.030>
- Thuijsman et al. 2021. Thuijsman, Sander B. ; Reniers, Michel A. ; Hendriks, Dennis: Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis. In: *Proceedings of the IEEE 17th International Conference on Automation Science and Engineering*, IEEE, 2021, p. 777–783. – URL <https://doi.org/10.1109/case49439.2021.9551593>
- Thuijsman and Reniers 2022. Thuijsman, Sander B. ; Reniers, Michel A.: Transformational supervisor synthesis for evolving systems. In: *Discrete Event Dynamic Systems* 32 (2022), Nr. 2, p. 317–358. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-021-00354-0>. – Note: A Correction to this article is in press.
- Thuijsman et al. 2023a. Thuijsman, Sander B. ; Cai, Kai ; Reniers, Michel A.: Transformational supervisor localization. In: *IEEE Control Systems Letters* 7 (2023), p. 1682–1687. IEEE. – URL <https://doi.org/10.1109/LCSYS.2023.3278248>
- Thuijsman et al. 2023b. Thuijsman, Sander B. ; Kahraman, Gökhan ; Mohamadkhani, Alireza ; Timmers, Ferry ; Cleophas, Loek G. W. A. ; Geilen, Marc C. W. ; Groote, Jan Friso ; Reniers, Michel A. ; Schiffelers, Ramon R. H. ; Voeten, Jeroen P. M.: Tool interoperability for model-based systems engineering. In: *arXiv*, Cornell University Library, 2023. – URL <https://arxiv.org/abs/2302.03503>
- Thuijsman and Reniers 2023. Thuijsman, Sander B. ; Reniers, Michel A.: Supervisory control for dynamic feature configuration in product lines. In: *Transactions on Embedded Computing Systems* (2023). ACM Press. – URL <https://doi.org/10.1145/3579644>. – In press.
- Tijssse Claase 2020. Tijssse Claase, Ramon G.: *Symbolic transformational supervisor synthesis*, Eindhoven University of Technology, Master thesis, 2020. – URL <https://research.tue.nl/en/studentTheses/symbolic-transformational-supervisor-synthesis>
- Tuitert 2017. Tuitert, Mark: *Supervisory controller synthesis for dynamic software product lines*, Eindhoven University of Technology, Master thesis, 2017. – URL https://michelreniers.files.wordpress.com/2018/07/mfp_m_tuitert.pdf
- Vahidi et al. 2006. Vahidi, Arash ; Fabian, Martin ; Lennartson, Bengt: Efficient supervisory synthesis of large systems. In: *Control Engineering Practice* 14 (2006), Nr. 10, p. 1157–1167. Elsevier BV. – URL <https://doi.org/10.1016/j.conengprac.2006.02.013>
- Verbakel et al. 2021. Verbakel, Jeroen J. ; Vos de Wael, Erik M. E. W. ; Mortel-Fronczak, Joanna M. van de ; Fokkink, Wan J. ; Rooda, Jacobus E.: A configurator for supervisory controllers of roadside systems. In: *Proceedings of the IEEE 17th International Conference on Automation Science and Engineering*, IEEE, 2021, p. 784–791. – URL <https://doi.org/10.1109/case49439.2021.9551485>
- Vilela and Hill 2022. Vilela, Juliana N. ; Hill, Richard: Hierarchical planning in a supervisory control context with compositional abstraction. In: *Discrete Event Dynamic Systems* 32 (2022), Nr. 1, p. 89–113. Springer Science and Business Media LLC. – URL <https://doi.org/10.1007/s10626-021-00349-x>
- Vos 2020. Vos, Zowi: *Efficient supervisor synthesis for feature models*, Eindhoven University of Technology, Master thesis, 2020. – URL <https://research.tue.nl/en/studentTheses/initialization-and-termination-of-flexible-manufacturing-systems>
- Ware and Su 2016. Ware, Simon ; Su, Rong: Time optimal synthesis based upon sequential abstraction and its application to cluster tools. In: *Transactions on Automation Science and Engineering* 14 (2016), Nr. 2, p. 772–784. IEEE. – URL <https://doi.org/10.1109/tase.2016.2613911>
- Wetzels 2021. Wetzels, Bart H. J.: *Efficient supervisor synthesis for feature models*, Eindhoven University of Technology, Master thesis, 2021. – URL <https://research.tue.nl/en/studentTheses/efficient-supervisor-synthesis-for-feature-models>

- Wonham et al. 2018. Wonham, W. Murray ; Cai, Kai ; Rudie, Karen: Supervisory control of discrete-event systems: a brief history. In: *Annual Reviews in Control* 45 (2018), p. 250–256. Elsevier BV. – URL <https://doi.org/10.1016/j.arcontrol.2018.03.002>
- Wonham and Cai 2019. Wonham, W. Murray ; Cai, Kai: *Supervisory Control of Discrete-Event Systems*. Springer International Publishing, 2019. – URL <https://doi.org/10.1007/978-3-319-77452-7>. – ISBN 978-3-319-77451-0
- Yoo et al. 2007. Yoo, Seong-eun ; Kim, Jae eon ; Kim, Taehong ; Ahn, Sungjin ; Sung, Jongwoo ; Kim, Daeyoung: A2S: automated agriculture system based on WSN. In: *Proceedings of the IEEE International Symposium on Consumer Electronics*, IEEE, 2007, p. 1–4. – URL <https://doi.org/10.1109/isce.2007.4382216>
- Ziller and Schneider 2003. Ziller, Roberto ; Schneider, Klaus: Reducing complexity of supervisor synthesis. In: *Proceedings of the 2nd IFAC Conference on Control Systems Design* (2003), p. 183–191. Elsevier BV. – URL [https://doi.org/10.1016/s1474-6670\(17\)34666-9](https://doi.org/10.1016/s1474-6670(17)34666-9)

Appendix A

Proofs for atomic TSS algorithms

In this appendix, we provide the proofs of Theorem 2.2 for the atomic TSS algorithms provided in Section 2.4. Subsequently the proofs for Lemma 2.5 are provided for an added state, removed state, added event, and removed event.

Added Initial Property (Algorithm 2.5)

We denote automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, $A' = (X, \Sigma, \longrightarrow, X_0 \cup \{x^\delta\}, X_m)$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSAIP}(A, Y, G, x^\delta)$.

- For any $(X_0, X'_0) \subseteq X \times X$ it holds that $G = G'$ when computing $(Y, G) = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X_0, X_m)$, $(Y', G') = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X'_0, X_m)$, as the initial state does not influence the computation of G . We also observe that for all switchcases in Algorithm 2.5, $\hat{G} = G$ is computed. It follows that $\hat{G} = G'$.
- Y' are all reachable states in G' . Since we have proven that $\hat{G} = G'$, it suffices to prove that \hat{Y} are all reachable states in \hat{G} to show that $\hat{Y} = Y'$.
 - In case that x^δ is in Y , the state x^δ will already have been found in the FRS (line 8 Algorithm 2.2), so in this case $\hat{Y} = Y = Y'$, which is also found by Algorithm 2.5.
 - In case that x^δ is in $X \setminus G$, it will also be in $X \setminus G'$. As x^δ is not in G' , the change of initial property cannot influence the reachable part of G' , so $Y = Y'$. This is also found in Algorithm 2.5. So $\hat{Y} = Y = Y'$.
 - In case that x^δ is in $G \setminus Y$, for the supervisor synthesis of A' , the reachable part is determined by $Y = \text{FRS}(G', \Sigma, \longrightarrow, X'_0)$, where we know that $G' = G$ and $X'_0 = X_0 \cup \{x^\delta\}$. \hat{Y} is calculated by $\text{FRS}(G, \Sigma, \longrightarrow, Y \cup \{x^\delta\})$. The result of these FRSs is the same, as we know that all states in Y are reachable in G from X_0 from the base synthesis. So $\hat{Y} = Y'$.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $x^\delta \in X \setminus X_0$, so Theorem 2.2 holds for Algorithm 2.5. \square

Removed Initial Property (Algorithm 2.6)

Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0 \setminus \{x^\delta\}, X_m)$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSRIP}(A, Y, G, x^\delta)$.

- For any $(X_0, X'_0) \subseteq X \times X$ it holds that $G = G'$ when computing $(Y, G) = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X_0, X_m)$, $(Y', G') = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X'_0, X_m)$, as the initial state does not influence the computation of G . We also observe that for all switchcases in Algorithm 2.6, $\hat{G} = G$ is computed. It follows that $\hat{G} = G'$.
- Y' are all reachable states in G' . Since we have proven that $\hat{G} = G'$, it suffices to prove that \hat{Y} are all reachable states in \hat{G} to show that $\hat{Y} = Y'$.
 - In case that x^δ is not in Y , the state x^δ was not in the reachable part of G . Thus, the set of reachable states in G is not influenced by x^δ being initial. So in this case $\hat{Y} = Y = Y'$, which is also found by Algorithm 2.6.
 - In case that x^δ is in Y , for the supervisor synthesis of A' , the reachable part is determined by $Y = \text{FRS}(G', \Sigma, \longrightarrow, X'_0)$, where we know that $G' = G$ and $X'_0 = X_0 \setminus \{x^\delta\}$. \hat{Y} is calculated by $\text{FRS}(Y, \Sigma, \longrightarrow, X_0 \setminus \{x^\delta\})$. The result of these FRSSs is the same, as we know that all states in $G \setminus Y$ are not reachable from the base synthesis. So $\hat{Y} = Y'$.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $x^\delta \in X_0$, so Theorem 2.2 holds for Algorithm 2.6. \square

Added Marked Property (Algorithm 2.7)

Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0, X_m \cup \{x^\delta\})$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSAMP}(A, Y, G, x^\delta)$.

- In case that x^δ is in G , it was already in the maximal coreachable and controllable set of states. It will remain so after making it marked, so $G' = G$. We observe $\hat{G} = G$ is computed in Algorithm 2.7, so $\hat{G} = G'$. As the initial states did not change, and $G' = G$, the reachable part Y will remain the same. So $\hat{Y} = Y = Y'$.
- In case that x^δ is in $X \setminus G$, states in Y remain reachable, coreachable, and controllable. So, $\text{computeFixpoint}((X, \Sigma, \longrightarrow, Y \cup X_0, Y \cup X_m \cup \{x^\delta\})) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \cup \{x^\delta\}))$.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $x^\delta \in X \setminus X_m$, so Theorem 2.2 holds for Algorithm 2.7. \square

Removed Marked Property (Algorithm 2.8)

Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0, X_m \setminus \{x^\delta\})$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSRMP}(A, Y, G, x^\delta)$.

- We know that $x^\delta \in X_m$ (model delta is valid). And all states in X_m are coreachable by definition. G was the maximal controllable coreachable set to X_m . After removing x^δ as a marked state; $G' \subseteq G$ and $Y' \subseteq Y$. So $X \setminus G \subseteq X \setminus G'$. Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow, X_0, X_m \setminus \{x^\delta\})) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \setminus \{x^\delta\}))$. Thus, in case $x^\delta \in Y$, then $\hat{Y} = Y'$ and $\hat{G} = G'$.
- In case that x^δ is in $G \setminus Y$, all states in Y are coreachable and controllable for $X_m \setminus \{x^\delta\}$, as x^δ is not reachable from Y , otherwise it would be contained in Y . Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow, X_0, (Y \cup X_m) \setminus \{x^\delta\})) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \setminus \{x^\delta\}))$. So in case $x^\delta \in G \setminus Y$, $\hat{Y} = Y'$ and $\hat{G} = G'$.
- In case that $x^\delta \notin G$, x^δ must have an uncontrollable path to a non-coreachable state, as it is coreachable as a marked state, it would have been in G if it were controllable. States that uncontrollably reach x^δ were removed from G in the base synthesis. G remains the same, and consequently Y will also remain the same, so $\hat{Y} = Y = Y'$ and $\hat{G} = G = G'$, which is also found by Algorithm 2.8.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $x^\delta \in X_m$, so Theorem 2.2 holds for Algorithm 2.8. \square

Added Transition (Algorithm 2.9)

Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m)$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSAT}(A, Y, G, (x_{or}, \sigma, x_{tar}))$.

- In case $x_{or} \in Y$ and $x_{tar} \in Y$, all states in Y remain reachable, coreachable, reachable, and controllable. Also the (co-)reachability and controllability of the states in $X \setminus Y$ remains the same. So $\hat{Y} = Y = Y'$ and $\hat{G} = G = G'$.
- In case $x_{or} \in Y$ and $x_{tar} \in G \setminus Y$, the coreachability and controllability of all states remains unchanged. So $\hat{G} = G = G'$. All states in Y remain reachable, so $\text{FRS}(G, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y) = \text{FRS}(G, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, X_0)$.
- In case x_{or} in $X \setminus G$ and $x_{or} \neq x_{tar}$, then all states in Y remain (co-)reachable and controllable. Therefore, $\text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, Y \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$.

- In case $x_{or} \in G$ and $x_{tar} \in X \setminus G$. The states in $X \setminus G$ remain non-coreachable or non-controllable. In case that
 - $\sigma \in \Sigma_c$. The supervisor can disable the added transition. So $\hat{Y} = Y = Y'$ and $\hat{G} = G = G'$.
 - $\sigma \in \Sigma_u$. The supervisor cannot disable the added transition. Thus x_{or} is non-controllable. Because also states in $X \setminus G$ remain non-coreachable or non-controllable, $\text{computeFixpoint}((G, \Sigma, \longrightarrow \cap ((G \setminus \{x_{or}\}) \times \Sigma \times G), X_0 \setminus \{x_{or}\}, X_m \setminus \{x_{or}\})) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$.
- In case $x_{or} \in G \setminus Y$ and $x_{tar} \in G$ the coreachability and controllability of all states does not change; $\hat{G} = G = G'$. x_{or} is non-reachable, so the added transition does not change the reachability of any state. Thus, $\hat{G} = G = G'$.
- In case $x_{or} = x_{tar}$, the (co-)reachability and controllability of any state does not change. So $\hat{G} = G = G'$ and $\hat{Y} = Y = Y'$.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $(x_{or}, \sigma, x_{tar}) \in (X \times \Sigma \times X) \setminus \longrightarrow$, so Theorem 2.2 holds for Algorithm 2.9. \square

Removed Transition (Algorithm 2.10)

Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m)$. Additionally, we denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and $(\hat{Y}, \hat{G}) = \text{TSSRT}(A, Y, G, (x_{or}, \sigma, x_{tar}))$.

- In case $x_{or} \in Y$, $x_{tar} \in Y$, and $x_{or} \neq x_{tar}$, states in $X \setminus G$ remain non-coreachable or non-controllable. Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$.
- In case $x_{or} \in G \setminus Y$, $x_{tar} \in G$, and $x_{or} \neq x_{tar}$, states in $X \setminus G$ remain non-coreachable or non-controllable. Also, states in Y remain (co-)reachable and controllable. Therefore, $\text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, Y \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$.
- In case $x_{or} \in X \setminus G$, $x_{tar} \in X \setminus G$, and $x_{or} \neq x_{tar}$, states in Y remain (co-)reachable and controllable. States in G remain coreachable and controllable. In case that
 - $\sigma \in \Sigma_c$, the non-coreachability or non-controllability of x_{or} and x_{tar} do not change. So $\hat{G} = G = G'$ and $\hat{Y} = Y = Y'$.
 - $\sigma \in \Sigma_u$, then x_{or} may have been non-controllable, but may be controllable in the variant model. Because states in Y remain (co-)reachable and controllable, and states in G remain coreachable and controllable; $Y \subseteq Y'$ and $G \subseteq G'$. Therefore $\text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, G \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m))$.
- A removed transition from $x_{or} \in Y$ to $x_{tar} \in G \setminus Y$ cannot exist, as x_{tar} was reachable in the base model by this transition, and x_{tar} would have existed in Y .

- In case $x_{or} \in Y$ and $x_{tar} \in X \setminus G$, then states in Y remain (co-)reachable and controllable, states in G remain coreachable and controllable, but not reachable, and states in $X \setminus G$ remain non-coreachable or non-controllable. Therefore $\hat{G} = G = G'$ and $\hat{Y} = Y = Y'$.
- In case $x_{or} \in X \setminus G$ and $x_{tar} \in G$, then x_{or} was coreachable in the base model. As it is not in G , x_{or} must be non-controllable. It will remain as such after removal of the transition $(x_{or}, \sigma, x_{tar})$. Thus, states in Y remain (co-)reachable and controllable, states in G remain coreachable and controllable, but not reachable, and states in $X \setminus G$ remain non-coreachable or non-controllable. Therefore $\hat{G} = G = G'$ and $\hat{Y} = Y = Y'$.
- In case $x_{or} = x_{tar}$, the (co-)reachability and controllability of any state does not change. So $\hat{G} = G = G'$ and $\hat{Y} = Y = Y'$.

We can conclude that $\hat{Y} = Y'$ and $\hat{G} = G'$ for any $(x_{or}, \sigma, x_{tar}) \in \longrightarrow$, so Theorem 2.2 holds for Algorithm 2.10. \square

Added state

In case a state x^δ is added as an atomic model adaptation, there are no transitions to or from this state, because for the base model it holds that $\longrightarrow \subseteq X \times \Sigma \times X$, and $x^\delta \notin X$. It is also not an initial state or marked state because for the base model it holds that $X_0 \subseteq X$ and $X_m \subseteq X$. Therefore the added state x^δ is non-coreachable and non-reachable in the variant model. Therefore $Y' = Y$, and $G' = G$, proving Lemma 2.5 for an added state. \square

Removed state

In case a state x^δ is removed as an atomic model adaptation, there are no transitions to or from this state, because for the variant model it holds that $\longrightarrow \subseteq (X \setminus \{x^\delta\}) \times \Sigma \times (X \setminus \{x^\delta\})$. It is also not an initial state or marked state because for the variant model it holds that $X'_0 \subseteq X \setminus \{x^\delta\}$ and $X'_m \subseteq X \setminus \{x^\delta\}$. Therefore the removed state x^δ is non-coreachable and non-reachable in the base model. Therefore $Y' = Y$, and $G' = G$, proving Lemma 2.5 for a removed state. \square

Added event

In case an event σ is added as an atomic adaptations, there are no transitions over this event, because for the base model it holds that $\longrightarrow \subseteq X \times \Sigma \times X$, and $\sigma \notin \Sigma$. Therefore,

adding the event is not going to influence the (co-)reachability or controllability of any state. Thus, $Y' = Y$, and $G' = G$, proving Lemma 2.5 for an added event. \square

Removed event

In case an event σ is removed as an atomic adaptations, there are no transitions over this event in the base model, because for the variant model it holds that $\longrightarrow' \subseteq X \times (\Sigma \setminus \{\sigma\}) \times X$, and $\longrightarrow' = \longrightarrow$. Therefore, removing the event is not going to influence the (co-)reachability or controllability of any state. Thus, $Y' = Y$, and $G' = G$, proving Lemma 2.5 for a removed event. \square

Appendix B

Proofs for ITSS

The lemmas in this appendix support Theorem 2.3 for Algorithm 2.11. Following from the lemmas, Theorem 2.3 is proven at the end of this appendix.

Lemma B.1 Following each line in Algorithm 2.11 that directly follows a call to an atomic TSS algorithm (i.e., lines 5,9,13,17,21,25), it holds for the intermediate automaton that is formed by $\hat{A} = (X', \Sigma', \longrightarrow', X'_0, X'_m)$ that `computeFixpoint` (\hat{A}) = (Y', G') , where (Y', G') is the result of the atomic TSS algorithm in the line above.

Proof The correct result (by Theorem 2.2) of each atomic TSS algorithm is proven in Appendix A. \hat{A} correctly constructs the variant model after applying the atomic model delta, following the definitions in Section 2.3. □

Lemma B.2 Each time an atomic TSS algorithm is initiated by Algorithm 2.11, the atomic adaptation is a valid model delta for the automaton that is input.

Proof We subdivide the proof over all calls to the atomic TSS algorithms, Algorithms 2.5 - 2.10:

- From Section 2.3, we know that $X_0^+ \subseteq (X \cup X^+) \setminus X^-$, and $X_0^+ \cap X_0 = \emptyset$. At the time Algorithm 2.5 (TSSAIP) is initiated with $x^\delta \in X_0^+$, this is with an automaton with state set $X' = X \cup X^+$ and initial state set X'_0 . It holds that $\{x^\delta\} \subseteq X'$. It holds that $\{x^\delta\} \cap X'_0 = \emptyset$, as x^δ is only added to X'_0 after the call to Algorithm 2.5 with x^δ as added initial state. We can conclude that $X_0^+ = \{x^\delta\}$ is a valid model delta for the automaton that is input when Algorithm 2.5 is initiated.
- From Section 2.3, we know that $X_0^- \subseteq X_0$. At the time Algorithm 2.6 (TSSRIP) is initiated with $x^\delta \in X_0^-$, this is with an automaton with initial state set $X'_0 \subseteq X \cup X^+$. x^δ is in X'_0 when Algorithm 2.6 is called with x^δ , as x^δ is only removed from X'_0 after this call. We can conclude that $X_0^- = \{x^\delta\}$ is a valid model delta for the automaton that is input when Algorithm 2.6 is initiated.

- From Section 2.3, we know that $X_m^+ \subseteq (X \cup X^+) \setminus X^-$, and $X_m^+ \cap X_m = \emptyset$. At the time Algorithm 2.7 (TSSAMP) is initiated with $x^\delta \in X_m^+$, this is with an automaton with state set $X' = X \cup X^+$ and marked state set X'_m . It holds that $\{x^\delta\} \subseteq X'$. It holds that $\{x^\delta\} \cap X'_m = \emptyset$, as x^δ is only added to X'_m after the call to Algorithm 2.7 with x^δ as added marked state. We can conclude that $X_m^+ = \{x^\delta\}$ is a valid model delta for the automaton that is input when Algorithm 2.7 is initiated.
- From Section 2.3, we know that $X_m^- \subseteq X_m$. At the time Algorithm 2.8 (TSSRMP) is initiated with $x^\delta \in X_m^-$, this is with an automaton with initial state set $X'_m \subseteq X \cup X^+$. x^δ is in X'_m when Algorithm 2.8 is called with x^δ , as x^δ is only removed from X'_m after this call. We can conclude that $X_m^- = \{x^\delta\}$ is a valid model delta for the automaton that is input when Algorithm 2.8 is initiated.
- From Section 2.3, we know that: $\longrightarrow \cup \longrightarrow^+ \subseteq X' \times \Sigma' \times X'$, and $\longrightarrow^+ \cap \longrightarrow = \emptyset$. At the time Algorithm 2.9 (TSSAT) is initiated with $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^+$, this is with an automaton with state set $X' = X \cup X^+$ and event set $\Sigma' = \Sigma \cup \Sigma^+$. It holds that $\{(x_{or}, \sigma, x_{tar})\} \subseteq X' \times \Sigma' \times X'$. It holds that $\{(x_{or}, \sigma, x_{tar})\} \cap \longrightarrow' = \emptyset$, as $\{(x_{or}, \sigma, x_{tar})\}$ is only added to \longrightarrow' after the call to Algorithm 2.9 with $(x_{or}, \sigma, x_{tar})$ as added transition. We can conclude that $\longrightarrow^+ = \{(x_{or}, \sigma, x_{tar})\}$ is a valid model delta for the automaton that is input when Algorithm 2.9 is initiated.
- From Section 2.3, we know that $\longrightarrow^- \subseteq \longrightarrow$. At the time Algorithm 2.10 (TSSRT) is initiated with $(x_{or}, \sigma, x_{tar}) \in \longrightarrow^-$, this is with an automaton with state set $X' = X \cup X^+$ and event set $\Sigma' = \Sigma \cup \Sigma^+$. $(x_{or}, \sigma, x_{tar})$ is in \longrightarrow' when Algorithm 2.10 is called with $(x_{or}, \sigma, x_{tar})$, as $(x_{or}, \sigma, x_{tar})$ is only removed from \longrightarrow' after this call. We can conclude that $\longrightarrow^- = \{(x_{or}, \sigma, x_{tar})\}$ is a valid model delta for the automaton that is input when Algorithm 2.10 is initiated.

Together, the above cases prove Lemma B.2. □

Lemma B.3 Each time states are added or removed, or events are added or removed in Algorithm 2.11, this adaptation is a valid model delta for the automaton it is performed on.

Proof From Section 2.3, we know that $X^+ \cap X = \emptyset$ and $\Sigma^+ \cap \Sigma = \emptyset$. The added states and added events are only added once to state set X in Algorithm 2.11, at the point they are added, this is a valid model delta. The removed states and removed events are removed from automaton $A' = (X', \Sigma', \longrightarrow', X'_0, X'_m)$ in line 28. At this point, all added transitions are added to \longrightarrow' and removed transitions are removed from \longrightarrow . Also all states with added marked property are added to X'_m , all states with removed marked property are removed from X'_m , all states with added initial property are added to X'_0 , and all states with removed initial property are removed from X'_0 . So in automaton A' , $X^- \cap X'_m = \emptyset$, $X^- \cap X'_0 = \emptyset$, and $\longrightarrow' \subseteq ((X \cup X^+) \setminus X^-) \times ((\Sigma \cup \Sigma^+) \setminus \Sigma^-) \times ((X \cup X^+) \setminus X^-)$. In other words, the removed states are not initial or marked, there are no transitions to-or from removed states, and there are no transitions over removed events. Thus, Lemma B.3 is proven for Algorithm 2.11. □

Lemma B.4 If final fixpoint result Y in SS is equal to fixpoint result Y' in ITSS, then supervisor S is also the same.

Proof From Lemma B.1 it follows that the final fixpoint result Y' in ITSS is equal to the final fixpoint result Y in SS. By following the steps of Algorithm 2.11, it follows that at line 27; $\Sigma' = (\Sigma \cup \Sigma^+)$. Line 27 of Algorithm 2.11 is equal to line 2 of Algorithm 2.1 when $Y' = Y$ and $\Sigma' = (\Sigma \cup \Sigma^+) \setminus \Sigma^-$, thus computing the same automaton S . \square

From Lemmas B.1-B.3 it follows that each intermediate result (Y', G') is correctly constructed in Algorithm 2.11, so that the final result (Y', G') is equal to `computeFixpoint(A')`. From Lemma B.4 it follows that the supervisor automaton computed by Algorithm 2.11 is the same as the supervisor automaton computed by Algorithm 2.1, for the same supervisor states in Y . Together, the lemmas show that Theorem 2.3 holds. \square

Appendix C

Proofs for GTSS

First we prove the application of a set Δ^\sim atomic model adaptations in Δ^\times , defined in Section 2.5.2, in the same manner as Appendix A. Following, we provide the proof for Theorem 2.4 for Algorithm 2.12.

Lemma C.1 For a set Δ^\sim , the same statements associated with a case statements of Algorithms 2.5 - 2.10 can be applied, as long as $\forall \delta \in \Delta^\sim$ the same case condition holds.

Proof We structure our proof the same way as Appendix A, for each atomic TSS algorithm the possible Δ^\sim in Δ^\times is discussed. We denote $(Y, G) = \text{computeFixpoint}(A)$, $(Y', G') = \text{computeFixpoint}(A')$, and (\hat{Y}, \hat{G}) is calculated by the TSS algorithm.

- We consider $\Delta^\sim = X_0^+ \cap (G \setminus Y)$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0 \cup \Delta^\sim, X_m)$.
 - For any $(X_0, X'_0) \in X \times X$ it holds that $G = G'$ when computing $(Y, G) = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X_0, X_m)$, $(Y', G') = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X'_0, X_m)$, as the initial state does not influence the computation of G . We also observe that for all switchcases in Algorithm 2.5, $\hat{G} = G$ is computed. It follows that $\hat{G} = G'$.
 - As $\Delta^\sim \subseteq G \setminus Y$, for the supervisor synthesis of A' , the reachable part is determined by $Y = \text{FRS}(G', \Sigma, \longrightarrow, X'_0)$, where we know that $G' = G$ and $X'_0 = X_0 \cup \Delta^\sim$. \hat{Y} is calculated by $\text{FRS}(G, \Sigma, \longrightarrow, Y \cup \Delta^\sim)$. The result of these FRSs is the same, as we know that all states in Y are reachable in G from X_0 from the base synthesis. So $\hat{Y} = Y'$.
- We consider $\Delta^\sim = X_0^- \cap Y$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0 \setminus \Delta^\sim, X_m)$.
 - For any $(X_0, X'_0) \in X \times X$ it holds that $G = G'$ when computing $(Y, G) = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X_0, X_m)$, $(Y', G') = \text{computeFixpoint}(X, \Sigma, \longrightarrow, X'_0, X_m)$, as the initial state does not influence the computation of

G . We also observe that for all switchcases in Algorithm 2.5, $\hat{G} = G$ is computed. It follows that $\hat{G} = G'$.

- In case that $\Delta^\sim \subseteq Y$, for the supervisor synthesis of A' , the reachable part is determined by $Y = \text{FRS}(G', \Sigma, \longrightarrow, X'_0)$, where we know that $G' = G$ and $X'_0 = X_0 \setminus \Delta^\sim$. \hat{Y} is calculated by $\text{FRS}(Y, \Sigma, \longrightarrow, X_0 \setminus \Delta^\sim)$. The result of these FRSs is the same, as we know that all states in $G \setminus Y$ are not reachable from the base synthesis. So $\hat{Y} = Y'$.
- We consider $\Delta^\sim = X_m^+ \cap X \setminus G$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0, X_m \cup \Delta^\sim)$.
 - In case that $\Delta^\sim \subseteq X \setminus G$, states in Y remain reachable, coreachable, and controllable. So, $\text{computeFixpoint}((X, \Sigma, \longrightarrow, Y \cup X_0, Y \cup X_m \cup \Delta^\sim)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \cup \Delta^\sim))$.
- We consider $\Delta^\sim = X_m^- \cap Y$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0, X_m \setminus \Delta^\sim)$.
 - We know that $\Delta^\sim \subseteq X_m$ (model delta is valid). And all states in X_m are coreachable by definition. G was the maximal controllable coreachable set to X_m . After removing Δ^\sim as marked states; $G' \subseteq G$ and $Y' \subseteq Y$. So all states in $X \setminus G \subseteq X \setminus G'$. Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow, X_0, X_m \setminus \Delta^\sim)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \setminus \Delta^\sim))$. So in case $\Delta^\sim \subseteq Y$, $\hat{Y} = Y'$ and $\hat{G} = G'$.
- We consider $\Delta^\sim = X_m^- \cap G \setminus Y$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow, X_0, X_m \setminus \Delta^\sim)$.
 - In case that $\Delta^\sim \subseteq G \setminus Y$, all states in Y are coreachable and controllable for $X_m \setminus x^\delta$, as states in Δ^\sim are not reachable from Y , otherwise they would be contained in Y . Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow, X_0, (Y \cup X_m) \setminus \Delta^\sim)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow, X_0, X_m \setminus \Delta^\sim))$. So in case $\Delta^\sim \subseteq G \setminus Y$, $\hat{Y} = Y'$ and $\hat{G} = G'$.
- We consider $\Delta^\sim = \longrightarrow^+ \cap Y \times \Sigma \times Y$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \cup \Delta^\sim, X_0, X_m)$.
 - The coreachability and controllability of all states remains unchanged. So $\hat{G} = G = G'$. All states in Y remain reachable, so $\text{FRS}(G, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y) = \text{FRS}(G, \Sigma, \longrightarrow \cup \Delta^\sim, X_0)$.
- We consider $\Delta^\sim = \longrightarrow^+ \cap G \times \Sigma_u \times X \setminus G$.
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \cup \Delta^\sim, X_0, X_m)$.
 - The states in $X \setminus G$ remain non-coreachable or non-controllable. The supervisor cannot disable the added transitions. All states in X_{or} are non-controllable, where $X_{or} = \{x_{or} | (x_{or}, \sigma, x_{tar}) \in \Delta^\sim, \sigma \in \Sigma_u, x_{tar} \in X\}$. Because also states in $X \setminus G$ remain non-coreachable or non-controllable, computeFixpoint

$$((G, \Sigma, \longrightarrow \cap ((G \setminus X_{or}) \times \Sigma \times G), X_0 \setminus X_{or}, X_m \setminus X_{or})) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, X_0, X_m)).$$

- We consider $\Delta^\sim = \{(x_{or}, \sigma, x_{tar}) \in \longrightarrow^+ \mid x_{or} \in X \setminus G \wedge x_{or} \neq x_{tar}\}$
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \cup \Delta^\sim, X_0, X_m)$.
 - All states in Y remain (co-)reachable and controllable. Thus we conclude, $\text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \{(x_{or}, \sigma, x_{tar})\}, Y \cup X_0, Y \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \cup \Delta^\sim, X_0, X_m))$.
- We consider $\Delta^\sim = \{(x_{or}, \sigma, x_{tar}) \in \longrightarrow^- \mid x_{or} \in Y \wedge x_{tar} \in Y \wedge x_{or} \neq x_{tar}\}$
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m)$.
 - States in $X \setminus G$ remain non-coreachable or non-controllable. Therefore $\text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m))$.
- We consider $\Delta^\sim = \{(x_{or}, \sigma, x_{tar}) \in \longrightarrow^- \mid x_{or} \in G \setminus Y \wedge x_{tar} \in G \wedge x_{or} \neq x_{tar}\}$
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m)$.
 - States in $X \setminus G$ remain non-coreachable or non-controllable. Also, states in Y remain (co-)reachable and controllable. It follows that $\text{computeFixpoint}((G, \Sigma, \longrightarrow \setminus \Delta^\sim, Y \cup X_0, Y \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m))$.
- We consider $\Delta^\sim = \{(x_{or}, \sigma, x_{tar}) \in \longrightarrow^- \mid x_{or} \in X \setminus G \wedge x_{tar} \in X \setminus G \wedge \sigma \in \Sigma_u \wedge x_{or} \neq x_{tar}\}$
Automaton $A = (X, \Sigma, \longrightarrow, X_0, X_m)$, and $A' = (X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m)$.
 - States in Y remain (co-)reachable and controllable. States in G remain coreachable and controllable. The origin states of the removed transition may have been non-controllable, but may be controllable in the variant model. Because states in Y remain (co-)reachable and controllable, and states in G remain coreachable and controllable; $Y \subseteq Y'$ and $G \subseteq G'$. Therefore $\text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \Delta^\sim, Y \cup X_0, G \cup X_m)) = \text{computeFixpoint}((X, \Sigma, \longrightarrow \setminus \Delta^\sim, X_0, X_m))$.

For any possible Δ^\sim Lemma C.1 is proven. \square

Lemma C.2 Each time $X', \Sigma', \longrightarrow', X'_0$, or X'_m is constructed in Algorithm 2.12 (i.e., lines 1,6,8,12), it holds that $\text{computeFixpoint}(X', \Sigma', \longrightarrow', X'_0, X'_m) = (Y', G')$, for (Y', G') computed at that point in Algorithm 2.12.

Proof We structure our proofs over the different lines where automaton A' is constructed.

- Lines 1,12: In case states are added or removed, or events are added or removed in Algorithm 2.12, the same proofs as for Lemma 2.5 and Lemma B.3 hold here.
- Line 6: This is the same iterative application as in Algorithm 2.11 (ITSS). The same proofs as for Lemma B.1 and Lemma B.2 hold here.

- Line 8: The correct result of applying sets Δ^{\sim} is proven for Lemma C.1 above. In conjunction with the proof for Lemma B.2, this proves Lemma C.2 for line 8.

For each time X' , Σ' , \longrightarrow' , X'_0 , or X'_m is constructed in Algorithm 2.12, Lemma C.2 is proven. \square

From Lemma C.2 it follows that each intermediate result (Y', G') is correctly constructed in Algorithm 2.12, so that the final result (Y', G') is equal to `compute Fixpoint(A')`. Using the same proof as Lemma B.4, it follows that the supervisor automaton computed by Algorithm 2.12 is the same as the supervisor automaton computed by Algorithm 2.1, for automaton A' . Together, this shows that Theorem 2.4 holds. \square

Curriculum vitae



Sander Thuijsman was born on the 18th of September 1995 in Eijsden, the Netherlands. After finishing VWO in 2013 at Sint-Maartenscollege Maastricht, the Netherlands, he received his B.Sc. degree in 2016, and his M.Sc. degree ‘with great appreciation’ in 2019, both in Mechanical Engineering at Eindhoven University of Technology, the Netherlands. In 2018 he collaborated with Rong Su during a research visit at Nanyang Technological University, Singapore. His M.Sc. graduation project concerned measuring the computational effort of symbolic supervisor synthesis. In 2019 he started his Ph.D. project under the supervision of Michel Reniers at Eindhoven University of Technology, in collaboration with ASML, located in Veldhoven, the Netherlands. The results of his research are presented in this thesis. In 2022 he collaborated with Kai Cai during a research visit at Osaka Metropolitan University, Japan.

