Technische Universität München
Lehrstuhl III – Datenbanksysteme

Algorithms for the Efficient Verification and Planning of Information Technology Change Operations

Sebastian Hagen
M.Sc. with honours

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 31.01.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 14.05.2013 angenommen.

Dedicated to my supporting mother, who was diagnosed of Lou Gehrig's disease halfway through my Ph.D. and did not live long enough to see this work finished.

24.06.1953 - 23.08.2012

Abstract

The Information Technology Infrastructure Library (ITIL) describes a set of best practices on how to manage IT systems while reducing incidents and increasing reliability. Change Management, a core process of ITIL, is concerned with the application of IT changes to networks and services. With many modern companies relying on the availability of IT services, success of Change Management has become crucial for the well being of a company. Today Change Management heavily depends on the individual skills of IT personnel and algorithms to automate Change Management are rare. To further improve the reliability of Change Management, we introduce in this work efficient algorithms for the automated planning and verification of IT change operations, two important steps of the Change Management process.

First, we introduce an efficient algorithm for the verification of IT change operations. The algorithm is an extended, finer-grained version of the partial-order reduction model checking paradigm. Extended partial-order reduction can reduce the complexity to verify a global formula given a set of always applicable, concurrent effects to linear runtime complexity. The theory of extended partial-order reduction is introduced and a many-sorted logic for IT change verification is proven to adhere to it. We show that extended partial-order reduction significantly outperforms the state of the art general purpose model checkers SPiN and NuSMV for real-world change operations and configurations likely to have caused a recent outage in one of Amazon's cloud computing data centers. Thus, the extended partial-order reduction verification approach presented herein makes the verification of IT change operations feasible on large configurations for the first time.

Second, we examine several general purpose planning paradigms for the automated generation of IT change plans. We conclude that SHOP2, a Hierarchical Task Network planner, is the one to most naturally fit the domain of IT change planning while providing the best performance. However, as its scalability remains limited, we discuss and analyze optimizations to further improve the runtime complexity of decomposition-based total-order IT change planning. A sensitivity analysis shows that our optimizations can significantly reduce the runtime complexity for IT change plan generation from polynomial to linear or even constant complexity for a variety of characteristics of IT changes and configurations. A cloud deployment and a virtual network configuration case study demonstrate the feasibility of the approach and confirm the results of the sensitivity analysis.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With many modern companies relying on IT services, e.g., email, databases, application servers, or enterprise resource planning systems, the success of these businesses heavily depends on the availability of the IT infrastructure and the services provided by them. The hardware and software systems of a company face a magnitude of change operations every day. For example, BMW Financial Services Germany faces roughly 3600[1] IT change operations every year that need to be applied to the software and hardware systems providing the services of a financial institute 24 by 7. Typical changes are, for example, the upgrade/patching of a software, the reconfiguration of a software, e.g., adding a user to a database, the deletion of unnecessary database snapshots, the deployment of software, or the reconfiguration of a network. The complexity inherent to large IT infrastructures, which are estimated to comprise between 450,000 (Amazon)[2] and 900,000 (Google)[3] servers and a multitude of services supported by them, further complicates the reliable application of IT change operations to IT infrastructures. At this level of complexity the usual case to expect is that something goes wrong instead of right. Sometimes failures caused by faulty IT change operations cannot be concealed and outages with the potential to destroy the reputation of a company occur. For example, in April 2011 a network upgrade conducted in one of Amazon's data centers caused outages to customer services that lasted several days and even caused durable data loss. To prevent outages caused by faulty IT change operations, Change Management[4] [65], which is part of the Information Technology Infrastructure Library (ITIL) [78], describes a set of best practices on how to manage IT infrastructure and services such that disruptions on the business are minimized.

[1]The number was obtained from a conversation the author had with a change manager working at BMW Financial Services Germany.

[2]http://www.datacenterknowledge.com/archives/2012/03/14/estimate-amazon-cloud-backed-by-450000-servers/

[3]http://www.datacenterknowledge.com/archives/2011/08/01/report-google-uses-about-900000-servers/

[4]Whenever we refer to Change Management in this work, we mean Change Management in the context of IT Service Management (ITSM) and the Information Technology Infrastructure Library (ITIL).

The Change Management process [65] comprises steps to evaluate, authorize, plan, test, schedule, implement, document, and review IT changes. In practice the success of the process very much depends on the individual skills of IT personnel and it is hardly supported by automated tools and algorithms from a logical reasoning perspective. The development of automated algorithms to support steps of the Change Management process has the potential to increase the reliability of the process and makes it less dependent on the individual skills of IT personnel. Two steps of the Change Management process are particularly well suited for automation:

- **Planning of IT change operations:** In the planning step of the Change Management process, change managers - if at all - manually draft change plans, which comprise the basic change activities that need to be executed in order to achieve the Request for Change (RFC), i.e., the goal specification of an IT change. Given the specification of the preconditions and effects of basic IT change activities, automated planners [41] compute logically sound IT change plans that adhere to the logical constraints of the application and infrastructure. Consequently, the computed plans are guaranteed to be executable without violation of constraints.

- **Verification of IT change operations:** The evaluation, authorization, and test steps of the Change Management process heavily rely on the experience of a change manager and the skills of IT personnel to thoroughly assess and test IT changes. Those steps can be supported by the automated verification [11, 29] of IT change operations. Verification proves on a logical level that a set of pending IT change operations will not invalidate safety constraints when executed in any order. Thus, if the logical model accurately captures reality, change verification detects the occurrence of threatening conditions, for example, the overload of a network, which was the reason for a recent outage at one of Amazon's data centers.

1.2 Problem Statement

Although the automated planning [41] and verification [11, 29] communities have extensively discussed planning and verification algorithms, the application of these investigations to IT change planning [71] and IT change verification [61, 83] yields a common drawback: scalability. Because realistically sized IT infrastructures comprise up to millions of Configuration Items (CIs), i.e., IT infrastructure assets such as software and hardware artifacts, existing algorithms quickly hit a scalability barrier that makes them inapplicable to real-world IT change planning and verification problems. Thus, novel and domain-specific algorithms for the verification and planning of IT change operations need to be developed to aid in the tool-support of important steps of the ITIL Change Management process. Such algorithms need to have significantly better scalability than state of the art planning [41] and verification algorithms [11, 29], but, at the same time, they need to be expressive enough to describe realistic IT change operations.

To solve this shortcoming, we introduce in this work algorithms for the efficient planning and verification of IT change operations that - compared to general purpose planners and model checkers - significantly reduce the runtime complexity of IT change verification/planning from exponential/polynomial to linear or even constant runtime complexity. Thus, the results presented herein make IT change plan generation and verification feasible on large-scale IT infrastructures for the first time.

1.3 Contributions

The contributions of this work to the verification and the networks and service management communities are as follows.

IT Change Verification

- **Verification community:** This work describes the formal requirements to efficiently verify whether a formula ϕ holds on every state that can be obtained by the execution of a set of always enabled, concurrent effects to an initial state. Relationships between effects and formulas, which enable the efficient verification of $\forall\Box\phi$[1] computation tree logic (CTL) and $\Box\phi$[2] linear temporal logic (LTL) formulas, where ϕ is evaluated in a state and does not comprise temporal operators, are introduced. The optimization technique is best described as an extended, finer-grained version of the partial-order reduction [12] model checking paradigm, which enables the further reduction of the search space - even in cases when partial-order reduction is not applicable.
  We formalize the precise requirements for a logic to be efficiently verifiable using the extended partial-order reduction approach. It is shown that a domain specific many-sorted logic [72] for IT change verification satisfies the requirements of extended partial-order reduction. Compared to state of the art model checkers, such as NuSMV and SPiN, our approach can significantly reduce the runtime complexity for the verification of $\forall\Box\phi$ CTL and $\Box\phi$ LTL formulas from exponential/polynomial to linear runtime complexity.

- **Service management community:** This work presents a domain specific logic and efficient algorithms for the verification of IT change operations against infrastructure and software configuration safety constraints. We show that our solution, compared to the general purpose model checkers SPiN and NuSMV, significantly reduces the runtime complexity of IT change verification from exponential/polynomial to linear for many different configurations, models, and change activities. Several scenarios that could have caused a real-world network outage at one of Amazon's data centers are shown to be detectable using the verification approach. Different to previous investigations on change verification [22, 50, 61, 83], this work addresses and solves the scalability problems of IT change verification to very large configurations for the first time.

  Compared to previous work that also aims to increase the reliability of the Change Management process, e.g., by means of risk analysis [87, 96, 97] and the optimal assignment of human operators to IT change activities [67], change verification provides strict logical means to detect whether a change fails in a logical model. Furthermore, failures are detected at a much earlier stage while approaches to rollback IT changes [69, 70] or to monitor infrastructures [73] are often a step behind to prevent outages.

IT Change Planning

- **Service management community:** This work evaluates several general purpose planning paradigms [10, 18, 76, 94] for IT change planning with a focus on runtime performance and feasibility by IT change managers. Among the examined planning paradigms, we

[1] $\forall\Box\phi$ is the on all paths now and forever in the future ϕ formula from computation tree logic (CTL) [12]
[2] $\Box\phi$ is the always (now and forever in the future) ϕ formula from linear temporal logic (LTL) [12]

find that SHOP2 [76], a Hierarchical Task Network (HTN) [42] planner, is most performant and usable for IT change planning. Optimization techniques are introduced to further improve the runtime performance of decomposition-based IT change planning algorithms from polynomial to linear or even constant runtime complexity. A sensitivity and case study analysis shows that the optimizations outperform the SHOP2 planner in terms of runtime complexity for many characteristics of IT change operations and configurations of the infrastructure. In addition to that, the optimizations presented herein are more robust in respect to the characteristics of IT changes and the configuration of the infrastructure. Our work improves related Change Management problems as well: Faster planning aids in optimal plan generation because more plans can be explored in the same time and the best can be chosen among them. Furthermore, faster planners leave more time to optimize the scheduling of IT change operations [84, 101, 102].

IT change plan generation has been subject to several investigations, addressing topics such as the integration of planning and scheduling [59], the reuse of knowledge in IT change design [33, 35], algorithms for the hierarchical decomposition of IT changes [46, 91] or algorithms that are not based on decomposition [38, 48, 71]. Common to all investigations is that they have been evaluated to work on small to medium size IT change planning problems and that scalability to large CMDBs has either been out of their scope or cannot be achieved by them. Thus, to the best of our knowledge, the algorithms and optimizations discussed herein make IT change planning feasible on large infrastructures for the first time.

Previous Publications

Previous Work

Some of the contributions in this work have been previously published. The following publications appeared in peer-reviewed conference proceedings:

S. Hagen, W. L. da Costa Cordeiro, L. P. Gaspary, L. Z. Granville, M. Seibold, and A. Kemper. Plannig in the Large: Efficient Generation of IT Change Plans on Large Infrastructures. In *Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), Las Vegas, NV, USA, October 22-26, 2012*, pages 108–116. IEEE, 2012.

S. Hagen, M. Seibold, and A. Kemper. Efficient Verification of IT Change Operations or: How We Could Have Prevented Amazon's Cloud Outage. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS 2012), Maui, HI, USA, April 16-20, 2012*, pages 368–376. IEEE, 2012.

S. Hagen and A. Kemper. A Performance and Usability Comparison of Automated Planners for IT Change Planning. In *Proceedings of the 7th International Conference on Network and Service Management (CNSM 2011), Paris, France, October 24-28, 2011*, pages 1–9. IEEE, 2011.

S. Hagen and A. Kemper. Towards Solid IT Change Management: Automated Detection of Conflicting IT Change Plans. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), Dublin, Ireland, May 23-27, 2011*, pages 265–

272. IEEE, 2011.

S. Hagen and A. Kemper. Facing The Unpredictable: Automated Adaption of IT Change Plans for Unpredictable Management Domains. In *Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), Niagara Falls, Canada, October 25-29, 2010*, pages 33–40. IEEE, 2010.

S. Hagen and A. Kemper. Model-based Planning for State-related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD 2010), Miami, FL, USA, July 5-10, 2010*, pages 11–18. IEEE, 2010.

Previous Work Part of Previous Degrees

The following publication has been part of the author's preceding postgraduate master's degree and is not part of this work[1]:

S. Hagen, N. Edwards, L. Wilcock, J. Kirschnick, and J. Rolia. One Is Not Enough: A Hybrid Approach for IT Change Planning. In *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2009), Venice, Italy, October 27-28, 2009*, volume 5841 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2009. *This work has been part of the author's preceding master's degree.*

1.4 Nomenclature

This work is about the verification and planning of IT change operations. As there are many synonyms for IT changes, which are often perceived differently by different readers, we shortly explain the nomenclature of IT changes used throughout this work. First, we use the notion of *IT changes* and *IT change activities* interchangeably. Whenever necessary, we distinguish between *atomic* and *abstract* IT changes/change activities. An atomic change activity is an indivisible, elementary change operation that is applied to IT infrastructure or services. Synonyms for atomic change activities are, for example, *elementary change activity*, *elementary change operation*, *elementary IT change*, or *(elementary) change procedure*. Among atomic change activities are, for example, a change to turn on a machine, a change to increase the metric of a router interface, or a change to add an entry to a routing table. Different to that, *abstract IT change activities* reside on a higher level of abstraction and require the decomposition into finer-grained changes/change activities to be executable. The change to deploy a three-tier application is a typical abstract change activity that requires further decomposition into many steps to achieve the abstract change activity. A synonym for an abstract change activity is that of a *high-level IT change/change activity*.

In the chapters addressing IT change planning (Chapters 6 and 7) we frequently talk about atomic and abstract IT change activities. However, when it comes to verification (Chapters 2 through 5) our analysis always resides on the level of atomic change activities. In these chapters we deviate from the notion above and only talk about change activities unambiguously referring to atomic change activities.

[1]The work is addressed in the related work sections though.

1.5 Outline

The remainder of this work is roughly organized in two parts: The first part addresses IT change verification (Chapter 2 through 5) and the second part concerns the efficient generation of IT change plans (Chapter 6 and 7). Finally, Chapter 8 concludes the work.

- **Chapter 2** introduces the theory of extended partial-order reduction. Based on the notion of states, effects, and formulas, an efficiently computable criterion to verify CTL formulas of type $\forall\Box\phi$ and LTL formulas of type $\Box\phi$ given a set of concurrent, always executable effects is introduced. The correctness of the criterion is proven. [1]

- **Chapter 3** introduces the syntax and semantics of a many-sorted IT change verification logic that is used to logically describe IT change activities and hosting safety constraints. We prove that the change verification logic satisfies all requirements of the extended partial-order reduction approach introduced in Chapter 2. Thus, the theory underlying the extended partial-order reduction approach can be applied to the change verification logic to efficiently verify atomic change activities against safety constraints. [1]

- **Chapter 4** introduces verification algorithms for several IT change verification problems based on the extended partial-order reduction approach. In addition to that, their runtime complexity is analyzed. [1]

- **Chapter 5** evaluates the extended partial-order reduction approach in the context of several scenarios that could have caused a network outage at one of Amazon's data centers in April 2011. Our approach is evaluated against the general purpose model checkers NuSMV and SPiN with several scenarios, models, workloads, and configurations of the IT infrastructure to have contributed to the outage. The experimental evaluation shows that our approach significantly outperforms NuSMV and SPiN and scales to very large configurations. [1]

- **Chapter 6** compares four general purpose Artificial Intelligence (AI) planners (Graphlan [18], Prodigy [94], TLPlan [10], and SHOP2 [76]) in an IT change planning case study. We focus on two aspects: (1) the planning performance and (2) the usability of each planner by change managers untrained in automated planners. We find that Hierarchical Task Network (HTN) planners (SHOP2 in our case) are best suited for IT change planning due to their superior performance and their proximity to the domain of IT change planning. A representative cloud deployment case study demonstrates the superior performance of SHOP2. [2]

- **Chapter 7** proposes and evaluates optimization techniques for decomposition-based IT change planning algorithms that significantly reduce the runtime complexity of IT change plan generation from polynomial to linear or even constant runtime complexity. A sensitivity analysis examines the influence of several important characteristics of IT changes and the CMDB on the runtime complexity of the proposed optimizations. We show that our optimizations outperform the SHOP2 planner, the winner of the comparison of planners in Chapter 6, in terms of runtime complexity for a large percentage of IT changes

[1]Parts of this chapter previously appeared in [51].
[2]Parts of this chapter previously appeared in [49].

and CMDBs. A cloud deployment case study of a three-tier application and a virtual network configuration case study demonstrate the feasibility of the approach and confirm its scalability. [1]

- **Chapter 8** concludes the work and presents open research questions. [2]

[1]Parts of this chapter previously appeared in [45].
[2]Parts of this chapter previously appeared in [45, 49, 51].

CHAPTER 2

Extended Partial-order Reduction of Global Formulas

This chapter introduces extended partial-order reduction, a finer-grained but more restrictive version of partial-order reduction. Extended partial-order reduction can be used to efficiently verify a formula on all states that evolve from a given initial state by the execution of a set of always applicable concurrent effects.

First, we establish basic notation and introduce the verification problem solved by extended partial-order reduction in Section 2.1. The improvements of extended partial-order reduction over ordinary partial-order reduction are discussed in Section 2.2. Section 2.3 introduces the notion of effects and discusses their influence on formulas when deciding whether a formula always holds. After that, Section 2.4 further refines characteristics of effects in respect to formulas and the sequences of effects they can appear in. Section 2.5 introduces and proves a criterion to determine whether a formula always holds in all states that evolve from a given initial state by the execution of a set of always applicable effects. Section 2.6 refines the criterion into a theorem that can be practically used to efficiently decide the verification problem previously introduced in Section 2.1. Finally, Section 2.7 concludes the chapter on extended partial-order reduction.[1]

2.1 Verification Problem and Definitions

This section establishes basic notation used throughout this work and defines the verification problem solved by extended partial-order reduction. In the remainder of this work we use the following notation:

- $S = \{s_1, \ldots, s_n\}$ denotes a set of states.
- ϕ and ψ denote formulas that hold or do not hold in a state $s \in S$. For a formula ϕ and a state $s \in S$ we write $s \models \phi$ if ϕ holds in state s and $s \not\models \phi$ otherwise.

[1]Parts of this chapter previously appeared in [51].

- $E = \{e_1, \ldots, e_n\}$ denotes a set of effects where $e : S \rightarrow S$ is a function that is applied to a state $s \in S$ and yields a successor state $e(s) \in S$ that is the state that evolves from s by the application of effect e to s.
- We consider sequences of effects $\langle e_1, \ldots, e_n \rangle$, $n \in \mathbb{N}_0$, where $\forall i \in \{1, \ldots, n\} : e_i \in E$. The application of sequences of effects to states is defined as the subsequent application of effects. Thus for a state s and a sequence of effects $\langle e_1, \ldots, e_n \rangle$ we obtain $\langle e_1, \ldots, e_n \rangle(s) = e_n(\ldots(e_1(s)))$. We denote by $Seq(E)$ the set of all effect sequences of arbitrary length such that every effect in E occurs at most once in every sequence $seq \in Seq(E)$. More formally:

$$\begin{aligned} Seq(E) = \{\langle e_1, \ldots, e_n \rangle \mid\ & 0 \leq n \leq |E| \quad \wedge \\ & \forall i \in \{1, \ldots, n\} : e_i \in E \quad \wedge \\ & \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, n\}, i \neq j : e_i \neq e_j\} \end{aligned}$$

- For a sequence $seq \in Seq(E)$ and a set of effects $E' \subseteq E$, we denote by $seq \setminus E'$ the sequence obtained from seq when all effects in E' are removed from seq.

This chapter derives an easily computable criterion to determine whether a formula ϕ is satisfied on any state that can evolve from a given initial state s_{init} by the application of effects in E in any order and length, i.e., deciding whether

$$\forall seq \in Seq(E) : \quad seq(s_{init}) \models \phi$$

holds.

The verification problem stated above is equivalent to model checking the linear temporal logic (LTL) [11] formula $\Box\phi$[1] (where ϕ does not comprise temporal operators) or model checking the computation tree logic (CTL) [11] formula $\forall\Box\phi$[2] in the equivalent state transition systems. Notice that for now we do not enforce any restrictions on the syntax and semantics of formulas. Chapter 3 introduces a logic for IT change verification that satisfies all requirements of extended partial-order reduction introduced herein.

2.2 Motivation and Related Work: Partial-order Reduction

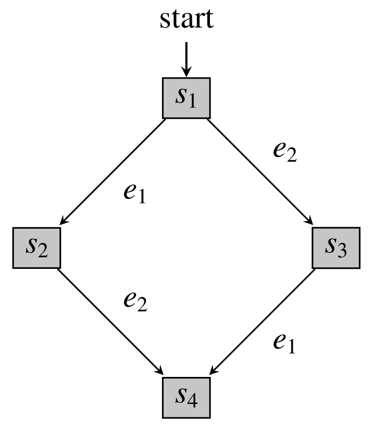
The verification approach most related to extended partial-order reduction is partial-order reduction model checking [12, 30]. Thus, extended partial-order reduction is heavily motivated by the limitations of partial-order reduction and can significantly reduce the search space for cases where partial-order reduction is not applicable anymore. The remainder of this section is organized as follows. First, partial-order reduction and extended partial-order reduction are compared with each other. Second, we provide a motivating example for extended partial-order reduction and an introduction to the basic idea underlying the verification approach.

2.2.1 Partial-order Reduction vs. Extended Partial-order Reduction

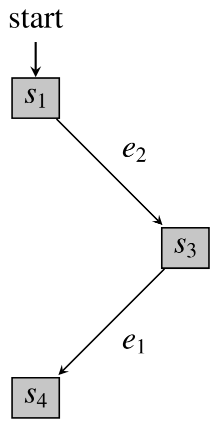
Partial-order reduction [12, 30] is a technique to reduce the search space explored by a model checker taking into account the commutativity of effects. If effects are state commutative and

[1] $\Box\phi$ is the always (now and forever in the future) ϕ formula from linear temporal logic (LTL) [12]

[2] $\forall\Box\phi$ is the on all paths now and forever in the future ϕ formula from computation tree logic (CTL) [12]



(a) Concurrent execution of e_1 and e_2.



(b) Sequential execution of e_2 and e_1.

Figure 2.1: Parallel and sequential execution of effects e_1 and e_2 and their influence on the number of states to be searched by a model checker.

independent of each other, not all interleavings of effects need to be considered during model checking, but it suffices only to model check specific interleavings for which conclusions can be drawn about the satisfiability of a formula on all interleavings. For example, consider Figure 2.1 that depicts the interleaved execution of effects e_1 and e_2. Instead of verifying a formula over any of the four states depicted in Figure 2.1a, partial-order reduction offers means to decide whether it suffices to verify the formula on a smaller state transition system (see Figure 2.1b) that satisfies the same formulas as the original system. Similar to partial-order reduction, extended partial-order reduction is an instance of what has come to be known as model checking by representatives [80, 81]: A simpler, but logically equivalent system in satisfiability, is used for model checking. For partial-order reduction to be applicable effects need to adhere to two important criteria: the *independence* and *stutter* criterion. If effects satisfy both criteria (we later discuss the criteria in more detail), the following modifications can be performed to simplify verification:

- **Adding of independent stutter actions:**
  If e is a stutter effect that is independent of $\{e_1, ..., e_n\}$, then every formula in LTL$\setminus_{\bigcirc}$[1] evaluates equivalently on the state sequence induced by the effect sequences $\langle e_1, ..., e_n\rangle$ and $\langle e, e_1, ..., e_n\rangle$.

- **Permutation of independent stutter effect:**
  If e is a stutter effect that is independent of $\{e_1, ..., e_n\}$, then e can be permuted in effect sequence $\langle e, e_1, ...e_n\rangle$ while every formula in LTL$\setminus_{\bigcirc}$ evaluates equivalently on the state sequences induced by the effect sequences.

Partial-order reduction makes use of these two important observations. Thus, if effects do not satisfy the stutter and independence criterion, partial-order reduction cannot be applied to reduce the search space.

We now discuss in more detail the definitions of stutter and independent effects to discuss their limitations. Let $e \in E$ be an effect. We say that e is enabled in a state s (write $e \in EN(s)$)

[1]Linear temporal logic (LTL) without the next ($\bigcirc$) operator.

iff[1] e can be applied in state s. For a state $s \in S$ we denote by $holds(s)$ the set of all atomic propositions that hold in state s.

- **Independent effects:**
  Two effects e_1 and e_2 are independent iff they satisfy the *Enabledness* and *Commutativity* criteria.
  - **Enabledness:** If e_1 and e_2 are enabled in a state s, then e_1 is enabled in $e_2(s)$ and e_2 is enabled in $e_1(s)$. More formally:

$$\forall s \in S : \quad (e_1 \in EN(s) \wedge e_2 \in EN(s)) \rightarrow (e_1 \in EN(e_2(s)) \wedge e_2 \in EN(e_1(s)))$$

  - **Commutativity:** If e_1 and e_2 are enabled in a state s, then $e_2(e_1(s))$ and $e_1(e_2(s))$ yield the same state. More formally:

$$\forall s \in S : \quad (e_1 \in EN(s) \wedge e_2 \in EN(s)) \rightarrow (e_1(e_2(s)) = e_2(e_1(s)))$$

- **Stutter Effect:**
  An effect e is called a stutter effect iff its application (whenever it is enabled) does not change the set of atomic proposition. More formally:

$$\forall s \in S : \quad e \in EN(s) \quad \rightarrow \quad (holds(s) = holds(e(s)))$$

Based on these definitions, which are central to partial-order reduction, the main differences between partial-order reduction and extended partial-order reduction are as follows.

- **Formulas covered:**
  While partial-order reduction can be applied to any formula in LTL$\backslash_{\bigcirc}$ (assuming that the independent and stutter effect requirements are met), extended partial-order reduction can only be used to verify CTL formulas of type $\forall\Box\phi$ and LTL formulas of type $\Box\phi$ where ϕ does not comprise temporal operators.

- **Enabledness of all effects required for extended partial-order reduction:**
  The specific verification problem ($\forall seq \in Seq(E) : seq(s) \models \phi$) solved by extended partial-order reduction implies that in the underlying state transition system effects are always enabled and that once a transition $s \xrightarrow{e} s'$ caused by effect e has been taken, e never appears again in any transition on any path starting in s' (effects can only be applied once). Thus, for the verification problem considered by extended partial-order reduction, all effects automatically satisfy the enabledness criterion of partial-order reduction. Different to that, partial-order reduction can still be partially applied to a subset of effects satisfying the enabledness criterion. Extended partial-order reduction also requires a commutativity criterion to hold, however commutativity is defined in respect to the evaluation of a formula and not states.

- **State commutativity not necessary for extended partial-order reduction:**
  Different to partial-order reduction, extended partial-order reduction does not require the state commutativity criterion of effects to hold. Thus, extended partial-order reduction can also be applied to effects that are not independent due to the violation of the commutativity criterion.

[1]Short for if and only if.

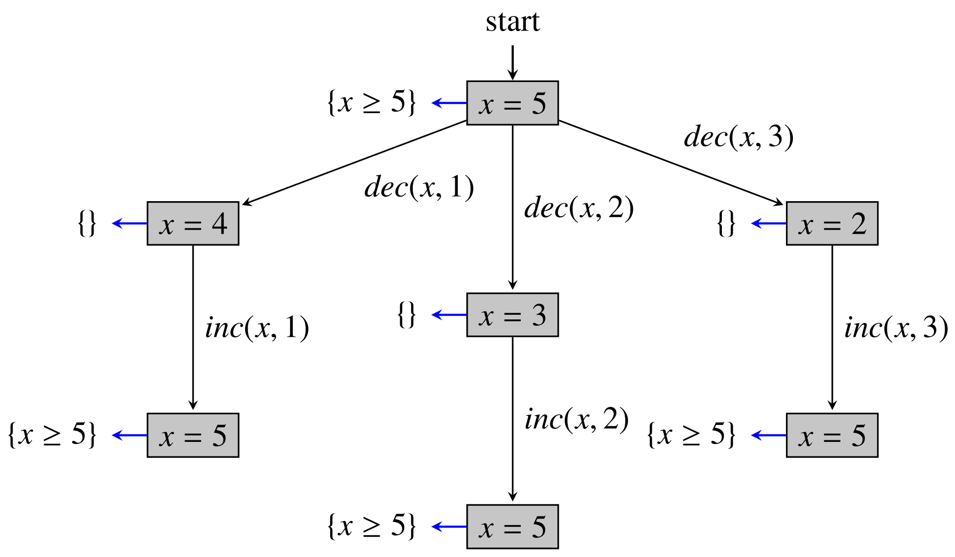


Figure 2.2: Examples to demonstrate the non-stutter effect characteristics (in respect to proposition $x \geq 5$) of the increment and decrement effects for an initial configuration where $x = 5$.

- **Novel interpretation of stutter effects in extended partial-order reduction:** Extended partial-order reduction uses a finer-grained categorization of effects. Instead of demanding that an effect never changes any proposition (stutter effect) we relax the constraint for effects to preserve a formula in one direction (once false always false, once true always true). In addition to that, the categorization is performed and exploited for all combinations of effects and formulas / atomic propositions to take into account how effects relate to parts of a compound formula.

2.2.2 Introduction to Extended Partial-order Reduction

This section provides an introductory example of extended partial-order reduction. Consider a formula ϕ that matches the simple proposition $x \geq 5$ that holds in a state $s \in S$ iff x is greater than or equal to 5. We consider four different effects:

- **Effects to increase x or y:** $inc(x, \Delta c_1)$ and $inc(y, \Delta c_2)$ increment x and y by $\Delta c_1 \geq 0$ respectively $\Delta c_2 \geq 0$.
- **Effects to decrease x or y:** $dec(x, \Delta c_1)$ and $dec(y, \Delta c_2)$ decrement x and y by $\Delta c_1 \geq 0$ respectively $\Delta c_2 \geq 0$.

For the remainder of this discussion we consider E, the set of all effects, to be:

$$E = \{dec(x, 1), dec(x, 2), dec(x, 3), inc(x, 1), inc(x, 2), inc(x, 3), inc(y, 1), dec(y, 1)\}$$

To apply partial-order reduction, stutter effects need to be present. The only stutter effects in E in respect to the language only comprising the proposition $x \geq 5$ are $inc(y, 1)$ and $dec(y, 1)$ because they change y and have no effect on x. All remaining effects in E (the increments and decrements to x) might or might not be stutter effects depending on the value of x in the initial state.

	$x \geq 5$	
Effect	Extended partial-order reduction	Partial-order reduction
$dec(x, 1)$	threat	$\neg$ stutter
$dec(x, 2)$	threat	$\neg$ stutter
$dec(x, 3)$	threat	$\neg$ stutter
$inc(x, 1)$	support	$\neg$ stutter
$inc(x, 2)$	support	$\neg$ stutter
$inc(x, 3)$	support	$\neg$ stutter
$inc(y, 1)$	neutral	stutter
$dec(y, 1)$	neutral	stutter

Table 2.1: Categorization of effects for extended partial-order reduction and partial-order reduction for initial state $x = 5$ and atomic proposition $x \geq 5$.

Figure 2.2 depicts a partial view of a state transition system that evolves from the initial state (where $x = 5$) by the application of the six decrement/increment effects to x in E. For every effect there exists a transition in Figure 2.2 for which the set of atomic propositions holding before the execution of the effect and after the execution of the effect differ. Thus, if x holds 5 in the initial state, none of these effects is a stutter effect. Thus, partial-order reduction cannot be applied to those effects leaving all possible interleavings to be verified by a partial-order reduction model checker.

In the given example, the definition of stutter effects turns out to be overly restrictive. Stutter effects are defined as neutral effects that need to preserve all atomic properties. For extended partial-order reduction we change the requirement in two aspects: First, we exploit that effects can behave differently in respect to different atomic propositions. Second, we relax the stutter criteria in such a way that we introduce effects that preserve a formula/proposition in one direction.

For example, consider the non-stutter effect $dec(x, \Delta c)$. We call such an effect *negatively-preserving* in respect to $x \geq 5$ because if $x \geq 5$ already does not hold for a state s, then it does not hold on state $e(s)$ as well. Thus, negatively-preserving effects preserve the *false* value of a formula/proposition on any state. If a negatively-preserving effect can also turn a formula from *true* to *false* for some state s, we call the effect a threat because it can invalidate a formula, but once a formula is *false*, it remains *false* by the application of the threat.

Similarly, we define *positively-preserving* effects and *supports* the other way round: A support preserves the *true* value of a formula and has the potential to turn it from *false* to *true* on some state s. For example, $inc(x, \Delta c)$ supports $x \geq 5$ because once x is larger than 5 so will it be when x is increased by $\Delta c > 0$ and an increase can turn $x \geq 5$ from *false* to *true*. To solve the extended partial-order reduction verification problem, consider the initial state. If $x \geq 5$ does not hold on the initial state, then the verification problem is trivially solved as we already found a counter-example. We assume $x \geq 5$ holds on the initial state. What is the worst-case sequence of effects that could happen to $x \geq 5$ in terms of satisfiability? In the worst-case all threats, i.e., effects that have the potential to turn the formula from *true* to *false* (decrease operations) have been applied to the initial state. If $x \geq 5$ holds on the state after that worst-case sequence, we could then easily add increment operations to x (supports) in the sequence or increment / decrement operations to other variables (called neutral effects), without changing satisfiability of $x \geq 5$. Thus,

$$\langle dec(x,1), dec(x,2), dec(x,3)\rangle(s) \models x \geq 5 \quad \rightarrow \quad \forall seq \in Seq(E) : seq(s) \models x \geq 5$$

The implication trivially holds in the opposite direction as well because the sequence on the left side is comprised in the set of all sequences on the right side. For this type of verification to work additional requirements have to be satisfied by effects and formulas than the pure definitions of supports and threats given above. Also notice that decrement and increment effects are commutative (as required by the independence criterion of partial-order reduction). However, state commutativity is not a requirement for the correctness of extended partial-order reduction. Instead, it suffices to enforce some requirements on the commutativity of effects in respect to the evaluation of a formula.

Assuming that extended partial-order reduction verification is correct (for some special subset of effects and formulas yet to be characterized in the remainder of this chapter), it suffices to create a single state by applying all threats of a formula in an arbitrary order to the initial state to determine whether a formula holds under every other sequence of effects. Thus, using extended partial-order reduction, the formula can be verified in time linear in the number of threats assuming that each threat of a formula can be determined and applied in constant time.

2.3 Relationships between Effects and Formulas

To reason about the influences effects have on the evaluation of formulas, their relationships to formulas need to be further categorized. This section provides definitions of how effects can influence the evaluation of a formula. We start by introducing effects that either preserve the *true* value of a formula in Section 2.3.1 or the *false* value of a formula in Section 2.3.2. After that, we further partition the set of all effects in Section 2.3.3 and prove propositions about the partition Section 2.3.4.

2.3.1 Positively-preserving (PP) Effects

Definition 1. ***Positively-preserving (PP) effect***
An effect $e \in E$ is called positively-preserving (PP) in respect to a formula ϕ iff

$$\begin{aligned} &\forall s \in S : (s \models \phi \rightarrow e(s) \models \phi) \quad \equiv \\ &\forall s \in S : (e(s) \not\models \phi \rightarrow s \not\models \phi) \end{aligned} \tag{2.1}$$

Consequently, e does not positively preserve ϕ iff

$$\exists s \in S : (s \models \phi \wedge e(s) \not\models \phi) \tag{2.2}$$

A PP effect is called decisive iff

$$\forall s \in S : e(s) \models \phi \tag{2.3}$$

otherwise indecisive, i.e., iff

$$\exists s \in S : e(s) \not\models \phi \tag{2.4}$$

We denote by $PPs(\phi) \subseteq E$ all effects in E that positively preserve formula ϕ and by $\overline{PPs(\phi)} = E \setminus PPs(\phi)$ its complement in E.

PP effects are good effects in the sense that they do not destroy the truth value of a formula once it holds.

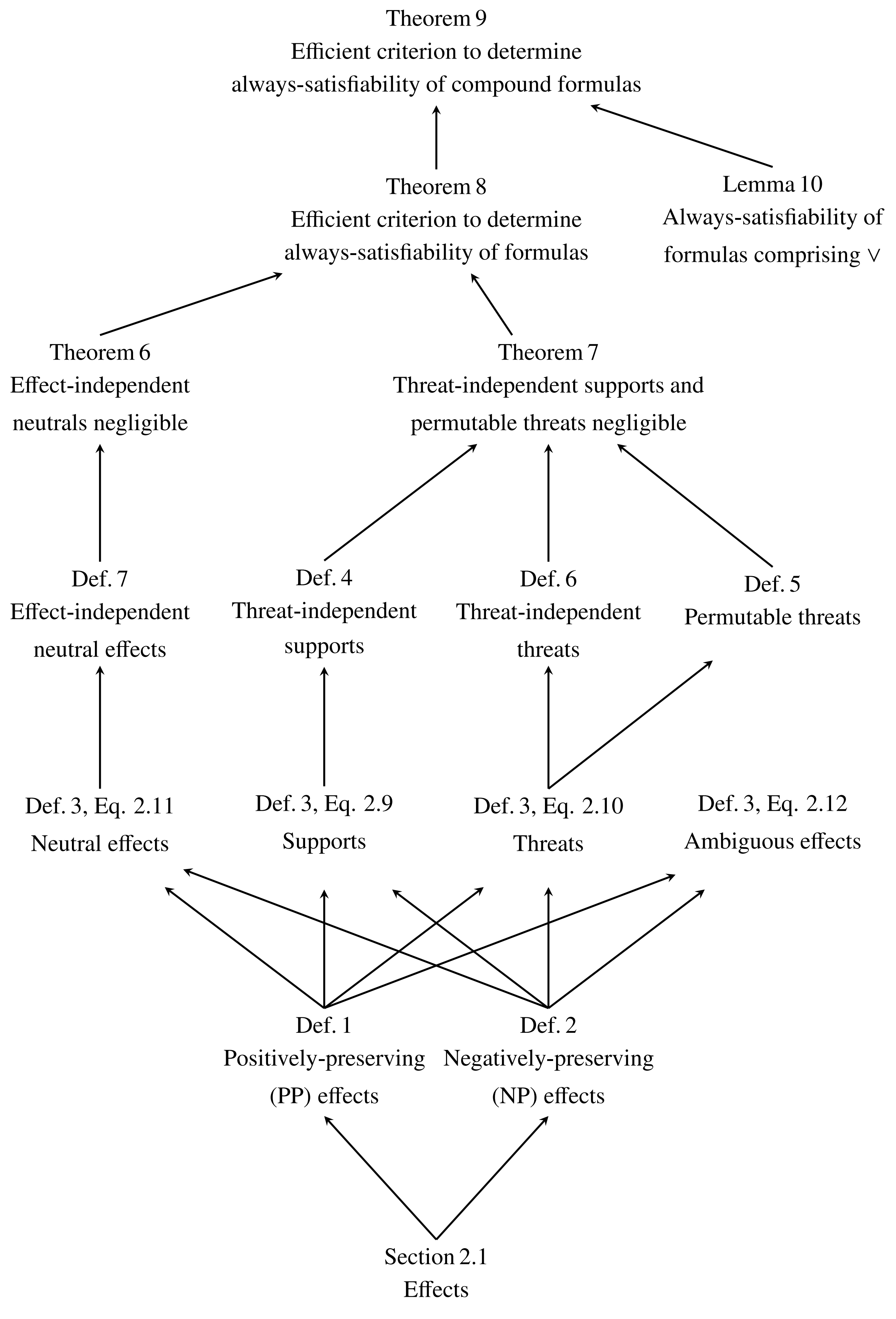


Figure 2.3: Definitions, theorems, and lemmas leading to an efficiently computable criterion (Theorem 9) to determine always-satisfiability of compound formulas.

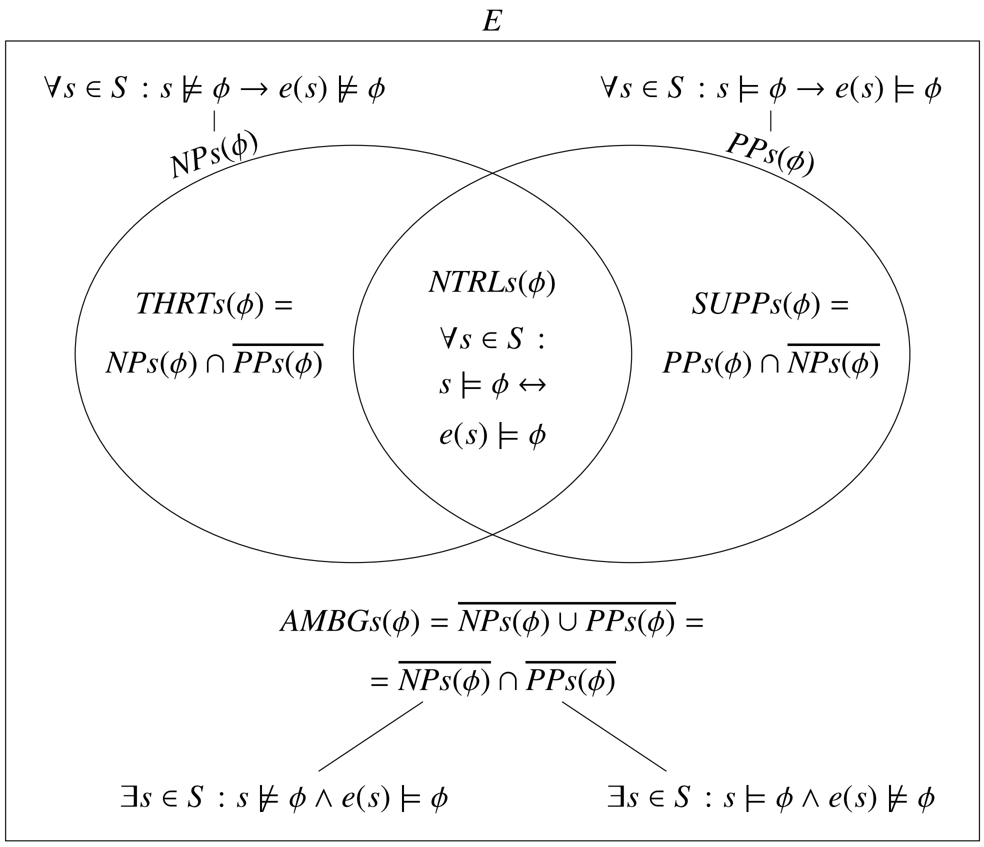

E
$\forall s \in S : s \not\models \phi \rightarrow e(s) \not\models \phi$
$\forall s \in S : s \models \phi \rightarrow e(s) \models \phi$
$NPs(\phi)$
$PPs(\phi)$
$THRTs(\phi) = NPs(\phi) \cap \overline{PPs(\phi)}$
$NTRLs(\phi)$ $\forall s \in S : s \models \phi \leftrightarrow e(s) \models \phi$
$SUPPs(\phi) = PPs(\phi) \cap \overline{NPs(\phi)}$
$AMBGs(\phi) = \overline{NPs(\phi) \cup PPs(\phi)} = \overline{NPs(\phi)} \cap \overline{PPs(\phi)}$
$\exists s \in S : s \not\models \phi \wedge e(s) \models \phi$
$\exists s \in S : s \models \phi \wedge e(s) \not\models \phi$

Figure 2.4: Partition of all effects in E (rectangle) in respect to a formula ϕ.

2.3.2 Negatively-preserving (NP) Effects

Definition 2. ***Negatively-preserving (NP) effect***
An effect $e \in E$ is called negatively-preserving (NP) in respect to a formula ϕ iff

$$\begin{aligned} \forall s \in S : (s \not\models \phi \rightarrow e(s) \not\models \phi) \quad &\equiv \\ \forall s \in S : (e(s) \models \phi \rightarrow s \models \phi) \end{aligned} \tag{2.5}$$

Consequently, e does not negatively preserve ϕ iff

$$\exists s \in S : (s \not\models \phi \wedge e(s) \models \phi) \tag{2.6}$$

A NP effect is called decisive iff

$$\forall s \in S : e(s) \not\models \phi \tag{2.7}$$

otherwise indecisive, i.e., iff

$$\exists s \in S : e(s) \models \phi \tag{2.8}$$

We denote by $NPs(\phi) \subseteq E$ the set of all effects in E that negatively preserve formula ϕ and by $\overline{NPs(\phi)} = E \setminus NPs(\phi)$ its complement in E.

NP effects are bad effects in the sense that they never turn formulas from *false* to *true*.

2.3.3 Neutrals, supports, threats, and ambiguous effects

Effects are further categorized based on the notion of positively-preserving and negatively-preserving effects as follows.

Definition 3. ***Neutrals, Supports, Threats, and Ambiguous Effects***
Let $e \in E$ be an effect and ϕ a formula, then

- ***Supports (supporting effects):***
  $SUPPs(\phi) \subseteq E$, the set of all supports of formula ϕ, is defined as

  $$SUPPs(\phi) = PPs(\phi) \cap \overline{NPs(\phi)}. \tag{2.9}$$

  $e \in SUPPs(\phi)$ is called decisive support iff e decisively positively preserves ϕ (see Definition 1). Otherwise, e is called indecisive support.

- ***Threats (threatening effects):***
  $THRTs(\phi) \subseteq E$, the set of all threats of formula ϕ, is defined as

  $$THRTs(\phi) = \overline{PPs(\phi)} \cap NPs(\phi). \tag{2.10}$$

  $e \in THRTs(\phi)$ is called decisive threat iff e decisively negatively preserves ϕ (see Definition 2). Otherwise, e is called indecisive threat.

- ***Neutrals (neutral effects):***
  $NTRLs(\phi) \subseteq E$, the set of all neutrals of formula ϕ, is defined as

  $$NTRLs(\phi) = PPs(\phi) \cap NPs(\phi). \tag{2.11}$$

- ***Ambiguous effects:***
  $AMBGs(\phi) \subseteq E$, the set of all ambiguous effects of formula ϕ, is defined as

  $$AMBGs(\phi) = \overline{PPs(\phi) \cup NPs(\phi)}. \tag{2.12}$$

Figure 2.4 depicts the partition of all effects in E in threats, supports, neutrals, and ambiguous effects in respect to a formula ϕ. We now explain in more detail the different types of effects introduced in Definition 3:

A support $supp \in SUPPs(\phi)$ of a formula ϕ preserves the *true* value of ϕ, but, different to a purely positively-preserving effect, there also exists at least one state where the application of *supp* turns ϕ from *false* to *true* in the subsequent state. Similarly, a threat $thr \in THRTs(\phi)$ of a formula ϕ preserves the *false* value of ϕ but, different to a purely negatively-preserving effect, there also exists at least one state where the application of *thr* turns ϕ from *true* to *false* in the subsequent state. Neutral effects are PP and NP effects at the same time. They are easy to reason about because they always preserve the truth value of a formula when applied to any state. All remaining effects are called ambiguous effects. These effects are difficult to reason about as they show different behavior depending on a state. For an ambiguous effect $ambg \in AMBGs(\phi)$ of a formula ϕ there exists a state where the application of the effect turns ϕ from *true* to *false*, but there also exists a state for which *ambg* behaves the other way round. Notice that an effect $e \in E$ can appear in different roles in respect to different formulas.

Corollary 1. ***Decisive threat in threat sequence invalidates a formula***
Let ϕ be a formula, $seq = \langle t_1, \ldots, t_n \rangle \in Seq(THRTs(\phi))$ a sequence of threats, such that $\exists k \in \{1, \ldots, n\} : t_k$ is a decisive threat of ϕ. Then,

$$\forall s \in S : \quad \langle t_1, \ldots, t_n \rangle(s) \not\models \phi$$

Proof. To prove: $\langle t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_n \rangle(s) \not\models \phi$. Because t_k is a decisive threat of formula ϕ, we have that $\forall s \in S : t_k(s) \not\models \phi$ (see Definition 2, Equation 2.7). Thus, with $s' = \langle t_1, \ldots, t_k \rangle(s)$ we obtain $s' \not\models \phi$ and it remains to be shown that

$$s' \not\models \phi \rightarrow \langle t_{k+1}, \ldots, t_n \rangle(s') \not\models \phi$$

Because each t_j, $j \in \{k+1, \ldots, n\}$ is a threat (and thus negatively-preserving), the implication directly follows from the definition of negatively-preserving effects (see Definition 2). □

2.3.4 Relationships Among Effects

In this section we describe how PP/NP effects relate to supports, threats, neutral, and ambiguous effects. The relationships proven in Proposition 2 are depicted in Figure 2.4.

Proposition 2. ***Relationships among effects***
Let ϕ be a formula, then:

1. $NTRLs(\phi) \cup SUPPs(\phi) = PPs(\phi)$
2. $NTRLs(\phi) \cup THRTs(\phi) = NPs(\phi)$
3. $NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) = PPs(\phi) \cup NPs(\phi)$
4. $NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) \cup AMBGs(\phi) = E$
5. $THRTs(\phi) \cap NTRLs(\phi) = \emptyset$
6. $SUPPs(\phi) \cap NTRLs(\phi) = \emptyset$
7. $SUPPs(\phi) \cap THRTs(\phi) = \emptyset$

Proof. The proof directly follows from the definitions of the sets and can be found in Appendix A.1. □

The following proposition describes how effects relate to each other when a formula is negated.

Proposition 3. ***Relationships for negated formulas***
Let ϕ be a formula, then:

1. $PPs(\phi) = NPs(\neg\phi)$
2. $SUPPs(\phi) = THRTs(\neg\phi)$
3. $NTRLs(\phi) = NTRLs(\neg\phi)$
4. $AMBGs(\phi) = AMBGs(\neg\phi)$

Proof. The proof directly follows from the definitions of the sets and can be found in Appendix A.2. □

Most important of the relationships is the fact that threats and supports change roles when a formula is negated and that neutral effects remain the same under negation. The observation saves work when we need to prove the relationships for concrete formulas and effects of a change verification logic because we can directly conclude from a formula to its negated formula. Notice that Proposition 3 does not yet state whether the decisive/indecisive property of a support or threat remains unaffected under negation. We address this in the subsequent proposition:

Proposition 4. ***Support and threat complements***
Let $e \in E$ be an effect and ϕ a formula, then

1. *e decisively supports ϕ $\quad\leftrightarrow\quad$ e decisively threatens $\neg\phi$*
2. *e indecisively supports ϕ $\quad\leftrightarrow\quad$ e indecisively threatens $\neg\phi$*

Proof. The proof directly follows from Proposition 3 and the definition of decisive and indecisive effects. It can be found in Appendix A.3. □

2.4 Single Effects in Sequences of Effects

This section introduces stronger definitions of supports, threats, and neutrals, namely the definitions of threat-independent supports (Section 2.4.1), threat-independent and permutable threats (Section 2.4.2), and effect-independent neutrals (Section 2.4.3). The presence of these stronger versions of supports, threats, and neutrals are an important requirement to apply extended partial-order reduction.

2.4.1 Threat-independent Supports

The presence of threat-independent supports is an important requirement to efficiently solve the verification problem previously defined in Section 2.1.

Definition 4. ***Threat-independent support***
A support $supp \in SUPPs(\phi)$ is called threat-independent iff

$$\forall ts \in Seq(THRTs(\phi)), \quad \forall s \in S : \quad ts(s) \models \phi \quad \rightarrow \quad ts(supp(s)) \models \phi$$

We denote by $SUPPs_{ind}(\phi) \subseteq SUPPs(\phi) \subseteq E$ the set of all threat-independent supports of ϕ.

Notice that the definition of a threat-independent support further narrows down the definition of a support (Definition 3). If threat sequence ts is the empty sequence, Definition 4 equals the definition of a PP effect. For a support to be also threat-independent, the original definition of a PP effect ($s \models \phi \rightarrow supp(s) \models \phi$) still has to hold after the same but arbitrary sequence of threats has been applied to the states on both sides of the implication (s and $supp(s)$).
Differently speaking, Definition 4 is equal to the observation that a threat-independent support can be inserted before an arbitrary sequence of threats if ϕ has already been satisfied after the application of the threat sequence. This is logically equivalent to the observation that a threat-independent support that is followed by a sequence of threats can be removed if ϕ does not hold after the whole sequence.

2.4.2 Threat-independent and Permutable Threats

The presence of permutable and threat-independent threats is an important requirement to efficiently solve the verification problem previously defined in Section 2.1.

Definition 5. ***Permutable threats***
Let ϕ be a formula, $ts \in Seq(THRTs(\phi))$ a sequence of threats. We denote by $\rho(ts)$ the set of all permutations of threat sequence ts. Threat sequence ts is called permutable iff

$$\forall s \in S : \quad (ts(s) \models \phi \quad \leftrightarrow \quad \forall ts' \in \rho(ts) : ts'(s) \models \phi)$$

Analogous to the definition of threat-independent supports (see Definition 4) we introduce the notion of threat-independent threats:

Definition 6. ***Threat-independent threats***
A threat $thr \in THRTs(\phi)$ is called threat-independent iff

$$\forall ts \in Seq(THRTs(\phi) \setminus \{thr\}), \quad \forall s \in S : \quad ts(s) \not\models \phi \rightarrow ts(thr(s)) \not\models \phi$$

We denote by $THRTs_{ind}(\phi) \subseteq THRTs(\phi) \subseteq E$ the set of all threat-independent threats of ϕ.

Notice that the definition of a threat-independent threat further narrows down the definition of a threat (Definition 3). If threat sequence *ts* is the empty sequence, Definition 6 equals the definition of a NP effect. For a threat to be also threat-independent, the original definition of a NP effect ($s \not\models \phi \rightarrow thr(s) \not\models \phi$) still has to hold after the same but arbitrary sequence of threats (not comprising *thr*) has been applied to the states on both sides of the implication (s and $thr(s)$).
Differently speaking, Definition 6 is equal to the observation that a threat-independent threat can be inserted before an arbitrary sequence of threats if ϕ has already been falsified after the application of the threat sequence. This is logically equivalent to the observation that a threat-independent threat at the beginning of a threat sequence can be removed if ϕ does already hold after the complete sequence of threats has been applied.

We now prove a corollary of Definition 6 that provides a simple criterion to determine threat-independence of threats:

Corollary 5. ***Permutable threats imply threat-independence***
Let ϕ be a formula and $THRTs(\phi)$ permutable. Then, $THRTs_{ind}(\phi) = THRTs(\phi)$, i.e., all threats are threat-independent.

Proof. Let $thr \in THRTs(\phi)$ be an arbitrary threat. To prove:

$$\forall ts \in Seq(THRTs(\phi) \setminus \{thr\}), \quad \forall s \in S : \quad ts(s) \not\models \phi \rightarrow ts(thr(s)) \not\models \phi$$

Without limitations let $ts = \langle t_1, \ldots, t_n \rangle$, then

$$\begin{array}{rcll} ts(s) \not\models \phi & \rightarrow & ts(thr(s)) \not\models \phi & \equiv_{ts=\langle t_1,\ldots,t_n \rangle} \\ ts(s) \not\models \phi & \rightarrow & \langle thr, t_1, \ldots, t_n \rangle(s) \not\models \phi & \equiv_{THRTs(\phi)\text{ permutable}} \\ ts(s) \not\models \phi & \rightarrow & \langle t_1, \ldots, t_n, thr \rangle(s) \not\models \phi & \equiv_{ts=\langle t_1,\ldots,t_n \rangle} \\ ts(s) \not\models \phi & \rightarrow & thr(ts(s)) \not\models \phi & \equiv_{s'=ts(s)} \\ s' \not\models \phi & \rightarrow & thr(s') \not\models \phi & \end{array}$$

The last formula is trivially satisfied because *thr* is a threat and thus a negatively-preserving effect (see Definition 2). □

2.4.3 Effect-independent Neutrals

This section introduces a stronger definition of neutrals, so called effect-independent neutrals. A neutral effect preserves the evaluation of a formula when applied to any state. This also holds for an effect-independent neutral, but, in addition to that, any sequence of effects can be applied to both sides of the implication while the implication still holds. More formally:

Definition 7. ***Effect-independent neutral***
A neutral effect $ntr \in NTRLs(\phi)$ is called effect-independent iff

$$\forall s \in S, \quad \forall seq \in Seq(E \setminus \{ntr\}) : \quad seq(s) \models \phi \quad \leftrightarrow \quad seq(ntr(s)) \models \phi$$

We denote by $NTRLs_{ind}(\phi) \subseteq NTRLs(\phi) \subseteq E$ the set of all effect-independent neutrals of ϕ.

Notice that the definition of an effect-independent neutral goes beyond the definition of a neutral effect. If the effect sequence *seq* is the empty sequence, Definition 7 equals the definition of a neutral effect (Definition 3, Equation 2.11). For a neutral to be also effect-independent, the original definition ($s \models \phi \leftrightarrow ntr(s) \models \phi$) still has to hold after an arbitrary effect sequence (not comprising *ntr*) has been applied to the states on both sides of the implication (s and $ntr(s)$).

Differently speaking, Definition 7 is equal to the observation that an effect-independent neutral can be added at the beginning of any effect sequence while preserving the truth value of ϕ. This is logically equivalent to the observation that an effect-independent neutral can be removed from the beginning of a sequence of effects without changing the truth value of ϕ.

2.5 A Criterion for Always-satisfiability of Formulas

This section introduces a criterion to verify whether a formula ϕ holds under every possible sequence of concurrent effects. The remainder of this section is organized as follows. In Section 2.5.1 we define always-satisfiability of formulas. After that, several theorems involving the roles of threat-independent supports, threat-independent threats, and effect-independent neutrals when determining always-satisfiability are proven. Finally, a correct, and easily computable criterion to determine always-satisfiability of a formula is given in Section 2.5.3.

2.5.1 Always-satisfiability of Formulas

Definition 8. ***Always-satisfiability of a formula***
Let ϕ be a formula and $s \in S$ a state. ϕ is called always-satisfiable iff ϕ holds on any state evolving from the execution of effects in E to s in any order. More formally, ϕ is always-satisfiable iff

$$\forall seq \in Seq(E) : \quad seq(s) \models \phi$$

Notice that the term always-satisfiability is used instead of satisfiability to avoid confusion with satisfiability as know from SAT solvers. A SAT solver computes an assignment of truth values to the atoms of a formula such that the formula holds. In turn, in the context of this work, a formula is always-satisfiable if it holds over the potentially factorial many states caused by the application of effects in E to the initial state.

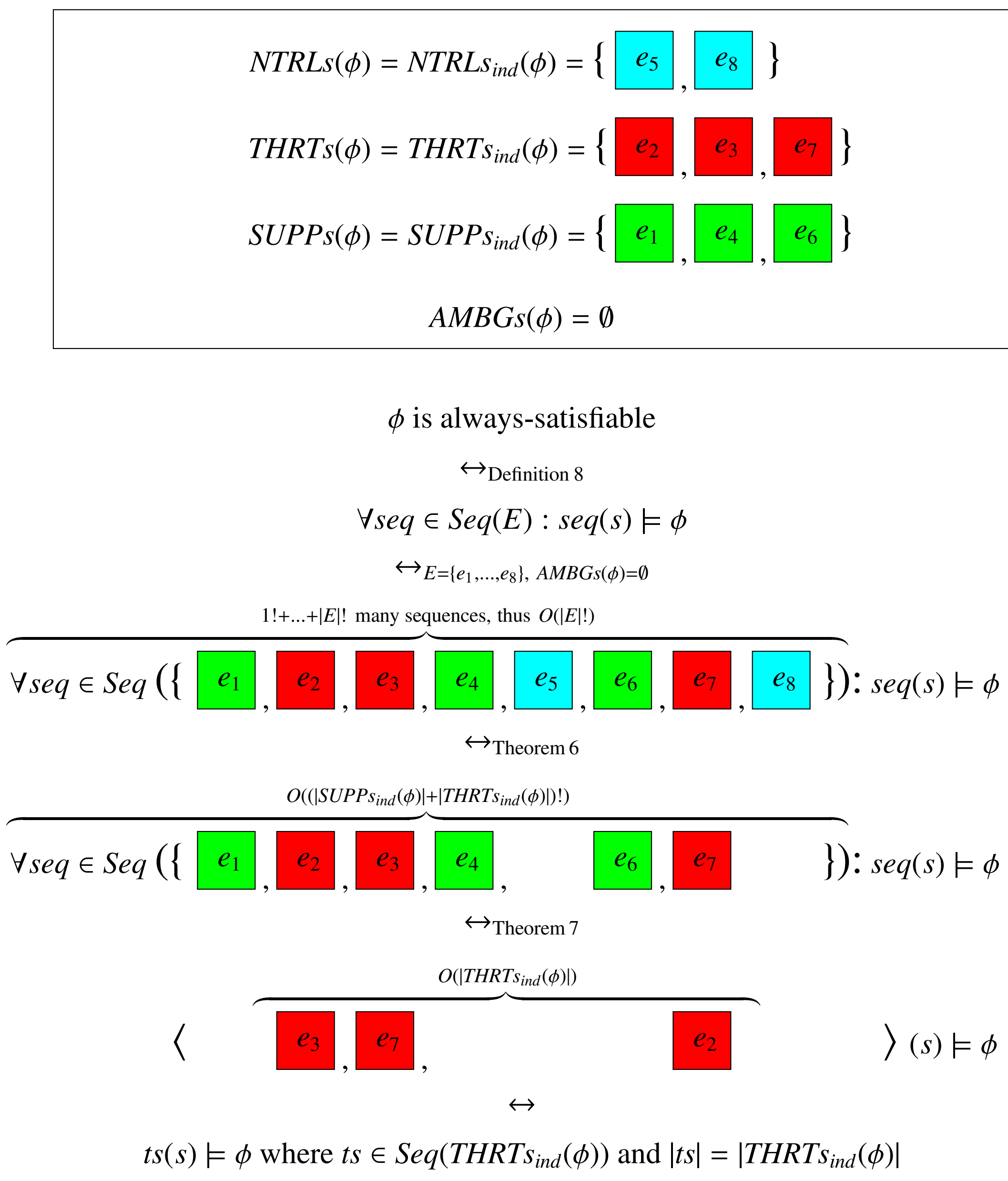

$NTRLs(\phi) = NTRLs_{ind}(\phi) = \{ e_5, e_8 \}$
$THRTs(\phi) = THRTs_{ind}(\phi) = \{ e_2, e_3, e_7 \}$
$SUPPs(\phi) = SUPPs_{ind}(\phi) = \{ e_1, e_4, e_6 \}$
$AMBGs(\phi) = \emptyset$
ϕ is always-satisfiable
$\leftrightarrow_{\text{Definition 8}}$
$\forall seq \in Seq(E) : seq(s) \models \phi$
$\leftrightarrow_{E=\{e_1,\ldots,e_8\},\ AMBGs(\phi)=\emptyset}$
$1!+\ldots+|E|!$ many sequences, thus $O(|E|!)$
$\forall seq \in Seq(\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}): seq(s) \models \phi$
$\leftrightarrow_{\text{Theorem 6}}$
$O((|SUPPs_{ind}(\phi)|+|THRTs_{ind}(\phi)|)!)$
$\forall seq \in Seq(\{e_1, e_2, e_3, e_4, e_6, e_7\}): seq(s) \models \phi$
$\leftrightarrow_{\text{Theorem 7}}$
$O(|THRTs_{ind}(\phi)|)$
$\langle e_3, e_7, e_2 \rangle (s) \models \phi$
$\leftrightarrow$
$ts(s) \models \phi$ where $ts \in Seq(THRTs_{ind}(\phi))$ and $|ts| = |THRTs_{ind}(\phi)|$

Figure 2.5: Visualization of the proof of Theorem 8.

2.5.2 Influence of Effects on Always-satisfiability

This section proves two important theorems that are later necessary to introduce an efficiently computable criterion to decide always-satisfiability of formulas. First, we prove that the appearance of effect-independent neutral effects within a sequence of effects does not influence the evaluation of a formula:

Theorem 6. ***Effect-independent neutrals negligible to decide always-satisfiability***
Let ϕ be a formula, $NTRLs_{ind}(\phi) = NTRLs(\phi)$ (all neutrals of ϕ effect-independent), and $AMBGs(\phi) = \emptyset$, then

$$\forall seq \in Seq(E), \quad \forall s \in S : \quad (seq(s) \models \phi \quad \leftrightarrow \quad (seq \setminus NTRLs_{ind}(\phi))(s) \models \phi)$$

Proof. Let $seq = \langle e_1, \ldots, e_n \rangle \in Seq(E) = Seq(THRTs(\phi) \cup SUPPs(\phi) \cup NTRLs_{ind}(\phi))$.
(1): $\forall k \in \{1, \ldots, n\} : e_k \notin NTRLs_{ind}(\phi)$. Then, $seq = seq \setminus NTRLs_{ind}(\phi)$, leaving nothing to proof.
(2): $\exists k \in \{1, \ldots, n\} : e_k \in NTRLs_{ind}(\phi)$. Then,

$$\begin{aligned}
\langle e_1, \ldots, e_{k-1}, e_k, e_{k+1}, \ldots, e_n \rangle(s) \models \phi \quad & \equiv_{s' = \langle e_1, \ldots, e_{k-1} \rangle(s)} \\
\langle e_{k+1}, \ldots, e_n \rangle(e_k(s')) \models \phi \quad & \equiv_{\text{Def. 7}} \\
\langle e_{k+1}, \ldots, e_n \rangle(s') \models \phi \quad & \equiv_{s' = \langle e_1, \ldots, e_{k-1} \rangle(s)} \\
\langle e_1, \ldots, e_{k-1}, e_{k+1}, \ldots, e_n \rangle(s) \models \phi \quad &
\end{aligned}$$

The complete proof follows by applying the equivalence steps to every neutral effect in the effect sequence starting from the beginning of the effect sequence until all neutrals have been eliminated.

□

We now prove another theorem that reduces the number of effect sequences that need to be considered when deciding always-satisfiability. Instead of testing whether a formula holds on all states that originate from the initial state by the application of any sequence of permutable threats and threat-independent supports, the formula can be verified on a state that has seen the execution of a randomly chosen sequence of all threats to the initial state.

Theorem 7. ***Threat-independent supports and permutable threats negligible to decide always-satisfiability***
Let ϕ be a formula, $s \in S$ a state, and $ts \in Seq(THRTs(\phi))$ an arbitrary threat sequence of length $|ts| = |THRTs(\phi)|$, i.e., ts comprises every threat in $THRTs(\phi)$ exactly once. Furthermore, let $THRTs(\phi)$ be permutable and $SUPPs_{ind}(\phi) = SUPPs(\phi)$ (all supports of ϕ threat-independent). Then,

$$ts(s) \models \phi \quad \leftrightarrow \quad \forall seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi)) : \quad seq(s) \models \phi$$

Proof. Direction $\leftarrow$: This directly follows from the fact that

$$ts \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi))$$

Direction $\rightarrow$: Assumption: $ts(s) \models \phi$. To prove:

$$\forall seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi)) : \quad seq(s) \models \phi$$

We proof this direction by showing that any sequence $seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi))$ can be constructed from $ts \in Seq(THRTs(\phi))$, where $|ts| = |THRTs(\phi)|$, by the multiple application of three transformation operations (c_1, c_2, and c_3) to threat sequence *ts* while positively preserving the evaluation of ϕ. More formally, we show the existence of transformation operations c_1, c_2, and c_3 such that for any arbitrary sequence *seq*, $seq = c_3^b(c_2(c_1^a(ts)))$ ($a, b \in \mathbb{N}_0$ the number of applications of c_1 and c_3 necessary to create *seq*) and

$$ts(s) \models \phi \quad \rightarrow \quad c_3^b(c_2(c_1^a(ts)))(s) \models \phi$$

(Step 1, c_1): Let $ts = \langle t_1, \ldots, t_n \rangle \in Seq(THRTs(\phi))$, $|ts| = |THRTs(\phi)|$. Because $THRTs(\phi)$ is permutable by assumption, we obtain $THRTs_{ind}(\phi) = THRTs(\phi)$ (all threats are threat-independent) due to Corollary 5. The first step removes all threats from *ts* that are not contained in *seq* anymore by the multiple application of c_1, i.e., c_1 removes a single threat $t_k \in ts$, $t_k \notin seq$. Let $\langle t_1, \ldots, t_n \rangle(s) \models \phi$. To prove: $c_1(\langle t_1, \ldots, t_n \rangle)(s) \models \phi$

$$\begin{aligned}
\langle t_1, \ldots, t_n \rangle(s) \models \phi \quad & \equiv \\
\langle t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_n \rangle(s) \models \phi \quad & \equiv_{s'=\langle t_1,\ldots,t_{k-1}\rangle(s)} \\
\langle t_{k+1}, \ldots, t_n \rangle(t_k(s')) \models \phi \quad & \rightarrow_{\text{Def. 6},\ t_k \in THRTs_{ind}(\phi)} \\
\langle t_{k+1}, \ldots, t_n \rangle(s') \models \phi \quad & \equiv_{s'=\langle t_1,\ldots,t_{k-1}\rangle(s)} \\
\langle t_1, \ldots, t_{k-1}, t_{k+1}, \ldots, t_n \rangle(s) \models \phi \quad & \equiv \\
c_1(\langle t_1, \ldots, t_n \rangle)(s) \models \phi \quad &
\end{aligned}$$

By applying c_1 a–times (starting from the beginning of the threat sequence) a new threat sequence $ts^* = \langle t_1, \ldots, t_n \rangle \setminus \{t'_1, \ldots, t'_k\}$ is obtained where $\{t'_1, \ldots, t'_k\} \subseteq THRTs_{ind}(\phi)$ are the threats of *ts* that are not present in *seq* anymore. Thus, $ts^*(s) \models \phi$.

(Step 2, c_2): The second step reorders the threats in ts^* in such a way that their order matches the order of threats in $seq \setminus SUPPs_{ind}(\phi)$. Let ρ be the permutation that describes the reordering. We chose $\rho = c_2$. It remains to be proven that:

$$ts^*(s) \models \phi \quad \rightarrow \quad (\rho(ts^*))(s) \models \phi.$$

This directly follows from Definition 5 because by assumption $THRTs(\phi)$ is permutable.

(Step 3, c_3): Let $ts^{**} = \rho(ts^*) = c_2(c_1^a(ts))$ be the threat sequence after the application of Step 1 and Step 2 to *ts*. To finally obtain *seq* from ts^{**}, every support in *seq* has to be added to ts^{**} at the appropriate position. To prove:

$$\begin{aligned}
& \forall supp \in SUPPs_{ind}(\phi) : \\
& \langle t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_n \rangle(s) \models \phi \quad \rightarrow \quad \langle t_1, \ldots, t_{k-1}, supp, t_k, \ldots, t_n \rangle(s) \models \phi
\end{aligned}$$

where t_k is the threat in threat sequence ts^{**} before which *supp* is to be inserted. Then,

$$\begin{aligned}
\langle t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_n \rangle(s) \models \phi \quad & \equiv_{s'=\langle t_1,\ldots,t_{k-1}\rangle(s)} \\
\langle t_k, \ldots, t_n \rangle(s') \models \phi \quad & \rightarrow_{\text{Def. 4},\ supp \in SUPPs_{ind}(\phi)} \\
\langle t_k, \ldots, t_n \rangle(supp(s')) \models \phi \quad & \equiv_{s'=\langle t_1,\ldots,t_{k-1}\rangle(s)} \\
\langle t_1, \ldots, t_{k-1}, supp, t_k, \ldots, t_n \rangle(s) \models \phi \quad &
\end{aligned}$$

By applying the steps above to every $supp \in seq$ starting from the beginning of the threat sequence we achieve by construction that $seq = c_3^b(c_2(c_1^a(ts)))$ and by applying each of the three steps above to every sequence $seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi))$ we finally obtain that

$$ts(s) \models \phi \qquad \rightarrow \quad \forall seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi)) : \quad seq(s) \models \phi$$

□

2.5.3 An Always-satisfiability Criterion for Formulas

This section provides an efficiently computable criterion to determine always-satisfiability of a formula.

Theorem 8. ***An efficient criterion to determine always-satisfiability of formulas***
Let ϕ be a formula, $s \in S$ a state, and ts an arbitrary sequence of threats, such that $ts \in Seq(THRTs(\phi))$ and $|ts| = |THRTs(\phi)|$. Furthermore, let

- $AMBGs(\phi) = \emptyset$
- $SUPPs_{ind}(\phi) = SUPPs(\phi)$ *(all supports of ϕ are threat-independent supports)*
- $NTRLs_{ind}(\phi) = NTRLs(\phi)$ *(all neutrals of ϕ are effect-independent effects)*
- $THRTs(\phi)$ *permutable*

Then,

$$\phi \textit{ is always-satisfiable} \qquad \text{iff} \qquad ts(s) \models \phi$$

Proof.

$$\begin{aligned}
\phi \text{ is always-satisfiable} & \quad \equiv_{\text{Def. 8}} \\
\forall seq \in Seq(E) : \quad seq(s) \models \phi & \quad \equiv_{AMBGs(\phi)=\emptyset\ \&\ \text{Prop. 2 (4.)}} \\
\forall seq \in Seq(SUPPs_{ind}(\phi) \cup THRTs(\phi) \cup NTRLs_{ind}(\phi)) : \quad seq(s) \models \phi & \quad \equiv_{\text{Theorem 6}} \\
\forall seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi) \cup NTRLs_{ind}(\phi)) : & \\
(seq \setminus NTRLs_{ind}(\phi))(s) \models \phi & \quad \equiv \\
\forall seq \in Seq(THRTs(\phi) \cup SUPPs_{ind}(\phi)) : \quad seq(s) \models \phi & \quad \equiv_{\text{Theorem 7}} \\
ts(s) \models \phi &
\end{aligned}$$

where $ts \in Seq(THRTs(\phi))$ and $|ts| = |THRTs(\phi)|$. □

Theorem 8 provides a criterion to determine whether a formula is always-satisfiable by looking at a single state instead of factorial many. To determine the truth value of a formula under any combination of effects in E, it suffices to apply all threats of the formula in any order to the initial state and evaluate the formula on that state. However, for Theorem 8 to be applicable, threats, supports, neutrals, and ambiguous effects need to be determined in respect to formula ϕ. Performing this categorization can be very difficult for complex formulas. In the subsequent section we simplify Theorem 8 for formulas that are built of atomic propositions using the standard logical connectors $\wedge$ and $\vee$.

2.6 An Efficient Criterion for Always-satisfiability of Compound Formulas

In this section we introduce a recursive criterion to determine always-satisfiability of compound formulas.

Theorem 9. ***Efficient criterion to determine always-satisfiability of compound formulas***
Let

$$\phi \quad = \quad pdc_i \quad | \quad \psi_1 \wedge \psi_2 \quad | \quad \phi_1 \vee \phi_2.$$

be a formula in negation normal form where $pdc_i \in \{pdc_1, \ldots, pdc_n\}$, $n \in \mathbb{N}$, are atomic propositions that either hold or not in a state $s \in S$.

- *Let ψ_1 and ψ_2 be arbitrary formulas. Furthermore, let ϕ_1 and ϕ_2 be formulas such that either $E \subseteq NTRLs(\phi_1)$ (all effects are neutral effects of ϕ_1) or $E \subseteq NTRLs(\phi_2)$ (all effects are neutral effects of ϕ_2)*
- *Let $ts_i \in Seq(THRTs(pdc_i))$ such that $|ts_i| = |THRTs(pdc_i)|$*
- *For every atomic proposition $pdc \in \{pdc_1, \ldots, pdc_n\}$ let*
  - *$AMBGs(pdc) = \emptyset$*
  - *$THRTs(pdc)$ permutable*
  - *$SUPPs_{ind}(pdc) = SUPPs(pdc)$ (all supports of pdc are threat-independent)*
  - *$NTRLs_{ind}(pdc) = NTRLs(pdc)$ (all neutrals of pdc are effect-independent)*

Then,

1. *$pdc_i \in \{pdc_1, \ldots, pdc_n\}$ is always-satisfiable* iff $\quad ts_i(s) \models pdc_i$.
2. *$\psi_1 \wedge \psi_2$ is always-satisfiable* iff $\quad$ *ψ_1 is always-satisfiable* $\quad$ and $\quad$ *ψ_2 is always-satisfiable.*
3. *$\phi_1 \vee \phi_2$ is always-satisfiable* iff $\quad$ *ϕ_1 is always-satisfiable* $\quad$ or $\quad$ *ϕ_2 is always-satisfiable.*

Proof. We proof each statement of the theorem:

1. Directly follows from the assumption and Theorem 8.
2. For $\psi_1 \wedge \psi_2$:

$$\begin{aligned}
(\psi_1 \wedge \psi_2) \text{ is always-satisfiable} \quad & \equiv_{\text{Def. 8}} \\
\forall seq \in Seq(E): \quad seq(s) \models (\psi_1 \wedge \psi_2) \quad & \equiv \\
\forall seq \in Seq(E): \quad (seq(s) \models \psi_1 \wedge seq(s) \models \psi_2) \quad & \equiv_{\forall \text{ and } \wedge \text{ distributive}} \\
(\forall seq \in Seq(E): seq(s) \models \psi_1) \quad \wedge \quad (\forall seq \in Seq(E): seq(s) \models \psi_2) \quad & \equiv_{\text{Def. 8}} \\
(\psi_1 \text{ is always-satisfiable}) \quad \wedge \quad (\psi_2 \text{ is always-satisfiable}) \quad &
\end{aligned}$$

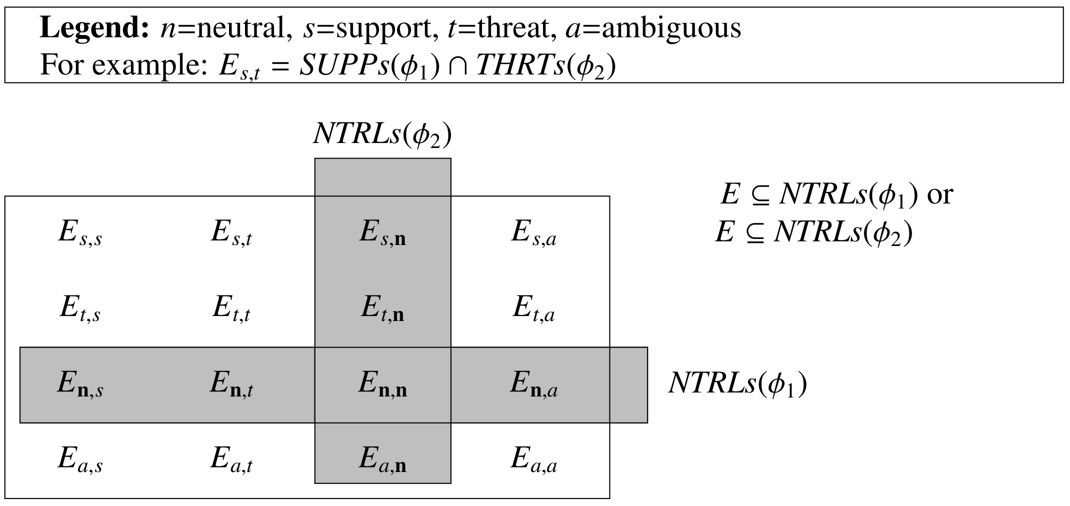

Legend: n=neutral, s=support, t=threat, a=ambiguous
For example: $E_{s,t} = SUPPs(\phi_1) \cap THRTs(\phi_2)$
$NTRLs(\phi_2)$
$E_{s,s}$ $E_{s,t}$ $E_{s,\mathbf{n}}$ $E_{s,a}$
$E_{t,s}$ $E_{t,t}$ $E_{t,\mathbf{n}}$ $E_{t,a}$
$E_{\mathbf{n},s}$ $E_{\mathbf{n},t}$ $E_{\mathbf{n},\mathbf{n}}$ $E_{\mathbf{n},a}$
$E_{a,s}$ $E_{a,t}$ $E_{a,\mathbf{n}}$ $E_{a,a}$
$E \subseteq NTRLs(\phi_1)$ or
$E \subseteq NTRLs(\phi_2)$
$NTRLs(\phi_1)$

Figure 2.6: Limited types of effects (shaded gray) that are allowed to exist in respect to formulas ϕ_1 and ϕ_2 such that the efficient verification of formula $\phi_1 \vee \phi_2$ remains feasible using Theorem 9 and Lemma 10.

3. For $\phi_1 \vee \phi_2$ and $E \subseteq NTRLs(\phi_1)$ or $E \subseteq NTRLs(\phi_2)$:

$$\begin{aligned}
(\phi_1 \vee \phi_2) \text{ is always-satisfiable} &\quad \equiv_{\text{Def. 8}} \\
\forall seq \in Seq(E): \quad seq(s) \models (\phi_1 \vee \phi_2) &\quad \equiv \\
\forall seq \in Seq(E): \quad (seq(s) \models \phi_1 \vee seq(s) \models \phi_2) &\quad \equiv_{\text{Lemma 10}} \\
(\forall seq \in Seq(E): seq(s) \models \phi_1) \quad \vee \quad (\forall seq \in Seq(E): seq(s) \models \phi_2) &\quad \equiv_{\text{Def. 8}} \\
(\phi_1 \text{ is always-satisfiable}) \vee (\phi_2 \text{ is always-satisfiable}) &
\end{aligned}$$

□

Theorem 9 can be used to efficiently verify formulas compound of atomic propositions with the logical operators $\wedge$ and $\vee$. Lemma 10, which has been used in the proof of Theorem 9, remains to be proven.

Lemma 10. ***Always-satisfiability of formulas comprising*** $\vee$
Let $s \in S$ be an arbitrary state, ϕ_1 and ϕ_2 formulas such that $E \subseteq NTRLs(\phi_1)$ (all effects are neutrals of ϕ_1) or $E \subseteq NTRLs(\phi_2)$ (all effects are neutrals of ϕ_2). Then,

$$\begin{gathered}
\forall seq \in Seq(E): \quad (seq(s) \models \phi_1 \vee seq(s) \models \phi_2) \quad \leftrightarrow \\
(\forall seq \in Seq(E): seq(s) \models \phi_1) \quad \vee \quad (\forall seq \in Seq(E): seq(s) \models \phi_2)
\end{gathered}$$

Proof. We proof both cases ($E \subseteq NTRLs(\phi_1)$ and $E \subseteq NTRLs(\phi_2)$) separately and for both directions at the same time.

- Let $E \subseteq NTRLs(\phi_1)$. Then,

$$\begin{aligned}
\forall seq \in Seq(E): \quad (seq(s) \models \phi_1 \vee seq(s) \models \phi_2) &\quad \equiv_{E \subseteq NTRLs(\phi_1)} \\
\forall seq \in Seq(E): \quad (s \models \phi_1 \vee seq(s) \models \phi_2) &\quad \equiv \\
s \models \phi_1 \quad \vee \quad (\forall seq \in Seq(E): \quad seq(s) \models \phi_2) &\quad \equiv_{E \subseteq NTRLs(\phi_1)} \\
(\forall seq \in Seq(E): \quad seq(s) \models \phi_1) \quad \vee \quad (\forall seq \in Seq(E): \quad seq(s) \models \phi_2) &
\end{aligned}$$

- Let $E \subseteq NTRLs(\phi_2)$. Then,

$$\begin{aligned}
\forall seq \in Seq(E): \quad (seq(s) \models \phi_1 \lor seq(s) \models \phi_2) \quad &\equiv_{E \subseteq NTRLs(\phi_2)} \\
\forall seq \in Seq(E): \quad (seq(s) \models \phi_1 \lor s \models \phi_2) \quad &\equiv \\
(\forall seq \in Seq(E): \quad seq(s) \models \phi_1) \quad \lor \quad s \models \phi_2 \quad &\equiv_{E \subseteq NTRLs(\phi_2)} \\
(\forall seq \in Seq(E): \quad seq(s) \models \phi_1) \quad \lor \quad (\forall seq \in Seq(E): \quad seq(s) \models \phi_2) &
\end{aligned}$$

□

2.7 Conclusions

In this chapter we introduced the definitions and theorems of extended partial-order reduction, an extended version of the partial-order reduction model checking paradigm. Extended partial-order reduction enables the efficient verification of CTL formulas of type $\forall\Box\phi$ and LTL formulas of type $\Box\phi$ in a state transition system with effects executable at any time but at most once. Different to partial-order reduction, extended partial-order reduction does not require state commutativity and the stutter criterion to hold, but further assumptions between effects and atomic propositions / formulas need to hold.

We showed that extended partial-order reduction is applicable in cases where partial-order reduction fails to further reduce the search space. Extended partial-order reduction was proven to be correct and Theorem 9 provides a simple criterion to compute always-satisfiability of a formula bottom up starting with the atomic propositions. Left unanswered in this chapter remains the question for which kind of effects and formulas extended partial-order reduction can be applied. In the subsequent chapter we introduce a many-sorted logic for the efficient verification of IT change operations that satisfies the requirements of extended partial-order reduction.

CHAPTER 3

A Many-Sorted Logic for the Efficient Verification of IT Change Operations

This chapter introduces the change verification logic, a many-sorted logic adhering to the requirements of extended partial-order reduction (see Chapter 2) that can be used to efficiently verify IT change operations.
Section 3.1 provides syntax and semantics of the change verification logic. After that, formal definitions of change activities, safety constraints, and the verification problems solvable with the change verification logic are provided in Section 3.2. Section 3.3 introduces the effects and predicates of the change verification logic and Section 3.4 proves that they comply with the requirements of extended partial-order reduction.[1]

3.1 A Many-Sorted Logic for IT Change Verification

This section introduces syntax and semantics of an efficiently verifiable many-sorted change verification logic.

3.1.1 Syntax

This section introduces the syntax of the many-sorted change verification language. Whenever appropriate, we highlight deviations of the change verification logic from the usual definition of many-sorted logic.

Definition 9. ***Signature***
A signature Σ is a tuple $\Sigma = (S, P, C, PREDs)$ where

1. $S = \{\sigma_1, \ldots, \sigma_n\}$ *is a nonempty set of sorts.*

2. $P = \{p, l, p_1, \ldots, p_n\}$ *is a nonempty set of constant symbols whose sorts are sorts in* S*.*

[1]Parts of this chapter previously appeared in [51].

3. *$C = \{c, c_1, \Delta c_1, \ldots, c_n, \Delta c_n\}$ is a nonempty set of fixed-constant symbols whose sorts are sorts in S.*

4. *PREDs is a nonempty set of predicate symbols whose arities are constructed using sorts of S.*

For a signature $\Sigma = (S, P, C, PREDs)$ we also write Σ^S for S, Σ^P for P, Σ^C for C, and Σ^{PREDs} for PREDs.

Notice that the definition of a signature in change verification logic differs in two aspects compared to signatures in ordinary many-sorted logic: First, functions are not comprised in the signatures of the change verification language. Second, we distinguish two different types of constants: P (called constants describing properties of Configuration Items) and C (called fixed-constants) that later will be interpreted differently (Definitions 12 and 13).

Definition 10. ***Σ-terms***
Let Σ be a signature. The set of Σ-terms of sort σ is defined as follows.

- *Every constant symbol $p \in \Sigma^P$ of sort σ is a Σ-term of sort σ.*
- *Every fixed-constant symbol $c \in \Sigma^C$ of sort σ is a Σ-term of sort σ.*

Different to Σ-terms in ordinary many-sorted logic, function symbols and variables are not comprised in the Σ-terms of the verification language.

Definition 11. ***Σ-formulas***
Σ-formulas are the strings obtained by finitely many applications of the following rules:

1. *If $pdc \in \Sigma^{PREDs}$ is a predicate symbol of arity $\sigma_1 \times \ldots \times \sigma_n$ and t_i, $i \in \{1, \ldots, n\}$ is a Σ-term of sort σ_i, then $pdc(t_1, \ldots, t_n)$ is a Σ-formula.*
2. *If ϕ is a Σ-formula, then $\neg\phi$ is a Σ-formula.*
3. *If ϕ and ψ are Σ-formulas, then $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, and $(\phi \leftrightarrow \psi)$ are Σ-formulas as well.*

Different to Σ-formulas in ordinary many-sorted logic, Σ-formulas in change verification logic do not comprise quantifiers over variables as the change verification language does not comprise variables. In the remainder of this work we assume without limitations that every Σ-formula ϕ is given in negation normal form. For Σ-formulas we use the letters ϕ and ψ.

3.1.2 Semantics

This section introduces the semantics of the IT change verification logic. We start by introducing Σ-interpretations in Definition 12. After that, Definition 13 describes states / configurations of the CMDB followed by effects (Definition 14. Finally, Definition 15 describes the semantics of formulas in change verification logic.

Definition 12. ***Σ-interpretation***
Let Σ be a signature. A Σ-interpretation $\mathcal{A}$ is a map satisfying the following properties:

1. *Each sort $\sigma \in \Sigma^S$ is mapped to a nonempty domain $D_\sigma^{\mathcal{A}}$.*

2. *Each constant symbol* $p \in \Sigma^P$ *of sort* σ *is mapped to an element* $p^{\mathcal{A}} \in D_\sigma^{\mathcal{A}}$.
3. *Each fixed-constant symbol* $c \in \Sigma^C$ *of sort* σ *is mapped to an element* $c^{\mathcal{A}} \in D_\sigma^{\mathcal{A}}$
4. *Each predicate symbol* $pdc \in \Sigma^{PREDs}$ *of arity* $\sigma_1 \times \ldots \times \sigma_n$, $n \in \mathbb{N}$, *is mapped to a subset* $pdc^{\mathcal{A}} \subseteq D_{\sigma_1}^{\mathcal{A}} \times \ldots \times D_{\sigma_n}^{\mathcal{A}}$

Different to Σ-interpretations of ordinary many-sorted logic, a Σ-interpretation in change verification logic does not comprise mappings for variable symbols or function symbols as both do not exist in verification logic.

Definition 13. ***States***
Let Σ *be a signature. Let* $D = (D_\sigma)_{\sigma \in \Sigma^S}$ *be a family of* pre-defined domains *where each* D_σ *is a non-empty set; let* $V = (v_c)_{c \in \Sigma^C}$ *be a family of* pre-defined constant values *where* $v_c \in D_\sigma$ *if c is of sort* σ *and* $V' = (v'_p)_{p \in \Sigma^P}$ *a family of* constant values *where* $v'_p \in D_\sigma$ *if p is of sort* σ*; let* $R = (R_{pdc})_{pdc \in \Sigma^{PREDs}}$ *be a family of* pre-defined predicate relations *where* $R_{pdc} \subseteq D_{\sigma_1} \times \cdots \times D_{\sigma_n}$ *if pdc has arity* $\sigma_1 \times \cdots \times \sigma_n$.

The Σ-(D, V, V', R)-states*, denoted by* $States(\Sigma, D, R, V, V')$*, are given by the set of* Σ *- interpretations* $\mathcal{A}$ *such that*

- $D_\sigma^{\mathcal{A}} = D_\sigma$ *for all* $\sigma \in \Sigma^S$
- $c^{\mathcal{A}} = v_c$ *for all* $c \in \Sigma^C$
- $pdc^{\mathcal{A}} = R_{pdc}$ *for all* $pdc \in \Sigma^{PREDs}$
- $\forall(\mathfrak{a}_1, \mathfrak{a}_2) \in \mathcal{A} \times \mathcal{A}, \quad \mathfrak{a}_1 \neq \mathfrak{a}_2 : \quad \exists v'_p \in V' : \quad v'^{\mathfrak{a}_1}_p \neq v'^{\mathfrak{a}_2}_p$

In the remainder of this work we denote by $D_\sigma^{def} = D_\sigma$ *the default domain of sort* σ*, by* $c^{def} = v_c$ *the default interpretation of fixed-constant c and by* $pdc^{def} = R_{pdc}$ *the default interpretation of predicate pdc.*

Definition 14. ***Effects***
An Σ-(D, V, V', R)-effect *is given by a function* $e : States(\Sigma, D, R, V, V') \rightarrow States(\Sigma, D, R, V, V')$.

For Configuration Management Databases, we assume some signature Σ_{CMDB} and some pre-defined domains, constant values, and predicate relations $(D_{\text{CMDB}}, V_{\text{CMDB}}, V'_{\text{CMDB}}, R_{\text{CMDB}})$ for Σ_{CMDB}. The signature captures the configuration of the IT infrastructure and services as an object relational graph. Clearly, the signature and the previous definitions depend on the actual CMDB under consideration.

For the remainder of this section, we assume one fixed set-up and therefore abbreviate $States(\Sigma_{\text{CMDB}}, D_{\text{CMDB}}, R_{\text{CMDB}}, V_{\text{CMDB}}, V'_{\text{CMDB}})$ by *CMDBs*. In particular, a *CMDBs*-effect is a function $e : CMDBs \rightarrow CMDBs$.

Definition 15. ***Semantics of change verification language***
Let $cmdb \in CMDBs$ *be a* Σ*-interpretation and* ϕ *a* Σ*-formula. Without limitations let* ϕ *be in negation normal form. We write*

$$cmdb \models \phi$$

iff ϕ evaluates to true for Σ-interpretation cmdb where

$$\begin{aligned}
cmdb \models pdc(t_1,\dots,t_n) &\quad \text{iff} \quad (t_1^{cmdb},\dots,t_n^{cmdb}) \in pdc^{def} \\
cmdb \models \phi \wedge \psi &\quad \text{iff} \quad cmdb \models \phi \text{ and } cmdb \models \psi \\
cmdb \models \phi \vee \psi &\quad \text{iff} \quad cmdb \models \phi \text{ or } cmdb \models \psi
\end{aligned}$$

3.2 Change Activities, Safety Constraints, and IT Change Verification Problems

This section introduces the logical specification of change activities and safety constraints in verification logic (Section 3.2.1). After that, Section 3.2.2 introduces three IT change verification problems that can be efficiently decided using the change verification logic and the theorems of extended partial-order reduction.

3.2.1 IT Changes and Safety Constraints

We provide formal definitions of change activities and safety constraints:

Definition 16. ***Change activity***
A change activity act is a tuple act = (ϕ_{act}, E_{act}) where

- *ϕ_{act} is a Σ-formula in change verification logic (see Definition 11) that describes the precondition of change activity act.*
- *E_{act} is a finite set of Σ-$(D_{CMDB}, V_{CMDB}, V'_{CMDB}, R_{CMDB})$-effects that describe the effects of act on the CMDB.*

We denote by ACTs the set of all change activities.

Definition 17. ***Safety constraint (SC):*** *A safety constraint sc is a Σ–formula ϕ_{sc}.*

We denote by SCs the set of all safety constraints.

3.2.2 Verification Problems for IT Change Management

This section defines three verification problems in the context of IT Change Management that can be efficiently solved using extended partial-order reduction and the change verification logic.

Definition 18. *Let ACTs = $\{act_1,\dots,act_n\}$, $act_i = (\phi_{act_i}, E_{act_i})$, be a set of pending, i.e., yet to execute, IT change activities and SCs = $\{\phi_{sc_1},\dots,\phi_{sc_k}\}$ a set of safety constraints, i.e., Σ-formulas. Let $cmdb_{init}$ be the current configuration of the CMDB, i.e., a Σ-interpretation. We define the following three verification problems for IT Change Management:*

- ***IT change verification problem:***
  For every safety constraint $\phi_{sc} \in SCs$: Does ϕ_{sc} hold on all configurations of the CMDB evolving from $cmdb_{init}$ by the execution of effects of change activities in any order and

length?
More formally deciding whether

$$\forall \phi_{sc} \in SCs, \quad \forall seq \in Seq(E_{act_1} \cup \ldots \cup E_{act_n}) \quad : \quad seq(cmdb_{init}) \models \phi_{sc}$$

holds.

This problem is equivalent to model checking the computation tree logic (CTL) formula $\forall\Box\phi_{sc}$ or the linear temporal logic (LTL) formula $\Box\phi_{sc}$ for every safety constraint formula $\phi_{sc} \in SCs$

- ***IT change conflict detection problem**:*
  For every change activity $act_i \in ACTs$: Is act_i feasible, i.e., does ϕ_{act_i} hold on every configuration of the CMDB evolving from $cmdb_{init}$ by the execution of effects of change activities in $ACTs \setminus \{act\}$ in any order and length?
  More formally verifying whether

  $$\forall act_i \in ACTs, \quad \forall seq \in Seq((E_{act_1} \cup \ldots \cup E_{act_1}) \setminus E_{act_i}) : \quad seq(cmdb_{init}) \models \phi_{act_i}$$

  holds. This problem is equivalent to model checking the computation tree logic (CTL) formula $\forall\Box\phi_{act_i}$ or the linear temporal logic (LTL) formula $\Box\phi_{act_i}$ for every precondition ϕ_{act_i} of a change activity $act_i \in ACTs$.

- ***IT change verification and conflict detection problem:***
  Solve both problems at the same time. This problem is equivalent to model checking all previously mentioned formulas.

All problems can be solved using the previously introduced Theorem 9 (Section 2.6). However, for Theorem 9 to be applicable to the change verification logic, the predicates and effects of the change verification logic need to comply with the requirements of extended partial-order reduction. The next section introduces the predicates and effects and proves their compliance with the theorems underlying extended partial-order reduction.

3.3 Effects and Predicates of Change Verification Logic

3.3.1 Predicates

This section describes in more detail the predicates of the change verification logic. They are the basic building blocks of Σ-formulas (see Definition 11) used in the logical specification of safety constraints and change activities. Table 3.1 provides an overview of the predicates supported by the change verification logic. The predicates are categorized as follows.

- **Arithmetic predicates:** The change verification language supports four binary predicates ($\geq, >, \leq$, and $<$) to compare number constants (i.e., properties of Configuration Items) against constants or fixed-constants. For example, let p_1 be a constant and p_2 a constant or fixed-constant, then $p_1 \geq p_2, p_1 > p_2, p_1 \leq p_2$, and $p_1 < p_2$ are valid formulas of the number comparison predicates. The predicates operate upon a generic sort numbers that can be interpreted by a Σ-interpretation with different totally ordered sets, usually $\mathbb{N}, \mathbb{Z}, \mathbb{Q}$, or $\mathbb{R}$.

Category of predicate	Predicate	Description
Integer comparison	$p_1 \geq / > p_2$	Value of constant p_1 is greater than (or equal) the value of (fixed) constant p_2.
	$p_1 \leq / < p_2$	Value of constant p_1 is less than (or equal) the value of (fixed) constant p_2.
Checking value of a property	*hasValue*(p, c)	*true* if constant p (property of a Configuration Item) holds the value of fixed-constant c (i.e., c^{def}), otherwise *false*.
	$\neg$*hasValue*(p, c)	*true* if constant p (property of a Configuration Item) does not hold the value of fixed-constant c (i.e., c^{def}), otherwise *false*.
Set and list elements	*contains*(l, c)	*true* if set / list l contains c^{def}, otherwise *false*.
	$\neg$*contains*(l, c)	*true* if set / list l not contains c^{def}, otherwise *false*.

Table 3.1: Predicates of the change verification logic and their semantics.

Category of effect	Effect	Description
Integer arithmetic	*inc*$(p, \Delta c)$	Increases the value of constant p by the value of fixed-constant Δc, i.e., p holds $p^{cmdb} + \Delta c^{def}$ ($\Delta c^{def} > 0$) after application of *inc*$(p, \Delta c)$ to configuration *cmdb*. *dec*$(p, \Delta c)$ decreases the value of constant p, i.e., p holds $p^{cmdb} - \Delta c^{def}$ ($\Delta c^{def} > 0$) after application of *dec*$(p, \Delta c)$ to configuration *cmdb*.
Set value of a property	*set*(p, c)	Sets the value of a constant p (i.e., of a property of a Configuration Item) to c^{def}, the value of fixed-constant c. Property p holds value c^{def} after the application.
Set and list operations	*add*(l, c)	Adds the value of fixed-constant c to a set / list l. For sets: If l already contains c^{def}, then l remains unchanged, otherwise l contains c^{def} exactly once after adding c^{def} to l. For lists: If the list contains n instances of c^{def}, it contains $n + 1$ instances after adding c^{def}.
	remove(l, c)	Removes the value of fixed-constant c^{def} from a set / list l. For sets: If l already contains c^{def}, then c^{def} is not contained in l after the application of *remove*(l, c), otherwise l remains unchanged by *remove*(l, c). For lists: If the list contains $n \geq 1$ instances of c^{def}, it contains $n - 1$ instances of c^{def} after *remove*(l, c). Otherwise, the list remains unchanged.

Table 3.2: Effects of change verification logic and their semantics.

Arithmetic predicates are frequently used to model constraints about the availability of resources or to determine the cheapest route offered to a network. For example, consider *mem* to be a constant that describes the memory property of a physical machine and 1024 a fixed-constant. Then, $mem \geq 1024$ describes the predicate that checks whether the physical machine has at least 1024 MB of memory. Similarly, let $cost_1$ and $cost_2$ be properties of two network interface Configuration Items describing the cost of an interface. Then, $cost_1 > cost_2$ describes a predicate that checks whether the cost of the first interface is more expensive than that of the second.

- **Checking the value of properties:** The change verification language supports predicates to determine whether a property of a Configuration Item, i.e., a constant, holds the value of a fixed-constant or not. For example, let p be a constant and c a fixed-constant, then $hasValue(p, c)$ is true in a configuration *cmdb* iff property p holds the value c^{def} of fixed-constant c, i.e., iff $p^{cmdb} = c^{def}$. Otherwise predicate $\neg hasValue(p, c)$ is true. Predicates of that type are useful when to determine whether 1:N relationships among Configuration Items hold, e.g., if a virtual machine runs on a specific physical machine or if a configuration parameter is set to a specific value.

  For example, let *runs_on* be the property of a virtual machine Configuration Item that references the physical machine the virtual machine runs on. Furthermore let *pm* be the fixed-constant describing a physical machine. Then, predicate $hasValue(runs\_on, pm)$ checks whether the virtual machine runs on *pm*.

- **Checking lists and sets for elements:** The change verification language supports predicates to determine whether a list or set property of a Configuration Item, i.e., a list or set constant, contains the value of a fixed-constant or not. For example, let l be a list or set constant and c a fixed-constant, then $contains(l, c)$ is true in a Σ-configuration *cmdb* iff l^{cmdb} contains c^{def}. Otherwise predicate $\neg contains(l, c)$ is true. The list predicates are useful when to determine whether N:M relationships among CIs hold or to model a list of protocols that are routed using a routing table.

Readers interested in specific example formulas that make use of the predicates to describe change activities and safety constraints in change verification logic are referred to Section 5.1, which provides change activities and safety constraints of the Amazon outage.

3.3.2 Effects

This section describes the effects of the change verification logic. Effects describe how change activities modify properties of Configuration Items stored in the CMDB or, more formally, how the value of constants assigned by Σ-interpretations are altered by the application of effects. Table 3.2 provides an overview of the effects supported by the change verification logic. The effects are categorized as follows.

- **Arithmetic effects:** The change verification language supports the modification of number properties of Configuration Items using relative increase and decrease effects. For example, let p be a number constant, e.g., interpreted in $\mathbb{N}, \mathbb{Q}, \mathbb{Z}$, or $\mathbb{R}$ by a Σ-interpretation. Furthermore let Δc be a fixed number constant, i.e., a constant whose value is always the same in any CMDB. Then, $inc(p, \Delta c)$ and $dec(p, \Delta c)$ are effects that increment / decrement the value of p by the value of Δc, $\Delta c^{def} > 0$. Thus, for an arbitrary $cmdb \in CMDBs$

$$p^{inc(p,\Delta c)(cmdb)} = p^{cmdb} + \Delta c^{def} \text{ and } p^{dec(p,\Delta c)(cmdb)} = p^{cmdb} - \Delta c^{def}.$$

The arithmetic effects can be used to describe resource allocation changes, e.g., the decrease of RAM or CPU resources caused by a deployment change activity or the decrease / increase of the metric of a network interface. For example, consider *mem* to be a constant that describes the memory property of a physical machine and 1024 to be a fixed-constant. Then, $dec(mem, 1024)$ describes an effect that decreases the value of *mem* by 1024. Such an effect occurs, for example, when describing a change activity to deploy a virtual machine on a physical machine.

- **Setting the value of properties:** The change verification logic supports effects that assign the value of a fixed-constant to a constant, i.e., effects to assign new values to the properties of Configuration Items. For example, let p be a constant and c a fixed-constant, then $set(p, c)$ is the effect that assigns property p of a Configuration Item the value c^{def} of fixed-constant c, i.e., for a $cmdb \in CMDBs$: $p^{set(p,c)(cmdb)} = c^{def}$. This effect can be used, for example, when a virtual machine is assigned to a physical machine or an application is configured to use a specific port.

  For example, consider the property *runs_on* of a virtual machine Configuration Item that describes the physical machine the virtual machine runs on. A change activity to migrate a virtual machine to a new host can use the effect $set(runs\_on, pm)$ to assign the value of fixed-constant *pm* to the *runs_on* property of the virtual machine.

- **Set and list effects:** The change verification logic supports effects to add and remove fixed-constants from list/set constants, i.e., from list/set properties of Configuration Items. For example, let l be a list or set and c a fixed-constant, then $add(l, c)$ and $remove(l, c)$ are the effects that add/remove the value c^{def} of fixed-constant c to/from the value of list l. Add and remove effects to change sets and lists can be frequently used to describe relationships among many Configuration Items.

  For example, consider a property *hosts* of type set of a physical machine Configuration Item that holds the virtual machines running on the physical machine. For a fixed-constant *vm* (describing a virtual machine) we can use the $add(hosts, vm)$ effect to add the value of *vm* to the set of virtual machines running on the physical machine. Such an effect can be used to describe a migration change that assigns *vm* to a different physical machine.

We successfully used the effects of the change verification logic to model a rich set of different IT change activities in our previous work [45, 46, 47, 48, 49, 50, 51]. The effects of the change verification logic can describe a wide range of change activities because they cover many modifications to an object-oriented instance graph, frequently used to describe the state of a data center in commercial CMDB products [60] or object-oriented IT infrastructure graphs [21]. Thus, assuming that a change activity has an impact on the configuration of a CMDB, it can be modeled using all effects provided by the change verification logic (see Table 3.2).

3.4 Compliance of Change Verification Logic with Extended Partial-Order Reduction

This section proves that the predicates and effects of the change verification logic satisfy the requirements imposed by extended partial-order reduction in Chapter 2.

3.4.1 Relationships between Effects and Predicates

	Predicates	
Effects	$p_1 \geq / > p_2$	$p_1 \leq / < p_2$
$inc(p_2, \Delta c_2)$	**threat(¬d)**	**support(¬d)**
$inc(p_1, \Delta c_1)$	**support(¬d)**	**threat(¬d)**
$dec(p_2, \Delta c_2)$	**support(¬d)**	**threat(¬d)**
$dec(p_1, \Delta c_1)$	**threat(¬d)**	**support(¬d)**

(a) Arithmetic effects and predicates. $\Delta c_2^{def} > 0$ and $\Delta c_1^{def} > 0$.

	Predicates	
Effects	$hasValue(p, c_1)$	$\neg hasValue(p, c_1)$
$set(p, c_1)$	**support(d)**	**threat(d)**
$set(p, c_2)$	**threat(d)**	**support(d)**

(b) Setting and checking of properties. $c_1^{def} \neq c_2^{def}$.

	Predicates	
Effects	$contains(p, c_1)$	$\neg contains(p, c_1)$
$add(p, c_1)$	**support(d)**	**threat(d)**
$add(p, c_2)$	**neutral**	**neutral**
$remove(p, c_1)$	**threat(d)** / [1]**threat(¬d)**	**support(d)** / [1]**support(¬d)**
$remove(p, c_2)$	**neutral**	**neutral**

(c) Effects and predicates on sets and [1]lists. $c_1^{def} \neq c_2^{def}$.

Legend:
d=decisive
¬d=indecisive
[1]**=list op**

(d)

Table 3.3: Support, threat, and neutral relationships among the effects and predicates of the IT change verification logic.

Proposition 11. ***Correctness of support, threat, and neutral relationships:***
The support, threat, and neutral relationships among effects and predicates as depicted in Table 3.3 are correct.

Proof. Proofing Table 3.3 is a straightforward process that is achieved by proofing the definitions of threats, neutrals, and supports for every entry in the subtables of Table 3.3. For the sake of brevity we do not provide the proofs here but in Appendix A.4. □

3.4.2 Predicates Have No Ambiguous Effects

Corollary 12. ***Predicates do not have ambiguous effects***
Let pdc be an arbitrary predicate in Table 3.3. Then, $AMBGs(pdc) = \emptyset$.

Proof. Let pdc and e be a predicate effect pair within Table 3.3a, Table 3.3b, or Table 3.3c. Then, according to Proposition 11

$$e \in (SUPPs(pdc) \cup THRTs(pdc) \cup NTRLs(pdc)).$$

Let pdc and e be a predicate effect pair from different tables. Then, $e \in NTRLs(pdc)$ because we disallow the mixture of predicates and effects across different categories of effects and predicates, i.e., across different tables (most of them do not make sense anyway as they are defined on different sorts).

□

3.4.3 Supports of Predicates are Threat-independent

Proposition 13. ***Supports of predicates are threat-independent***
Every support in Table 3.3 is threat-independent.

Proof. To prove:

$$\begin{aligned}
\forall \text{ predicates } pdc,\\
\forall supp \in SUPPs(pdc),\\
\forall cmdb \in CMDBs,\\
\forall ts \in Seq(THRTs(pdc)) : \quad & ts(cmdb) \models pdc \rightarrow ts(supp(cmdb)) \models pdc
\end{aligned}$$

For $ts = \langle\rangle$: To prove: $cmdb \models pdc \rightarrow supp(cmdb) \models pdc$. This is trivially satisfied by a support which is positively-preserving (see Definition 1).
For $n > 0$ there are two cases:
(1) $\exists i \in \{1, \ldots, n\} : t_i \in \langle t_1, \ldots, t_n\rangle$ and t_i is a decisive threat to pdc. In this case according to Corollary 1: $\forall cmdb \in CMDBs : ts(cmdb) \not\models pdc$ which trivially satisfies the implication.
(2) $\forall i \in \{1, \ldots, n\} : t_i \in \langle t_1, \ldots, t_n\rangle$ indecisively threatens pdc. We prove this case for every predicate in Table 3.3 that has indecisive threats:
(2a) For $p_1 \geq p_2$ or $p_1 > p_2$. Without limitations we only prove case $p_1 \geq p_2$ because the proof for $p_1 > p_2$ can be obtained from $p_1 \geq p_2$ when $\geq$ is substituted by $>$. Let

$$ts(cmdb) \models p_1 \geq p_2 \quad \leftrightarrow_{\text{Def. 15}} \quad \geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)})$$

Then, it remains to be shown that

$$ts(supp(cmdb)) \models p_1 \geq p_2 \quad \leftrightarrow_{\text{Def. 15}} \quad \geq^{def}(p_1^{ts(supp(cmdb))}, p_2^{ts(supp(cmdb))})$$

for any support $supp$ of predicate $p_1 \geq p_2$, i.e., $supp \in \{inc(p_1, \Delta c_1), dec(p_2, \Delta c_2)\}$.
(2a1) Let $supp = inc(p_1, \Delta c_1)$ where $\Delta c_1^{def} > 0$ is the absolute increase in p_1. Then,

$$\begin{aligned}
p_1^{ts(inc(p_1,\Delta c_1)(cmdb))} &= p_1^{ts(cmdb)} + \Delta c_1^{def} \qquad \text{and}\\
p_2^{ts(inc(p_1,\Delta c_1)(cmdb))} &= p_2^{ts(cmdb)}
\end{aligned}$$

With the assumption $\geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)})$ we obtain

$$\geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)}) \quad \rightarrow \quad \geq^{def}(p_1^{ts(cmdb)} + \Delta c_1^{def}, p_2^{ts(cmdb)})$$

(2a2) Let $supp = dec(p_2, \Delta c_2)$ where $\Delta c_2^{def} > 0$ is the absolute decrease in p_2. Then,

$$\begin{aligned}
p_1^{ts(dec(p_2,\Delta c_2)(cmdb))} &= p_1^{ts(cmdb)} \qquad \text{and}\\
p_2^{ts(dec(p_2,\Delta c_2)(cmdb))} &= p_2^{ts(cmdb)} - \Delta c_2^{def}
\end{aligned}$$

Consequently with the assumption $\geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)})$ we obtain

$$\geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)}) \quad \rightarrow \quad \geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)} - \Delta c_2^{def})$$

(2b) For $p_1 \leq p_2$ or $p_1 < p_2$. Without limitations we only proof case $p_1 \leq p_2$ because the proof

for $p_1 < p_2$ emerges from $p_1 \leq p_2$ when $\leq$ is substituted for $<$. Because $p_1 \leq p_2 \equiv p_2 \geq p_1$, the proof given in (2a) also holds for $p_1 \leq p_2$ respectively $p_1 < p_2$ when exchanging p_2 and p_1.
(2c) For *contains*(l, c). The only indecisive threat to *contains*(l, c) is *remove*(l, c) on lists. Let $ts \in Seq(THRTs(contains(l, c)))$, $|ts| = n$, and $\forall i \in \{1, \dots, n\} : t_i = remove(l, c)$. The only support for *contains*(l, c) is *add*(l, c). Thus, to prove:

$$\begin{aligned}
&\forall cmdb \in CMDBs : \\
&ts(cmdb) \models contains(l, c) \quad \rightarrow \quad ts(add(l, c)(cmdb)) \models contains(l, c) \qquad \leftrightarrow_{\text{Def. 15}} \\
&contains^{def}(l^{ts(cmdb)}, c^{def}) \quad \rightarrow \quad contains^{def}(l^{ts(add(l,c)(cmdb))}, c^{def})
\end{aligned}$$

For the left side of the implication to hold l^{cmdb} needs to comprise at least $n + 1$ instances of c^{def}, otherwise $contains^{def}(l^{ts(cmdb)}, c^{def})$ would not hold. Thus, $l^{add(l,c)(cmdb)}$ comprises at least $n + 2$ instances of c^{def}. Consequently, $contains^{def}(l^{ts(add(l,c)(cmdb))}, c^{def})$ holds because l comprises at least $(n + 2) - n = 2$ instances of c^{def} on configuration / Σ-interpretation $ts(add(l, c)(cmdb))$.

□

3.4.4 Threats of Predicates are Permutable

Proposition 14. ***Threats of predicates are permutable***
Let pdc be a predicate in Table 3.3 and $ts = \langle t_1, \dots, t_n \rangle \in Seq(THRTs(pdc))$ *an arbitrary threat sequence. Then, ts is permutable.*

Proof. To prove:

$$\forall cmdb \in CMDBs : (ts(cmdb) \models pdc \quad \leftrightarrow \quad \forall ts' \in \rho(ts) : \quad ts'(cmdb) \models pdc)$$

Let *cmdb* be an arbitrary configuration of the CMDB.
Direction $\leftarrow$: trivial because $ts \in \rho(ts)$
Direction $\rightarrow$: Let $ts(cmdb) \models pdc$. For $ts(cmdb) \models pdc$ to hold, every $t_i \in ts$ needs to be an indecisive threat, otherwise $ts(cmdb) \not\models pdc$ due to Corollary 1. We prove this direction for every predicate in Table 3.3 and any combination of indecisive threats.
(1) For integer arithmetic predicates we only proof the case for predicates $p_1 \geq p_2$ and $p_1 > p_2$. We note when changes need to be made to adapt the proof for $p_1 \leq p_2$ or $p_1 < p_2$.
Let $ts(cmdb) \models p_1 \geq p_2$, where $t_i \in ts$ and $t_i \in \{inc(p_2, \Delta c_2), dec(p_1, \Delta c_1)\}$
(for $p_1 \leq p_2$ and $p_1 < p_2 : t_i \in \{inc(p_1, \Delta c_1), dec(p_2, \Delta c_2)\}$). Then,

$$\forall ts' \in \rho(ts) : \quad (p_2^{ts(cmdb)} = p_2^{ts'(cmdb)} \quad \wedge \quad p_1^{ts(cmdb)} = p_1^{ts'(cmdb)})$$

because reordering $inc(p_2, \Delta c_2)$ and $dec(p_1, \Delta c_1)$ operations (likewise $inc(p_1, \Delta c_1)$ and $dec(p_2, \Delta c_2)$ operations for predicates $p_1 \leq p_2$ and $p_1 < p_2$) does not influence the value of p_2 and p_1 after the application of any permutation of the threat sequence. Thus,

$$\begin{aligned}
ts(cmdb) &\models p_1 \geq p_2 \quad \text{iff}_{\text{Def. 15}} \\
&\geq^{def}(p_1^{ts(cmdb)}, p_2^{ts(cmdb)}) \quad \rightarrow \\
\forall ts' \in \rho(ts) : \quad &\geq^{def}(p_1^{ts'(cmdb)}, p_2^{ts'(cmdb)}) \quad \text{iff}_{\text{Def. 15}} \\
\forall ts' \in \rho(ts) : \quad &ts'(cmdb) \models p_1 \geq p_2
\end{aligned}$$

(2) For list operations: Let $pdc = contains(l, c)$. Then, $\forall t_i \in \langle t_1, \ldots, t_n \rangle : t_i = remove(l, c)$ because $remove(l, c)$ is the only indecisive threat to $contains(l, c)$. Then, every threat sequence $ts' \in \rho(ts)$ removes the same number of instances of c^{def}. Consequently, $ts(cmdb) \models pdc$ iff $ts'(cmdb) \models pdc$ for every permutation ts' of ts. □

3.4.5 Neutrals of Predicates are Effect-independent

Proposition 15. ***Neutrals of predicates are effect independent***
Every neutral effect in Table 3.3 is effect-independent.

Proof. Let pdc be a predicate and $ntr \in NTRLs(pdc)$. To prove:

$$\forall cmdb \in CMDBs, \quad \forall seq \in Seq(E \backslash \{ntr\}) : \quad (seq(cmdb) \models pdc \quad \leftrightarrow \quad seq(ntr(cmdb)) \models pdc)$$

Let $seq = \langle e_1, \ldots, e_n \rangle \in Seq(E)$ and $cmdb$ an arbitrary configuration of the CMDB.
(1) Let $n = 0$. To prove: $\forall cmdb \in CMDBs : (cmdb \models pdc \leftrightarrow ntr(cmdb) \models pdc)$. This is the definition of a neutral effect which is positively- (Definition 1) and negatively-preserving (Definition 2).
(2) Let $n > 0$. We show this case for every combination of predicate pdc and neutral effect $ntr \in NTRLs(pdc)$.
(2a) Let $pdc = contains(l, c_1)$ respectively $\neg contains(l, c_1)$ and $ntr = add(l, c_2)$. Without limitations we only prove the case for $pdc = contains(l, c_1)$. The proof remains the same for $\neg contains(l, c_1)$. To prove:

$$seq(cmdb) \models contains(l, c_1) \quad \leftrightarrow \quad seq(add(l, c_2)(cmdb)) \models contains(l, c_1) \quad \text{iff}_{\text{Def. 15}}$$
$$contains^{def}(l^{seq(cmdb)}, c_1^{def}) \quad \leftrightarrow \quad contains^{def}(l^{seq(add(l,c_2)(cmdb))}, c_1^{def})$$

Because $c_1^{def} \neq c_2^{def}$, the number of instances of c_1^{def} is the same in l^{cmdb} and $l^{add(l,c_2)(cmdb)}$. Consequently, $l^{seq(cmdb)}$ and $l^{seq(add(l,c_2)(cmdb))}$ comprise the same number of instances of c_1^{def} as well. Thus,

$$contains^{def}(l^{seq(cmdb)}, c_1^{def}) \equiv contains^{def}(l^{seq(add(l,c_2)(cmdb))}, c_1^{def}).$$

(2b) Let $pdc = contains(l, c_1)$ respectively $\neg contains(l, c_1)$ and $ntr = remove(l, c_2)$. Without limitations we only prove the case for $pdc = contains(l, c_1)$. The proof remains the same for $\neg contains(l, c_1)$. To prove:

$$seq(cmdb) \models contains(l, c_1) \quad \leftrightarrow \quad seq(remove(l, c_2)(cmdb)) \models contains(l, c_1) \quad \text{iff}_{\text{Def. 15}}$$
$$contains^{def}(l^{seq(cmdb)}, c_1^{def}) \quad \leftrightarrow \quad contains^{def}(l^{seq(remove(l,c_2)(cmdb))}, c_1^{def})$$

Because $c_1^{def} \neq c_2^{def}$, the number of instances of c_1^{def} is the same in l^{cmdb} and $l^{remove(l,c_2)(cmdb)}$. Consequently, $l^{seq(cmdb)}$ and $l^{seq(remove(l,c_2)(cmdb))}$ comprise the same number of instances of c_1^{def} as well. Thus,

$$contains^{def}(l^{seq(cmdb)}, c_1^{def}) \equiv contains^{def}(l^{seq(remove(l,c_2)(cmdb))}, c_1^{def}).$$

□

CHAPTER 4

Algorithms for IT Change Verification

This chapter introduces efficient algorithms for several verification problems in the context of IT Change Management. The algorithms are based on the theory of extended partial-order reduction previously introduced in Chapter 2 and make use of the specification of IT change activities and safety constraints in change verification logic (see Chapter 3).
First, we introduce an efficient algorithm and its complexity to compute always-satisfiability of predicates in Section 4.1. Second, efficient algorithms solving various change verification problems are described in Section 4.2. [1]

4.1 An Efficient Algorithm to Compute Always-satisfiability of Predicates

In this section we introduce the functions necessary to determine always-satisfiability for predicates in the context of the change verification, the change conflict detection, and the mixed change verification and conflict detection problems (see Section 3.2.2). Furthermore, we discuss the complexity to determine always-satisfiability of predicates for all three problems.

The remainder of this section is structured as follows. First, we explain the basic steps necessary to determine always-satisfiability of predicates for the different verification problems in Section 4.1.1. In Section 4.1.2 we introduce the functions necessary to implement these steps and discuss their runtime complexity. In Section 4.1.3 we continue with a discussion of characteristics of change activities and safety constraints such that linear-runtime complexity can be achieved for the previously discussed steps and functions.

4.1.1 Steps to Determine Always-satisfiability

This section introduces the basic steps necessary to determine always-satisfiability of predicates in the context of the IT change verification problem, the IT change conflict detection problem,

[1] Parts of this chapter previously appeared in [51].

Functions to be called to decide always-satisfiability	Different verification problems: IT change verification	IT conflict detection	Verification and conflict detect.
RecordPredicates(*SCs*)	✓	✗	✓
RecordPredicates(*ACTs*)	✗	✓	✓
RecordEffects(*ACTs*)	✓	✓	✓
DetermineAlwSatPred(*SCs*)	✓	✗	✓
DetermineAlwSatPred(*ACTs*)	✗	✓	✓

Table 4.1: Functions that need to be called (from top to bottom) to compute always-satisfiability of all predicates in formulas of safety constraints and change activities depending on the verification problem to be solved (see Definition 18).

and the mixed change verification and conflict detection problem. Table 4.1 depicts the steps, i.e., function calls, that need to be executed to determine always-satisfiability of predicates depending on the verification problem. To determine always-satisfiability of predicates, three steps are necessary:

1. **Record predicates for which always-satisfiability needs to be decided (Function RecordPredicates):**
   Depending on the verification problem to solve (see Definition 18) the in formulas of change activities and/or safety constraints need to be recorded in an index structure. Our prototypical implementation uses the Configuration Items as index structures to quickly access the predicates and effects that target the same Configuration Item. For the IT change verification problem it suffices to only record the predicates of safety constraints in the Configuration Items they are evaluated on because always-satisfiability only needs to be decided for formulas of safety constraints. Similarly, for the IT conflict detection problem only predicates of change activities need to be recorded because safety constraints are not relevant for this verification problem. Consequently, predicates of safety constraints and change activities need to be recorded for the mixed change verification and conflict detection problem (see Table 4.1).

2. **Record effects of change activities (Function RecordEffects):**
   The second step records all effects of change activities in the Configuration Items they modify and their relationships (threat, support, neutral) in respect to the previously recorded predicates in the same Configuration Item are determined. This step needs to be performed for every verification problem.

3. **Compute always-satisfiability for every predicate in every formula that needs to be verified (Function DetermineAlwSatPred):**
   The third step finally computes always-satisfiability for all predicates necessary to solve the verification problem. To solve the IT change verification problem, always-satisfiability has to be computed for every predicate occurring in a safety constraint. Similarly, for the IT change conflict detection problem always-satisfiability has to be computed for predicates of change activities. For the combined problem always-satisfiability needs to be computed for every predicate of a formula of a safety constraint and change activity.

Once the three steps have been executed we know for each predicate in a formula that needs to be verified whether it is always-satisfiable.

4.1.2 Functions and Complexities to Compute Always-satisfiability of Predicates

This section introduces the functions that match to the steps to compute always-satisfiability of predicates and discusses their runtime complexity. The functions introduced herein are the basis of the verification algorithms presented in Section 4.2.

Variable	Description
$\#pred(ACTs, SCs)$	Max. number of predicates of a change activity or safety constraint formula in $ACTs$ or SCs.
$\#pred(ACTs)$	Max. number of predicates of a formula of a change activity.
$\#pred(SCs)$	Max. number of predicates of a formula of a safety constraint.
$\#eff(ACTs)$	Max. number of effects of a change activity.
$\#CI(PREDs)$	Max. number of CIs a predicate is evaluated on, $\#CI(PREDs) \leq_{\text{Section 4.1.3}} 2$.
$\#thrts(EFFs, PREDs)$	Max. number of threats to a predicate.
$\#CI(THRTs)$	Max. number of CIs changed by a threat, $\#CI(THRTs) =_{\text{Section 4.1.3}} 1$.
$\#CI(EFFs)$	Max. number of CIs influenced by an effect, $\#CI(EFFs) =_{\text{Section 4.1.3}} 1$.
$\#pred(CIs)$	Max. number of predicates evaluated on a Configuration Item.

Table 4.2: Characteristics of change activities, safety constraints, and Configuration Items that influence the runtime complexity of the functions used to compute always-satisfiability of predicates.

Function	Complexity
RecordPredicates(SCs)	$O(\,|SCs| \cdot \#pred(SCs) \cdot \#CI(PREDs)\,)$ $\in O(\,|SCs| \cdot \#pred(SCs)\,)$
RecordPredicates($ACTs$)	$O(\,|ACTs| \cdot \#pred(ACTs) \cdot \#CI(PREDs)\,)$ $\in O(\,|ACTs| \cdot \#pred(ACTs)\,)$
RecordEffects($ACTs$)	$O(\,|ACTs| \cdot \#eff(ACTs) \cdot \#CI(EFFs) \cdot \#pred(CIs)\,)$ $\in O(\,|ACTs| \cdot \#eff(ACTs) \cdot \#pred(CIs)\,)$
DetermineAlwSatPred(SCs)	$O(\,|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs) \cdot \#CI(THRTs)\,)$ $\in O(\,|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs)\,)$
DetermineAlwSatPred($ACTs$)	$O(\,|ACTs| \cdot \#pred(ACTs) \cdot \#thrts(EFFs, PREDs) \cdot \#CI(THRTs)\,)$ $\in O(\,|ACTs| \cdot \#pred(ACTs) \cdot \#thrts(EFFs, PREDs)\,)$

Table 4.3: Complexity of function calls necessary to decide always-satisfiability of predicates depending on the parameters passed. See Table 4.2 for description of variables and Table 4.1 for which functions need to be called depending on the verification problem to be solved.

Recording Predicates

Function RecordPredicates(X): Records every predicate of a first-order formula of a change activity or safety constraint in X in the Configuration Items they are evaluated on.

```
input : $X$ a set of change activities or safety constraints
1 for for every $\phi$ of a safety constraint or change activity $\in X$ do      ▹$|X|$
2     for for every predicate $pdc \in \phi$ do                                  ▹$\#pred(ACTs, SCs)$
3         for every Configuration Item ci pdc is evaluated on do                 ▹$\#CI(PREDs)$
4             record $pdc$ in $ci$ ;                                             ▹$O(1)$
5         end
6     end
7 end
```

Function RecordPredicates depicts the function to record predicates of safety constraints or change activities in the Configuration Items they are evaluated on. Depending on the verification problem (see Table 4.1) this function needs to be called with all change activities and/or all safety constraints. For every formula ϕ of a change activity or safety constraint in X (Function RecordPredicates, Lines 1-7) the algorithm iterates over every predicate *pdc* in the formula (Lines 2-6) and records the predicate (Line 4) in every Configuration Item the predicate is evaluated on (Lines 3-5).

Let $\#pred(ACTs, SCs)$ be the maximum number of predicates comprised in a safety constraint or a change activity formula and $\#CI(PREDs)$ the maximum number of Configuration Items a predicate is evaluated on. Then, the worst-case runtime complexity of Function RecordPredicates(X) is

$$O(\,|X| \cdot \#pred(ACTs, SCs) \cdot \#CI(PREDs)\,).$$

Called with $X = ACTs$, the set of all change activities, we obtain with $\#pred(ACTs, SCs) = \#pred(ACTs)$ (where $\#pred(ACTs)$ is the maximum number of predicates of a change activity) runtime complexity

$$O(\,|ACTs| \cdot \#pred(ACTs) \cdot \#CI(PREDs)\,).$$

Similarly, when RecordPredicates is called with $X = SCs$, the set of all safety constraints, we obtain with $\#pred(ACTs, SCs) = \#pred(SCs)$ (where $\#pred(SCs)$ is the maximum number of predicates of a safety constraint formula) runtime complexity

$$O(\,|SCs| \cdot \#pred(SCs) \cdot \#CI(PREDs)\,).$$

Recording and Categorizing Effects

In the second step to compute always-satisfiability of predicates, the effects of change activities are recorded in the Configuration Items they change and are categorized as supports, threats, or neutral effects in respect to the previously recorded predicates of the same Configuration Item. Function RecordEffects depicts the corresponding function. The function always needs to be called independently of the verification problem to be solved (see Table 4.1).

Function RecordEffects(*ACTs*): Records the effects of change activities $\in$ *ACTs* in the Configuration Items they affect and categorizes their relationship in respect to the previously recorded predicates. This function relies on Function RecordPredicates to be run prior to accurately determine relationships.

```
input  : $ACTs$ a set of change activities
for $act = (\phi_{act}, E_{act}) \in ACTs$ do                            ▹$|ACTs|$
    for $e \in E_{act}$ do                                                ▹$\#eff(ACTs)$
        apply $e$ to $CMDB$ ;                                             ▹$O(1)$
        for every Configuration Item $ci$ changed by $e$ do               ▹$\#CI(EFFs)$
            record $e$ in $ci$ ;                                          ▹$O(1)$
            for every predicate $pdc$ recorded in $ci$ do                 ▹$\#pred(CIs)$
                categorize relationship ;                                 ▹$O(1)$
            end
            restore $ci$ ;                                                ▹$O(1)$
        end
    end
end
```

For every change activity in *ACTs* (Function RecordEffects, Lines 1-12) the algorithm iterates over each effect of a change activity (Lines 2-11) to process it as follows. The effect is applied to the CMDB (Line 3) and for every Configuration Item that has been modified by its application to the CMDB (Lines 4-10) we (1) record the effect in the Configuration Item (Line 5) and (2) for every predicate (Lines 6-8) that has been recorded in the Configuration Item (by the previous application of Function RecordPredicates), the algorithm determines the relationship of the effect (threat, support, or neutral) to it and (3) finally restores the Configuration Item to its previous configuration (Line 9).

Let $\#eff(ACTs)$ be the maximum number of effects of a change activity, $\#CI(EFFs)$ the maximum number of Configuration Items influenced by an effect, and $\#pred(CIs)$ the maximum number of predicates evaluated on a Configuration Item. Then, the worst-case runtime complexity of Function RecordEffects when called with change activities *ACTs* is:

$$O(\,|ACTs| \cdot \#eff(ACTs) \cdot \#CI(EFFs) \cdot \#pred(CIs)\,)$$

Computing Always-satisfiability of Predicates

Function DetermineAlwSatPred computes always-satisfiability for all predicates of change activity or safety constraint formulas. For every change activity or safety constraint formula (Function DetermineAlwSatPred, Lines 1-13) and every predicate of that formula (Lines 2-12), the following needs to be done: For every threatening effect of the predicate (Lines 3-5), the threat is applied to the CMDB. Afterwards, the predicate is evaluated (Line 6). Notice that Lines 3-6 implement Theorem 8. However, the CMDB has to be reverted to its original configuration after the threat sequence was applied to the CMDB. For this purpose every Configuration Item (Lines 8-10) modified by a threat (Lines 7-11) needs to be restored (Line 9) to its original configuration.

Function DetermineAlwSatPred(X): Computes always-satisfiability according to Theorem 8 of every predicate contained in a formula of a safety constraint or change activity in X. Requires Functions RecordPredicates and RecordEffects to be run prior.

```
input : $X$ a set of change activities or safety constraints
1  for for every formula $\phi$ of a change activity or safety constraint in $X$ do      ▷$|X|$
2      for for every predicate $pdc \in \phi$ do                                        ▷$\#pred(ACTs, SCs)$
3          for threat $thr \in THRTs(pdc)$ do                                           ▷$\#thrts(EFFs, PREDs)$
4              apply $thr$ to CMDB;                                                     ▷$O(1)$
5          end
6          evaluate $pdc$ and record its value ;                                        ▷$O(1)$
7          for threat $thr \in THRTs(pdc)$ do                                           ▷$\#thrts(EFFs, PREDs)$
8              for $ci$ modified by $thr$ do                                            ▷$\#CI(THRTs)$
9                  restore $ci$ ;                                                       ▷$O(1)$
10             end
11         end
12     end
13 end
```

Let $\#thrts(EFFs, PREDs)$ be the maximum number of threats of a predicate, $\#CI(THRTs)$ the maximum number of Configuration Items changed by a threat and X the set of change activity and safety constraint formulas for which to compute always-satisfiability. Then, the runtime complexity of Function DetermineAlwSatPred is

$$O(\,|X| \cdot \#pred(ACTs, SCs) \cdot \#thrts(EFFs, PREDs) \cdot \#CI(THRTs)\,).$$

The complexity can be further refined depending on the parameters the function is called with:

- **IT change verification problem:** In this case Function DetermineAlwSatPred only needs to be called once with $X = SCs$ (see Table 4.1), yielding complexity

  $$O(\,|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs) \cdot \#CI(THRTs)\,)$$

  where $\#pred(SCs)$ is the maximum number of predicates of a safety constraint.

- **IT change conflict detection problem:** In this case Function DetermineAlwSatPred needs to be called once with $X = ACTs$ (see Table 4.1), yielding complexity

  $$O(\,|ACTs| \cdot \#pred(ACTs) \cdot \#thrts(EFFs, PREDs) \cdot \#CI(THRTs)\,).$$

- **IT change verification and conflict detection problem:** In this case Function DetermineAlwSatPred is called twice with $X = ACTs$ and $X = SCs$ (see Table 4.1), yielding total runtime complexity

  $$O(\,\#thrts(EFFs, PREDs) \cdot \#CI(THRTs) \cdot (|SCs| \cdot \#pred(SCs) + |ACTs| \cdot \#pred(ACTs))\,).$$

4.1.3 Requirements for Linear Runtime Complexity

In the previous section we showed that the runtime of the functions used to determine always-satisfiability of predicates is potentially polynomial in several variables (see Tables 4.2 and 4.3). The runtime complexity of the functions becomes linear in the number of change activities or safety constraints if the variables are restricted by an upper bound during verification experiments that scale parameters such as the size of the CMDB or the number of change activities / safety constraints. This section discusses the necessary characteristics of change activities and safety constraints such that the variables have an upper bound and whether these characteristics hold in practice for the change verification logic and IT change activities:

- $\#pred(ACTs, SCs)$, $\#pred(ACTs)$, $\#pred(SCs)$: These variables have an upper bound if formulas of change activities or safety constraints are comprised of a maximum number of predicates. As quantifiers are not allowed within Σ-formulas (see Definition 11), every Σ-formula is evaluated on a static number of predicates. This does not restrict the applicability of the approach as formulas of change activities or safety constraints always address a limited number of Configuration Items (e.g., a database, a router, a part of the network, etc.) and not the whole data center.

- $\#eff(ACTs)$: This variable has an upper bound if the number of effects of a change activity has an upper bound. The specification of change activities only allows sets of effects with a finite number of effects (see Definition 16). This does not affect the applicability of the approach as change activities only reconfigure a few properties of Configuration Items and not the whole data center.

- $\#CI(PREDs)$, $\#thrts(EFFs, PREDs)$: The first variable has an upper bound if a predicate is evaluated on a maximum number of Configuration Items. This holds for the predicates of the change verification logic (see Table 3.3) because $\#CI(PREDs) \leq 2$. The second variable, the number of threatening effects of a predicate, has an upper bound as well if the effects of change activities are roughly equally distributed over Configuration Items. If a workload comprises change activities that always focus on the same Configuration Items and threaten the same predicates, then scaling out this workload linearly increases $\#thrts(EFFs, PREDs)$. In such a scenario the linear runtime complexity to determine always-satisfiability is lost.

- $\#CI(EFFs)$: This variable always has an upper bound because every effect of the change verification logic changes at most one property of a Configuration Item (see Table 3.3). Thus, $\#CI(EFFs) = 1$.

- $\#pred(CIs)$: This variable has an upper bound if the maximum number of predicates evaluated on a Configuration Item has an upper bound. This is the case if formulas of change activities and safety constraints are roughly equally distributed over the Configuration Items of the CMDB. For example, a linear increase in the number of change activities or safety constraints whose formulas are always evaluated on the same Configuration Items, destroys the linear runtime property.

- $\#CI(THRTs)$: An upper bound of $\#CI(EFFs)$ is also an upper bound of $\#CI(THRTs)$. Thus, $\#CI(THRTs) = 1$.

Consequently, the linear runtime complexity of the functions necessary to determine always-satisfiability of predicates is lost in cases for which

- the number of change activities or safety constraints is scaled out, but the predicates of their formulas are not equally distributed over the CMDB ($\#pred(CIs)$ is not restricted).
- the number of change activities is scaled out, but their threats to predicates are not equally distributed over the CMDB ($\#thrts(EFFs, PREDs)$ is not restricted).

Thus, change activities and safety constraints need to be roughly equally distributed over the Configuration Items of the CMDB if they are scaled out during a verification experiment. We can expect this requirement to hold because frequent changes to the same Configuration Item would already indicate a deeper routed problem with that Configuration Item.

4.2 Algorithms for Change Verification Problems

This section introduces the algorithms to solve the IT change verification problem, the IT change conflict detection problem, and the combined verification problem. Furthermore, the runtime complexity of each algorithm is discussed.

4.2.1 IT Change Verification Problem

Function ChangeVerification($SCs, ACTs$): returns *true* if all safety constraints in SCs hold under any workload of change activities in $ACTs$, otherwise *false* is returned.

```
input : $SCs$ a set of safety constraints and $ACTs$ a set of change activities
1 RecordPredicates($SCs$) ;                          ▹$O(|SCs| \cdot \#pred(SCs))$
2 RecordEffects($ACTs$) ;                            ▹$O(|ACTs| \cdot \#eff(ACTs) \cdot \#pred(CIs))$
3 DetermineAlwSatPred($SCs$) ;                       ▹$O(|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs))$
4 for every formula $\phi_{sc}$ of a safety constraint $sc \in SCs$ do   ▹$|SCs|$
5     if $\phi_{sc}$ is not always-satisfiable then                     ▹$O(\#pred(SCs))$
6         return false;
7     end
8 end
9 return true;
```

The IT change verification problem (see Section 3.2.2) addresses the question whether a set of safety constraints holds under any possible execution sequence of change activity effects.

First, always-satisfiability needs to be determined for all predicates comprised in formulas of safety constraints. Thus, the functions previously introduced in Section 4.1.2 need to be called in the order as depicted in Table 4.1. These functions are called in the first three lines of the algorithm (see Function ChangeVerification). After that, always-satisfiability can be determined for all formulas of safety constraints in SCs using Theorem 9. For that purpose the algorithm iterates over every formula ϕ_{sc} of a safety constraint $sc \in SCs$ (Lines 4-8). As soon as a formula ϕ_{sc} is not always-satisfiable (efficiently determined using Theorem 9), *false* is returned (Line 6), otherwise *true* (Line 9). Evaluating a formula has runtime complexity linear in the number of predicates comprised in it.

Adding up the runtime complexities of each step yields the total complexity:

$$\begin{aligned}&\text{ChangeVerification}(SCs, ACTs) \in\\ &\in O(\,|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs) + |ACTs| \cdot \#eff(ACTs) \cdot \#pred(CIs)\,)\\ &\in_{\text{Section 4.1.3}} O(\,|SCs| + |ACTs|\,)\end{aligned} \tag{4.1}$$

4.2.2 IT Change Conflict Detection Problem

Function ConflictDetection(*ACTs*): returns *true* if all change activities in *ACTs* are feasible, otherwise *false* (see Definition 18).

```
input : $ACTs$ a set of change activities
1 RecordPredicates($ACTs$) ;                 ▷$O(|ACTs| \cdot \#pred(ACTs))$
2 RecordEffects($ACTs$) ;                    ▷$O(|ACTs| \cdot \#eff(ACTs) \cdot \#pred(CIs))$
3 DetermineAlwSatPred($ACTs$) ;              ▷$O(|ACTs| \cdot \#pred(ACTs) \cdot \#thrts(EFFs, PREDs))$
4 for every formula $\phi_{act}$ of act $\in ACTs$ do      ▷$|ACTs|$
5     if $\phi_{act}$ evaluates to false then              ▷$\#pred(ACTs)$
6         return false;
7     end
8 end
9 return true;
```

The IT change conflict detection problem decides whether the execution of a set of change activities can render the precondition of another change activity infeasible. Thus, speaking in the terms of extended partial-order reduction, always-satisfiability needs to be decided for all formulas of change activities using Theorem 9.

To achieve this, the functions previously introduced in Section 4.1.2 need to be called in the order as depicted in Table 4.1 because always-satisfiability of all required predicates needs to be determined first. The functions are called in the first three lines of the algorithm (see Function ConflictDetection). After that, always-satisfiability can be determined for all formulas ϕ_{act}, which describe preconditions of change activities. For that purpose the algorithm iterates over every formula of a change activity (Lines 4-8). As soon as a formula is not always-satisfiable, *false* is returned (Line 6), otherwise *true*. Evaluating a formula using Theorem 9 has runtime complexity linear in the number of the predicates comprised in it.

Adding up the runtime complexities of each step yields the total complexity:

$$\begin{aligned}&\text{ConflictDetection}(ACTs) \in\\ &\in O(\,|ACTs| \cdot (\#eff(ACTs) \cdot \#pred(CIs) + \#pred(ACTs) \cdot \#thrts(EFFs, PREDs))\,)\\ &\in_{\text{Section 4.1.3}} O(\,|ACTs|\,)\end{aligned} \tag{4.2}$$

4.2.3 IT Change Verification and Conflict Detection Problem

The combined IT change verification and conflict detection problem requires the change verification and change conflict detection problem to be solved at the same time. Thus, given a set of change activities and safety constraints determine whether any execution sequence of effects of change activities invalidates a safety constraint or the precondition of a change activity.

Function ConflictDetectionAndChangeVerification($SCs, ACTs$): returns *true* if no arbitrary execution sequence of change activities in *ACTs* turns a safety constraint $sc \in SCs$ or a change activity $act \in ACTs$ infeasible, otherwise *false* is returned (see Definition 18).

```
input : $ACTs$ a set of change activities and $SCs$ a set of safety constraints
1  RecordPredicates($SCs$) ;                  $\triangleright O(|SCs| \cdot \#pred(SCs))$
2  RecordPredicates($ACTs$) ;                 $\triangleright O(|ACTs| \cdot \#pred(ACTs))$
3  RecordEffects($ACTs$) ;                    $\triangleright O(|ACTs| \cdot \#eff(ACTs) \cdot \#pred(CIs))$
4  DetermineAlwSatPred($SCs$) ;               $\triangleright O(|SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs))$
5  DetermineAlwSatPred($ACTs$) ;              $\triangleright O(|ACTs| \cdot \#pred(ACTs) \cdot \#thrts(EFFs, PREDs))$
6  $F = SCs \cup ACTs$;
7  for every formula $\phi$ of a safety constraint or change activity $\in F$ do    $\triangleright |SCs| + |ACTs|$
8      if $\phi$ is not always-satisfiable then          $\triangleright O(\#pred(ACTs))$ or $O(\#pred(SCs))$
9          return false;
10     end
11 end
12 return true;
```

First, the functions previously introduced in Section 4.1.2 need to be called in the order as depicted in Table 4.1 (see first three lines of Function ConflictDetectionAndChangeVerification). After that, always-satisfiability can be determined for every compound formula of a safety constraint or change activity using Theorem 9. For that purpose the algorithm iterates over each formula in $SCs \cup ACTs$ (Lines 6-11). If a formula is not always-satisfiable, *false* is returned (Line 9), otherwise *true* (Line 12). Evaluating a formula using Theorem 9 has runtime complexity linear in the number of predicates comprised in it.

Adding up the runtime complexities of each step yields the following complexity to solve the combined verification problem:

$$\begin{aligned}
&\text{ConflictDetectionAndChangeVerification}(SCs, ACTs) \in \\
&\in O(()\,|ACTs| \cdot (\#eff(ACTs) \cdot \#pred(CIs) + \#pred(ACTs) \cdot \#thrts(EFFs, PREDs)) + \\
&\quad + |SCs| \cdot \#pred(SCs) \cdot \#thrts(EFFs, PREDs)) \\
&\in_{\text{Section 4.1.3}} O(|ACTs| + |SCs|)
\end{aligned} \tag{4.3}$$

4.3 Conclusions

In this chapter we introduced algorithms based on Theorem 9 to solve several verification problems in the context of IT Change Management. A complexity analysis showed that the runtime performance of the algorithms is linear in the number of safety constraints and change activities, assuming that the formulas of safety constraints and change activities (more precisely the predicates in the formulas) are equally distributed over the Configuration Items and that the effects of change activities are equally distributed over Configuration Items (more precisely the threats of change activities in respect to the predicates).

The algorithms presented herein can also be implemented without the presence/use of Configuration Items as index structures to speedup the categorization of an effect in respect to previously recorded effects. In that case, index structures are necessary to efficiently look up the effects and predicates evaluated on constants of the Σ-interpretations. For example, sufficiently sized hash maps that allow for each constant to efficiently lookup the predicates evaluated on it.

CHAPTER 5

IT Change Verification in Practice

This chapter presents the evaluation of the extended partial-order reduction model checker in the context of an outage that recently happened at one of Amazon's data centers. Section 5.1 introduces the Amazon outage that is used as an evaluation case study in the remainder of this chapter. In Section 5.2 we introduce two model checkers, NuSMV and SPiN, which are used for evaluation against our special purpose model checker. After that, Section 5.3 describes the experimental setting in more detail. The results of our experimental evaluation are presented in Section 5.4. Finally, Section 5.5 discusses related work and Section 5.6 concludes the chapter.[1]

5.1 Amazon Data Center Outage

This section describes the data center outage that occurred in April 2011 at one of Amazon's data centers and gives a first overview of the change activities[2], safety constraints, and models that can be used to detect the outage using verification. First, Section 5.1.1 explains how a particular network change caused the outage. After that, we discuss in Section 5.1.2 the change activities that could have caused the outage in a static routing scenario, together with their logical description and models that can be used to detect the violation of safety constraints. Finally, scenarios, change activities, safety constraints, and models for the outage in the context of a dynamic routing scenario are introduced in Section 5.1.3.

5.1.1 How a Simple Change Activity Brought Down a Cloud Service

In this section we explain the background of Amazon's data center outage. First, we give a short introduction to the architecture of Amazon's Elastic Block Store (EBS) service that was

[1]Parts of this chapter previously appeared in [51].

[2]Whenever we refer to the notion of a change activity in this chapter, we mean atomic change activities, i.e., not decomposable, elementary change activities. A synonym for atomic change activities is change procedures. The notion of abstract and atomic change activities is of no relevance in this chapter as change verification only needs to address atomic change activities/change procedures that are currently pending for execution. See Section 1.4

involved in the outage. After that, we provide more details of the outage.

Amazon Elastic Block Store Service

This section provides a short introduction to the architecture of Amazon's Elastic Block Store (EBS) service to aid in the understanding of the data center outage caused by an IT change activity. A more detailed description of the EBS service can be found in the report released by Amazon [1]. The EBS service is a distributed storage service that stores virtual disc images that can be mounted to virtual machines (EC2 instances). The EBS service comprises (1) EBS clusters (comprising EBS nodes) that store the images and (2) control services that provide access to the EBS service.

An EBS cluster comprises several EBS nodes that store the EBS volumes in a replicated fashion. An EBS volume is automatically replicated across several EBS nodes for durability and availability purposes. Replication is handled in a rather aggressive way: Once a copy of an image becomes out of sync, e.g., because connection is lost, the system assumes the data to be lost and the volume is automatically replicated to another node.

The nodes of an EBS cluster can communicate with each other using two different networks: (1) A primary, high-capacity network that mainly handles the replication traffic among EBS nodes and communication with control services. (2) A secondary, low-capacity network that provides a low-capacity but highly reliable connection among the EBS nodes of a cluster. It is important to notice that the low-capacity network is not designed to accommodate the traffic of the high-capacity network as its focus lies on reliable communication and not on maximum throughput rates.

Outage Caused by Faulty IT Change Activity

On April 21st 2011 a network change had to be performed at one of Amazon's data centers. The abstract change activity was to upgrade the capacity of the primary, i.e., high-capacity network. Amazon's incident report [1] leaves the exact nature of the change open. However, to implement the upgrade, a change activity had to be executed to shift traffic off from the affected high-capacity router to a redundant router of the high-capacity network. A typical change that would cause such a scenario would be the upgrade of the memory of a router. Instead of routing the traffic to the redundant router of the high-capacity network, the traffic was routed onto the low-capacity network, i.e., the secondary network. As the secondary network could not handle the traffic from the high-capacity network due to limited capacity, the secondary network was overloaded. Consequently, there were nodes in a cluster (the ones located in different subnets of the cluster) that could neither communicate with each other using the primary nor the secondary network because traffic destined for the first was now routed to the second, which was overloaded.

Due to the traffic shift, many EBS nodes in the cluster lost connection between each other. The change manager quickly noted the overload and rolled back the network change activity to restore the connectivity. Once connectivity was restored, the nodes assumed that the replicas of images were lost and rapidly began to search the cluster for available space to replicate the images again. Because the network outage affected so many EBS nodes (and thus volumes), not all re-mirroring requests could be served because the remaining capacity of the cluster was quickly exhausted by the re-mirroring requests. As a consequence, the nodes got stuck in a loop of re-mirroring requests and requests made by customers to the control service could not be serviced anymore and started to fail. This caused failed customer requests. In the end the cluster could only be restored after several days by installing addition capacity. A small percentage of customers even suffered durable data loss. Besides design issues and software bugs that exacerbated the situation, it was a simple change activity to reroute traffic that caused a chain of events leading to an outage. The safety constraint that was invalidated by the change activity is rather simple: Never route high-capacity traffic over the low-capacity network - a constraint that was known to the network management group because the network was specifically designed this way. The constraint could have been easily identified, captured, and formalized using proper risk analysis [96, 97] for Change Management.

5.1.2 Network Overload in Static Routing Environments

This section provides an overview of the change activities, safety constraints and models that can be used to verify changes of the Amazon case study in a static routing scenario. First, we give an overview of the static network setup in Amazon's case. Second, we map the configuration to an object-oriented CMDB model. Third, we discuss the different scenarios that could have caused Amazon's network outage in a static routing environment. Finally, we introduce the different change activities and safety constraints (together with their logical specification in change verification logic) for the scenarios that could have caused Amazon's network outage in a static routing environment.

Overview of Network Setup

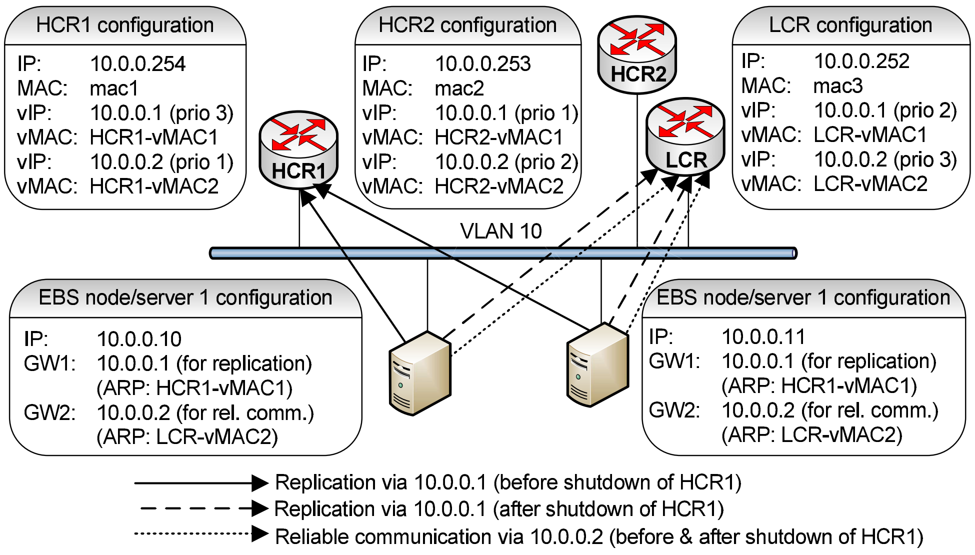

HCR1 configuration
IP: 10.0.0.254
MAC: mac1
vIP: 10.0.0.1 (prio 3)
vMAC: HCR1-vMAC1
vIP: 10.0.0.2 (prio 1)
vMAC: HCR1-vMAC2
HCR2 configuration
IP: 10.0.0.253
MAC: mac2
vIP: 10.0.0.1 (prio 1)
vMAC: HCR2-vMAC1
vIP: 10.0.0.2 (prio 2)
vMAC: HCR2-vMAC2
LCR configuration
IP: 10.0.0.252
MAC: mac3
vIP: 10.0.0.1 (prio 2)
vMAC: LCR-vMAC1
vIP: 10.0.0.2 (prio 3)
vMAC: LCR-vMAC2
HCR1
HCR2
LCR
VLAN 10
EBS node/server 1 configuration
IP: 10.0.0.10
GW1: 10.0.0.1 (for replication) (ARP: HCR1-vMAC1)
GW2: 10.0.0.2 (for rel. comm.) (ARP: LCR-vMAC2)
EBS node/server 1 configuration
IP: 10.0.0.11
GW1: 10.0.0.1 (for replication) (ARP: HCR1-vMAC1)
GW2: 10.0.0.2 (for rel. comm.) (ARP: LCR-vMAC2)
Replication via 10.0.0.1 (before shutdown of HCR1)
Replication via 10.0.0.1 (after shutdown of HCR1)
Reliable communication via 10.0.0.2 (before & after shutdown of HCR1)

Figure 5.1: Servers of an EBS cluster configured with static routing and a first hop gateway redundancy protocol. Policy routing is in place separating disc replication traffic from reliable EBS node communication.

In static routing environments servers of the EBS cluster are configured with a static gateway IP address of a router interface. This is a common configuration found in data centers [8] because it avoids the uncertainty emerging from the usage of an Interior Gateway Protocol (IGP). In order to still achieve routing redundancy with static routes, the gateway IP address of a server points to a virtual IP address that is shared among the interfaces of a group of routers that run a first hop redundancy protocol. Notice that the gateway IP address is only active on at most one of the interfaces, the interface of the primary router. Once a router becomes unavailable, the router with the second highest priority for the virtual interface takes over the virtual interface and routing is shifted to the backup router/interface.

Typical first hop redundancy protocols are, for example, Cisco's proprietary Hot Standby Router Protocol[1] (HSRP), the Gateway Load Balancing Protocol[2] (GLBP), or the IETF standard Virtual Router Redundancy Protocol[3] (VRRP). For example, consider Figure 5.1, which depicts an example configuration among two EBS servers, two high-capacity routers (HCR1, HCR2), and a low-capacity router (LCR). Without limitations we assume that each router has one physical interface (see IP address entry for each router) that resides on the same network as the EBS nodes. Each physical router interface hosts two virtual interfaces with the IP addresses 10.0.0.1 and 10.0.0.2. The two virtual interfaces are configured for each router, but at the same time each type of virtual interface is active on at most one router. For example, the virtual interface with vIP 10.0.0.1 (the gateway towards which high-capacity traffic is routed by the EBS nodes) is configured on all three routers using a first hop redundancy protocol, but it is currently only active on HCR1 because the virtual interface with vIP 10.0.0.1 has the highest priority on HCR1 (priority 3) compared to the priorities on HCR2 (priority 1) and LCR (priority 2) (see Figure 5.1). Should HCR1 become unavailable (and with it the virtual interface with vIP 10.0.0.1 on it), the virtual interface with the same vIP located on router LCR becomes active and takes over the traffic sent to vIP 10.0.0.1 because LCR has the second-highest priority for 10.0.0.1. Similar, the virtual interface with vIP 10.0.0.2, which is currently hosted by LCR (LCR has the highest priority among all routers for 10.0.0.2), is taken over by HCR2 because its virtual interface with vIP 10.0.0.2 has the second-highest priority (priority 2) on the network (see Figure 5.1).

Object-oriented Model of the CMDB

The static routing network configuration (see Figure 5.1) is modeled using an object-oriented instance diagram (see Figure 5.2) where instances of classes match to Configuration Items. We later introduce in Section 5.3.2 different models to describe the static routing environment and change activities. To provide a first overview, we focus on Model$_1$, the most detailed model of all infrastructure models (see Figure 5.2).

Figure 5.2 depicts an instance, i.e., configuration of the CMDB, in the most detailed model used to verify change activities in the static routing scenario. In Figure 5.2 a single EBS server resides on a network with two high-capacity routers (hcr1 and hcr2) and one low-capacity router (lcr). The left side of Figure 5.2 depicts the Configuration Items used to describe the configuration of an EBS server (Mark and RoutingTable Configuration Items) and the right side depicts the network Configuration Items the server is attached to (VRRPInterface and Router Configuration Items). Each server has two routing tables (rt1 and rt2) that have different default gateways, i.e., interfaces of routers the routing tables route to by default. The configuration of the first hop redundancy protocol has been simplified by using a failover reference among VRRPInterface Configuration Items, which describes the interface that takes over once an interface becomes unavailable. Mark Configuration Items are associated to the RoutingTable Configuration Items. They describe the traffic by port that is routed with the associated routing table. For example, the mark1 Configuration Item describes that traffic destined for port a (high-capacity

[1]http://www.cisco.com/en/US/tech/tk648/tk362/technologies_tech_note09186a0080094a91.shtml, retrieved
[2]http://www.cisco.com/en/US/docs/ios/12_2t/12_2t15/feature/guide/ft_glbp.html
[3]http://tools.ietf.org/html/rfc3768

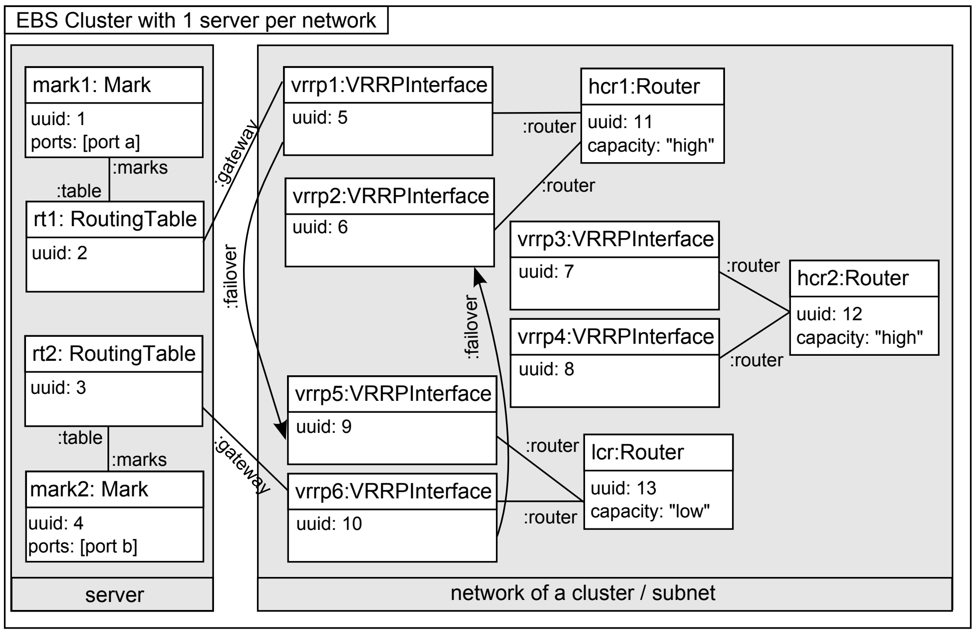


Figure 5.2: Object-oriented model of the CMDB describing the routing configuration in Figure 5.1 for one EBS server.

traffic) is routed using routing table rt1 (see reference between Mark and RoutingTable Configuration Item in Figure 5.2) and thus is forwarded towards virtual interface vrrp1 (see gateway reference), which is a high-capacity router interface (see reference between vrrp1 and hcr1 and property capacity of hcr1). Such a configuration to separate high- from low-capacity traffic based on destination ports and different default gateways can be achieved using the Linux commands `ip tables`, `ip rule`, and `ip route`. Refer to [20] for a detailed description on how to configure this setup in practice.

Network Overload Scenarios in Static Routing Environment

Two scenarios, differing on whether the change either targets the network or the configuration of the EBS servers, could have caused Amazon's network overload in a statically configured network with a first hop redundancy protocol as follows.

Scenario 1: When the primary (active) router is taken off the network to be upgraded, its network interfaces become unavailable. Once an interface is not reachable anymore, the failover interface takes over the network traffic by taking over vIP and vMAC of the deactivated interface. For example, in Figure 5.1 the servers are configured with policy routing such that high-capacity traffic (for replication) is routed via vIP 10.0.0.1 and reliable traffic via 10.0.0.2. The first high-capacity router (HCR1 in Figure 5.1) is the active primary for the replication traffic because it has the highest priority among all other routers for vIP 10.0.0.1. Similarly, the low-capacity router (LCR) has the highest priority for vIP 10.0.0.2. Thus, HCR1 is currently the active router for unreliable traffic and LCR the active one for reliable traffic.
For Scenario 1 we assume that the priorities among the interfaces of high- and low-capacity routers have been wrongly configured, such that taking down the current high-capacity router causes a failover to a low-capacity router interface. Figure 5.1 depicts such a configuration where LCR holds the second highest priority for the high-capacity vIP (10.0.0.1). Thus deacti-

vating HCR1 would cause LCR to take over vIP 10.0.0.1 and would shift high-capacity traffic to the low-capacity network. In Figure 5.2 this configuration is described in a more abstract way using a failover reference between vrrp1 and vrrp5, a low-capacity router interface.

Scenario 2: To achieve the traffic shift, the routing policy in the EBS nodes could have been accidentally changed such that high-capacity traffic is explicitly routed towards the low-capacity router. In Figure 5.2 the routing policy is described by a *gateway* association between a RoutingTable and a VRRPInterface Configuration Item. This association describes the default gateway configured for a RoutingTable Configuration Item. Mark Configuration Items, which are associated to RoutingTable Configuration Items, describe the type of traffic that is routed using the referenced routing table. For example, in Figure 5.2 high-capacity traffic (port a) is categorized as mark1 traffic that is routed using routing table rt1, which has a high-capacity router interface (vrrp1) as its default gateway.

Safety Constraints and Change Activities

Safety constraints are formulated over the object-oriented CMDB in Figure 5.2 using the predicates and effects of the change verification logic (see Table 3.3) in order to describe the previously introduced scenarios.

```
SC1(router:Router, interface:VRRPInterface, rt:RoutingTable, mark:Mark)
pred$_1$:  | router.capacity.hasValue("low")
pred$_2$:  | interface.router.hasValue(router)
pred$_3$:  | rt.gateway.hasValue(interface)
pred$_4$:  | rt.marks.contains(mark)
pred$_5$:  | mark.ports.!contains(port a)
pre:       | $(\wedge_{i=1...4}$ pred$_i) \rightarrow$ pred$_5 \equiv (\vee_{i=1...4} \neg$pred$_i) \vee$ pred$_5$
```

SC1 is the safety constraint that protects the network modeled in Figure 5.2 from a network overload. SC1 guarantees that high-capacity traffic (outgoing on port a) (*pred*$_5$) is only allowed to be routed via routing tables that route to VRRP interfaces (*pred*$_3$) that belong to (*pred*$_2$) a high-capacity router (*pred*$_1$). Safety constraints are directly evaluated on the configuration of a CMDB to determine whether they are violated. For example, the instance SC1(lcr,vrrp6,rt2,mark2) of safety constraint SC1 evaluates to true in the configuration depicted in Figure 5.2. The interested reader is encouraged to instantiate SC1 with the parameters, i.e., Configuration Items, depicted in Figure 5.2 to comprehend the evaluation of the precondition.

For **Scenario 1** we consider two different types of interface failover change activities:

- **Failover negative change activity (FOn)**: A *FOn* change activity occurs when an interface of a high-capacity router is switched off and the configured failover interface belongs to a low-capacity router. This causes a network overload because all nodes of the affected subnet automatically start routing towards the low-capacity interface causing its overload.
- **Failover positive change activity (FOp)**: A *FOp* change activity occurs when an interface of a low-capacity router is switched off and the failover is an interface of a high-capacity router. In this case traffic that was originally routed over the low-capacity router

is now routed over a high-capacity router. Different to a *FOn* change activity, it does not cause a network overload because we assume that the high-capacity network can accommodate the traffic of the low-capacity network.

Notice that the change activities introduced in this section are based on two assumptions. First, the high-capacity network can always accommodate the accumulated traffic of both networks. Second, the low-capacity network cannot handle the accumulated traffic of its own traffic and the high-capacity traffic of at least one EBS server. As we detail later in this Section, this assumption is not a requirement for our approach to describe the scenario because the change verification logic provides the logical expressiveness to describe scenarios for which only a portion of low- or high-capacity traffic can be accommodated on the opposite network.

Notice that *FOn* and *FOp* change activities can occur concurrently. For example, in Figure 5.2 high-capacity traffic via vrrp1 can be shifted to vrrp5 using a *FOn* change activity and vrrp2 takes over vrrp6 using a *FOp* change activity. The following template can be used to describe both change activities in change verification logic over the model depicted in Figure Figure 5.2:

```
FO(rt:RoutingTable, current:VRRPInterface, failover:VRRPInterface)
pred₁:  | rt.gateway.hasValue(current)
pred₂:  | current.failover.hasValue(failover)
pre:    | pred₁ ∧ pred₂
eff₁:   | rt.gateway.setValue(failover)
```

A failover change activity takes three parameters:

- `vrrp`, the VRRPInterface that is meant to be deactivated.
- `failover`, the VRRPInterface that is currently configured as failover interface of `vrrp`.
- The routing table `rt` that routes to the VRRPInterface meant to be deactivated (`vrrp`).

Depending on the parameters of the `FO` change template above, *FOp* or *FOn* change activities can be instantiated from the template. For example, `FO(vrrp1,vrrp5,rt1)` (see Figure 5.2 for the Configuration Items) yields an applicable *FOn* change activity and the choice `FO(vrrp6,vrrp2,rt2)` yields an applicable *FOp* change activity.

To model **Scenario 2**, we introduce two change activities that manipulate the routing policy of an EBS node by changing the mark that is carried by traffic that is destined for a particular port. This associates the traffic with another routing table that has a different default gateway. We distinguish two different instances of change activities:

- **Shift traffic negative change activity (SHTn)**: A *SHTn* change activity changes the routing policy of an EBS node in such a way that high-capacity traffic is assigned the mark of low-capacity traffic. Consequently, high-capacity traffic is routed using the routing policy of low-capacity traffic, i.e., towards a low-capacity router interface. This causes an overload of the low-capacity router.
- **Shift traffic positive change activity (SHTp)**: A *SHTp* change activity changes the routing policy of an EBS node in such a way that low-capacity traffic is assigned the mark of high-capacity traffic. Consequently, low-capacity traffic is routed using high-capacity

rules, i.e., towards a high-capacity router interface. Different to a *SHTn* change activity, this does not cause a network overload as we assume in our case study that the high-capacity network can handle the accumulated traffic of both networks.

Notice that *SHTp* and *SHTn* change activities can occur concurrently. For example, in Figure 5.2 a *SHTn* change activity is executable to shift traffic destined for port a from mark1 to mark2 and a *SHTp* change activity is applicable to shift traffic destined for port b from mark2 to mark1. To describe SHT change activities, the following description in verification logic can be used:

```
SHT(from: Mark, to: Mark, port: int)
pred1:  |  from.ports.contains(port)
pred2:  |  to.ports.!contains(port)
pre:    |  pred1 ∧ pred2
eff1:   |  from.ports.remove(port)
eff2:   |  to.ports.add(port)
```

A SHT change activity is applicable if its parameter `port` is not comprised in the list of ports that is routed via the `from` Mark (*pred*$_1$) and if the property `ports` of the destination Mark `to` (*pred*$_2$) does not yet contain the port. In this case the object-oriented model is changed by two effects: *eff*$_1$ removes the port from property `ports` of the source Mark Configuration Item (`from`) and *eff*$_2$ adds it to property `ports` of the destination Mark Configuration Item (`to`). `SHT` can be instantiated to either obtain *SHTp* or *SHTn* change activities. For example, `SHT(mark1,mark2,port a)` yields a *SHTn* change activity because high-capacity traffic (`port a`) is shifted to the low-capacity network and `SHT(mark2,mark1,port b)` yields a *SHTp* change activity. See Figure 5.2 for the instances of the Configuration Items.

The models introduced herein assume that already rerouting the high-capacity traffic of one EBS server over the low-capacity network causes a network overload. Finer grained scenarios that require a specific threshold of servers to be reached to cause a network overload are possible as well. The change verification logic is expressive enough to describe such a scenario. For example, integer counters can be used to describe the number of EBS servers routing via a specific interface/network. Routing changes then manipulate these counters and safety constraints check whether the counters adhere to thresholds. A possible safety constraint would then, for example, be that no more than *n* EBS servers are allowed to route high-capacity traffic via the low-capacity network. Change activities would simply have to be adapted with effects manipulating the counter and safety constraints with predicates that check the threshold constraint.

5.1.3 Network Overload in Dynamic Routing Environment

Overview of Network Setup

Instead of static routing, a data center network can also be configured with dynamic routing. In such a configuration routes are installed and withdrawn from routing tables of the EBS nodes by a routing protocol depending on current network conditions. When an EBS node learns of several routes via different gateways to the same network, it installs the cheapest route according to a metric into its routing table. Once the cheapest route is no longer available, the newest cheapest route is installed into the routing table and a traffic shift occurs.

In this work we focus on Open Shortest Path First[1] (OSPF), which is arguably the most widely used routing protocol for routing within an autonomous system, e.g., a large data center. It is important to notice that the verification logic is expressive enough to describe any routing protocol because it supports arithmetic operations (increase and decrease) and arithmetic comparisons frequently used by routing protocols to determine the cheapest route.

There are two ways how to configure the costs of routes that are sent within advertisements from network interfaces to the EBS servers:

- **Manual costs:** The cost / metric of an interface can be manually configured[2].
- **OSPF costs:** The cost / metric of an interface is automatically computed. For Cisco and Juniper routers the OSPF cost of an interface ($cost_{intf}$) is the quotient of the reference bandwidth bw_{ref} and the bandwidth of the interface bw_{intf}. Thus, $cost_{intf}$ can only be manipulated by changing bw_{ref} because the bandwidth of the interface is fixed.

Object-oriented Model of the CMDB

The object-oriented CMDB model to detect a network overload in the dynamic routing scenario can be less detailed than the one used for static configuration. For dynamic routing changes it suffices to only model network interfaces with properties for their current cost / metric, the reference bandwidth, and the link bandwidth because the dynamic routing change activities only affect VRRPInterface Configuration Items.

We later discuss in Section 5.3.2 in more detail the different CMDB models to describe and verify dynamic routing change workloads.

Scenarios of Network Overload in Dynamic Routing Environment

In a dynamic routing scenario the cheapest route towards a network is installed into the routing table. In the case of the Amazon outage, a network overload occurs when the cost of a route advertised by an interface of a low-capacity router is cheaper than the cost advertised by every interface of the high-capacity routers. In this case, the EBS nodes install the route via the low-capacity interface into their routing tables leading to a network overload. Generally the following dynamic routing scenarios could have caused the network overload in Amazon's case:

- **Scenario 1:** In this scenario the cost/metric of a low-capacity router interface is accidentally reduced such that it becomes the newest cheapest interface to offer a route.
- **Scenario 2:** In this scenario the costs/metrics of the high-capacity router interfaces are increased such that the low-capacity router interface offers the newest cheapest route.

Both scenarios cause a network shift towards a low-capacity router interface and consequently a network overload. Notice that the semantics of the cost/metric of an interface does not matter for the scenario.

[1]OSPF Version 2 has been standardized in RFC 2328.

[2]For example, by using the command `ip ospf cost` on Cisco routers.

Safety Constraints and Change Activities

In this section we introduce the change activities and safety constraints for dynamic routing environments together with their specification in change verification logic. To model the different scenarios, the following change activities to describe increases and decreases to the cost of an interface are used:

- **Decrease manually configured cost of low-capacity router interface (LCRDecrM):** An *LCRDecrM* change activity decreases the manually configured cost of a router interface by a specified delta. *LCRDecrM* change activities are used to implement Scenario 1. The logical description of an *LCRDecrM* change activity in verification logic is straightforward:

```
LCRDecrM(interface:VRRPInterface, int:delta)
eff1:   | interface.cost.dec(delta)
```

- **Increase manually configured cost of high-capacity router interface (HCRIncrM):** A *HCRIncrM* change activity increases the manually configured cost of a router interface by a specified delta. Two *HCRIncrM* change activities are used to implement Scenario 2. The logical description of a *HCRIncrM* change activity in verification logic is straightforward:

```
HCRIncrM(interface:VRRPInterface, int:delta)
eff1:   | interface.cost.inc(delta)
```

If the network is configured with automatically computed OSPF metrics, interface costs cannot be directly manipulated. Instead, common OSPF implementations, e.g., the implementations of Cisco or Juniper, automatically calculate the cost of an interface ($cost_{intf}$) as the quotient of the reference bandwidth (bw_{ref}) and the bandwidth of an interface (bw_{intf}). While the latter one is fixed, the reference bandwidth can be changed to influence the costs of an interface. From

$$cost_{intf} = bw_{ref}/bw_{intf}$$

we conclude that an increase of bw_{ref} by $\Delta d > 0$ increases $cost_{intf}$ by $\Delta d/bw_{intf}$. This analogously holds for decreases as well. Consequently, we introduce the following two change activities to change the metric of an OSPF interface by increasing / decreasing its reference bandwidth:

- **Decrease reference bandwidth of low-capacity router interface (LCRDecrOSPF):** An *LCRDecrOSPF* change activity decreases the reference bandwidth of a low-capacity router interface by a specified delta. This causes a decrease of the OSPF metric of that interface as OSPF metrics are calculated based on the reference bandwidth. An *LCRDecrOSPF* change activity is used to implement Scenario 1. The logical description of the *LCRDecrOSPF* change activity contains an additional effect compared to *LCRDecrM* to update the reference bandwidth.

```
LCRDecrOSPF(interface:VRRPInterface, int:delta)
eff1:   | interface.refbw.dec(delta)
eff2:   | interface.cost.dec(delta / interface.intfbw)
```

- **Increase reference bandwidth of high-capacity router interface (HCRIncrOSPF):** A *HCRIncrOSPF* change activity increases the reference bandwidth of a high-capacity router interface by a specified delta. This causes an increase of the OSPF metric of that interface as OSPF metrics are calculated based on the reference bandwidth. Two *HCRIncrOSPF* change activities are used to implement Scenario 2. The logical description of the *HCRIncrOSPF* change activity contains an additional effect compared to *HCRIncrM* to update the reference bandwidth:

```
HCRIncrOSPF(interface:VRRPInterface, int:delta)
eff1:  | interface.refbw.inc(delta)
eff2:  | interface.cost.inc(delta / interface.intfbw)
```

Independent of whether dynamic routing is modeled with manual or OSPF costs, the following constraint has to hold to prevent a network overload:

```
SC5/SC5'(vrrp1:VRRPInterface, vrrp2:VRRPInterface, vrrp3:VRRPInterface,
         vrrp4:VRRPInterface, vrrp:VRRPInterface)
pred1:    | vrrp1.cost < vrrp.cost
pred2:    | vrrp2.cost < vrrp.cost
pred3:    | vrrp3.cost < vrrp.cost
pred4:    | vrrp4.cost < vrrp.cost
pre SC5 : | ∨i=1...4 predi (for workloads not comprising LCRDecrM
                      or LCRDecrOSPF change activities to interface vrrp)
pre SC5': | ∧i=1...4 predi (for workloads comprising LCRDecrM
                      or LCRDecrOSPF change activities to interface vrrp)
```

SC5/SC5' depicts the constraint to protect a network comprising four high-capacity router interfaces (parameters vrrp1 through vrrp4) and a low-capacity router interface (vrrp) from a network overload caused by dynamic routing changes. Logically the satisfaction of SC5 guarantees that the cost of at least one high-capacity router interface is lower than the cost of low-capacity router interface vrrp. If SC5 is not satisfied, the low-capacity router interface has the cheapest metric and a network overload occurs as the EBS nodes route towards the low-capacity network. Notice that if we allow the occurrence of *LCRDecrM* or *LCRDecrOSPF* change activities, the Boolean or ($\vee$) in the precondition of SC5 has to be changed to a logical and ($\wedge$) because decrements to the cost of the low-capacity router interface (property vrrp.cost) violate Lemma 10 (decrements to vrrp.cost are threats to every predicate $\texttt{pred}_i$ of the precondition formula $\vee_{i=1...4}\ \texttt{pred}_i$) and thus turns Theorem 9 inapplicable. Thus, if *LCRDecrM* or *LCRDecrOSPF* change activities appear in the workload, we need to fall back to a more restrictive version of the safety constraint (SC5') that requires all high-capacity router interfaces to offer a cheaper route than the low-capacity router interface.

5.2 Model Checkers and Optimization Strategies

This section introduces the NuSMV [26] model checker, the SPiN [15, 54] model checker, and our own special purpose extended partial-order reduction model checker for IT change verification. The SPiN and NuSMV model checkers have been chosen for evaluation against our extended partial-order reduction model checker because both cover the most popular approaches for model checking. Among these are symbolic algorithms (binary decision diagram model checking, NuSMV), SAT-based model checking (NuSMV), and explicit-state model checking with partial-order reduction (SPiN).

5.2.1 NuSMV Model Checker

NuSMV [26] is a symbolic model checker making use of binary decision diagrams (BDDs). It originates from CMU SMV, the original BDD-based model checker developed at CMU. NuSMV supports several different ways to perform model checking: First, NuSMV supports two different temporal logics for the specification of safety constraints, Computation Tree Logic (CTL) and/or Linear Temporal Logic (LTL) [11]. Second, problems can either be model checked using NuSMV's internal binary decision diagram model checker (called BDD-based model checking herein) [27] or by using an external SAT solver [28] (called SAT-based model checking herein). Based on the choices we benchmark different optimization techniques for IT change verification using the NuSMV model checker.

- **BDD-based CTL model checking (BDD CTL):**
  Using BDD CTL model checking, binary decision diagrams are generated from the description of state transition systems that describe the current configuration and the effects of pending change activities. Safety constraints are specified in CTL and verified with NuSMV's internal symbolic model checker.

- **BDD-based LTL model checking (BDD LTL):**
  BDD LTL model checking works like BDD CTL model checking, but safety constraints are formalized in LTL.

- **SAT-based LTL model checking (SAT LTL):**
  Using the SAT LTL model checking optimization technique, the model checking problem is translated into a satisfiability (SAT) problem and an external SAT solver (Minisat) is used to verify the safety constraint in LTL [28].

5.2.2 SPiN Model Checker

SPiN [15, 54] is a model checker developed by Gerard J. Holzmann originally intended for the verification of communication protocols. SPiN was originally developed at Bell Labs in the Unix group of the Computing Sciences Research Center, starting in 1980. Since 1991 SPiN has been available as open-source software. Since then it continues to be adapted to keep pace with developments in the field of formal verification. In April 2002, SPiN was awarded the ACM Software Systems Award. Since then SPiN has been widely used in industries that have a need to verify critical systems and has become one of the most well known model checkers.

This section describes different optimization techniques that the SPiN model checker can be configured with. These techniques will later be used when it comes to benchmark SPiN against our own model checker and the NuSMV model checker. Different to NuSMV, SPiN is an explicit-state model checker that explicitly searches and checks the search space induced by the execution of change activities over the CMDB configuration. The different configurations of the CMDB (also called states herein) are created during the search and stored in a hash table for the purpose of detecting cycles. The performance of the SPiN model checker heavily depends on two factors [15]:

- Whether collisions occur in the hash table. This is the case if the hash table is not large enough compared to the search space to be explored. To prevent performance penalties caused by direct chaining, the strategy used by SPiN to cope with collisions, we use a hash map sufficiently large enough to avoid a significant performance impact caused by collisions.

- The size a state occupies in memory and the speed states can be generated and evaluated for conditions. SPiN offers a variety of different state representation techniques to optimize this aspect.

In the following we explain the different optimization techniques offered by the SPiN model checker when it comes to efficiently store and process states during verification. These optimization techniques will later be used to evaluate the performance of SPiN against our special purpose model checker and NuSMV.

- **No compression (NCP):**
  Using the no compression technique (NCP), states are stored uncompressed. The size of a state, i.e., the configuration of a data center, equals the sum of the sizes of all properties of all Configuration Items of a configuration. For example, consider a Configuration Item to hold two properties, an integer property (4 byte) and a byte property (1 byte). Then, the configuration of one Configuration Item consumes 5 bytes. The configuration of a CMDB comprising 1 million of these Configuration Items then consumes roughly 5MB.

- **Collapse compression (CP):**
  In real application scenarios state vectors of several hundred bytes are not uncommon - especially when verifying IT change workloads over large CMDBs. The SPiN model checker implements a solution to compress states called collapse compression (CP). Collapse compression comes at a trade-off between runtime performance and memory consumption.

- **Minimal automaton (DMA):**
  Instead of storing states using a hash map, state vectors can also be stored using a minimal automaton. According to Ben-Ari [15] this representation is similar to the binary decision diagrams used in other model checkers, e.g., NuSMV. Using a minimal automaton, the memory requirements can usually be reduced to a very small amount at the cost of additional runtime.

- **Partial-Order Reduction:**
  Partial-order reduction (see Section 2.2) is a technique to reduce the state-space that needs to be searched by a model checker. Partial-order reduction model checking exploits the commutativity criterion of actions, i.e., effects of change activities. If actions lead to the same state independent of the order they are applied and intermediate states do not matter for the verification problem, then intermediate states can be neglected for verification. Partial-order reduction can drastically reduce the size of the search space and can be used independently of the optimization technique to store states. Partial-order reduction is activated by default in all benchmarks.

A more detailed introduction to the SPiN model checker is provided in [15] or [54].

5.2.3 Special Purpose Model Checker

We have implemented a special purpose model checker (also called extended partial-order reduction model checker herein), which is based on the theory of extended partial-order reduction (Chapter 2) and the algorithms previously presented in Chapter 4. The special purpose model checker has been implemented in the Java programming language to achieve complete compatibility with object-oriented models, which are frequently used to describe IT configurations [21, 60]. It is important to note that the effects of change activities, i.e., the effects of the change verification logic (see Chapter 3), and the formulas of safety constraints, i.e., the predicates of the change verification logic, are implemented as executable Java code. For example, every change activity has a method *applyEffects()* that implements the effects of the change activity on Java objects that describe the Configuration Items of the CMDB. Similarly, the formulas of safety constraints are described using a Boolean method that is evaluated over the Java objects that describe the Configuration Items of the CMDB. Java programs are compiled in Java bytecode, which is interpreted by Java virtual machines. When enabled, Java's just-in-time compilation feature compiles frequently executed code into natively compiled machine code, which executes faster than the interpreted bytecode of the Java virtual machine. Thus, we distinguish two different optimization techniques for the extended partial-order reduction model checker:

- **Just-in-time compilation (JITC):**
  Using this optimization technique, the preconditions and effects of frequently appearing change activities are automatically compiled in machine code if decided to be necessary by the default setting of the Java virtual machine.

- **No just-in-time compilation (noJITC):**
  The preconditions and effects of change activities are always interpreted by the Java virtual machine.

The use of just-in-time compilation does not pose an unfair advantage because the SPiN model checker compiles the verification problem in native machine code as well. Notice that compilation does not change the actual complexity of an algorithm although it sometimes tends to disguise it a little bit as we will see later in Section 5.4.2.

5.3 Experimental Setup

This section describes the experimental setup of the evaluation. First, we discuss the influence of the inputs on the performance of a model checker. After that, different formalizations for models, safety constraints, and change activities are introduced.

5.3.1 Factors of Influence on the Verification Performance

This section discusses different characteristics of models, change activities, and safety constraints that influence the runtime performance of model checkers.

- **Complexity of the configuration model:** Depending on the model checker, the degree of detail of an infrastructure model can significantly influence the runtime performance. Complex, i.e., detailed, infrastructure models tend to consume more memory. Therefore, a trade-off needs to be found among the detail of a model and the goal to achieve verifiable problem instances. This problem is frequently increased as change mangers are not trained to write efficiently verifiable specifications.

- **Change workloads:** The workload of change activities to be verified significantly influences the runtime of a model checker: (1) The modeling detail of change activities (number of effects and number of predicates used to describe a change activity) has an impact on the runtime performance of model checkers. The more effects a change activity has, the more parts of a model are changed. This can result in a larger search space. The complexity of a change activities correlates with the complexity of the model. (2) A workload comprising different types of changes (opposed to only one type of change) can also increase the search space as more Configuration Items are changed by the effects of change activities and more detailed models are required.

- **Safety constraints:** A safety constraint to be verified can influence the runtime of a model checker as well: The modeling detail of a safety constraint (number of predicates used to describe a safety constraint) can have an impact on the runtime performance of a model checker. The more complex the safety constraint, the more change activities potentially influence the evaluation of the safety constraint. The complexity of a safety constraint correlates with the complexity of the model. Detailed models and change activities imply more complex safety constraints as more Configuration Items are changed.

Thus, for a fixed workload of change activities to be verified, there are different ways to model the IT infrastructure, safety constraints, and change activities. By considering variations in these parameters, we will assess the robustness of the different model checkers for IT change verification in terms of runtime performance. For this purpose we introduce in the subsequent section different models (with different specifications for change activities and safety constraints) to be used in the performance evaluation.

5.3.2 Models and Safety constraints

We introduce in this section different models and formalizations of change activities and safety constraints for the previously introduced static and dynamic routing change activities of the Amazon network outage. These models are used in a robustness and benchmark analysis to compare NuSMV and SPiN with the extended partial-order reduction model checker. The models differ in the following characteristics:

- The detail in which the IT infrastructure is modeled. For example, there are models that only comprise the minimal number of Configuration Items (each with minimal properties) necessary to verify a workload. These models are called minimal models because they describe the minimum information necessary to solve the verification problem. On the other hand, there are models that describe the infrastructure in more detail than actually necessary to perform the verification. The development of minimal models requires considerably more experience in modeling.

- The workload of change activities that can be verified using a model, i.e., the types of concurrent change activities supported by a model. For example, there are models that can verify any combination of static-routing changes (*SHTp*, *SHTn*, *FOp*, and *FOn*) or specialized models that can only verify a particular type of change (e.g., *FOp* or *FOn* changes).

	IT change activities supported by model				
Model	*SHTp*, *SHTn*	*FOp*, *FOn*	*HCRIncrM*, *LCRDecrM*	*HCRIncrOSPF*, *LCRDecrOSPF*	Minimal model?
Model_1	✓	✓			No
Model_2	✓				Yes
Model_3		✓			Yes
Model_4	✓	✓			Yes
Model_5			✓		Yes
Model_6				✓	Yes
Model_7			✓		No
Model_8				✓	No

Table 5.1: IT change activities supported by each model.

	CI Mark			CI RoutingTable			CI VRRPInterface						CI Router	
Model	*uuid*	*table*	*ports*	*uuid*	*gateway*	*marks*	*uuid*	*router*	*failover*	*reference bandwith*	*interface bandwith*	*cost*	*uuid*	*capacity*
Model_1	✓	✓	✓	✓	✓	✓	✓	✓	✓				✓	✓
Model_2			✓											
Model_3					✓		✓		✓					
Model_4			✓		✓		✓		✓					
Model_5							✓					✓		
Model_6							✓			✓	✓	✓		
Model_7	✓	✓	✓	✓		✓	✓	✓				✓	✓	✓
Model_8	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓

Table 5.2: Properties covered of Configuration Items comprised in the different models.

- The logical description of safety constraints and change activities. For example, depending on the model, the logical descriptions of change activities and safety constraints can be more or less complex.

Table 5.1 depicts the different models and the workload of change activities supported by each of them. For example, it can be observed that workloads comprising *SHTp* and *SHTn* change activities can be solved using Model$_1$, Model$_2$, and Model$_4$.

Table 5.2 describes the Configuration Items and their properties for each model (Model$_1$ - Model$_8$). For example, when comparing Model$_1$ with Model$_4$, it can be observed that Model$_4$ comprises less Configuration Items and less properties than Model$_1$ while both support the same change activities. In the following paragraphs we describe the different models and provide rationale for their inclusion in the robustness analysis.

Model 1: A Detailed Model for Static Routing Change Activities

Model$_1$ is the most detailed model to verify all types of static routing changes (*SHTp*, *SHTn*, *FOp*, and *FOn*). Compared to the other models also supporting static routing changes, Model$_1$ provides unnecessary modeling detail, but it is the model that most realistically depicts the whole configuration. Figures 5.3 and 5.4 depict instances of Model$_1$ that describe the configuration of a one-server EBS cluster and how instances of the static routing change activities alter the model. Alternatively, Table 5.2 describes the model in tabular fashion.

To protect a configuration described by Model$_1$ from a network overload caused by static routing change activities, safety constraints are necessary to detect unwanted configurations. The safety constraint used to protect a configuration described by Model$_1$ is the same as the one previously introduced in Section 5.1.2. For the purpose of completeness it is mentioned again:

```
SC1(router:Router, interface:VRRPInterface, rt:RoutingTable, mark:Mark)
pred1:  | router.capacity.hasValue("low")
pred2:  | interface.router.hasValue(router)
pred3:  | rt.gateway.hasValue(interface)
pred4:  | rt.marks.contains(mark)
pred5:  | mark.ports.!contains(port a)
pre:    | (∧i=1...4 predi) → pred5 ≡ (∨i=1...4 ¬predi) ∨ pred5
```

Please refer to Section 5.1.2 for the logical explanation of the constraint. Similar to SC1, the logical description of *SHTp* and *SHTn* remains the same as well. For the purpose of completeness we provide the specification again:

```
SHT(from:Mark, to:Mark, port:int)
pred1:  | from.ports.contains(port)
pred2:  | to.ports.!contains(port)
pre:    | pred1 ∧ pred2
eff1:   | from.ports.remove(port)
eff2:   | to.ports.add(port)
```

Depending on how the parameters of the SHT change activity are instantiated, *SHTp* or *SHTn* change activities are obtained. For example, Figure 5.3 depicts instances of SHT that result in

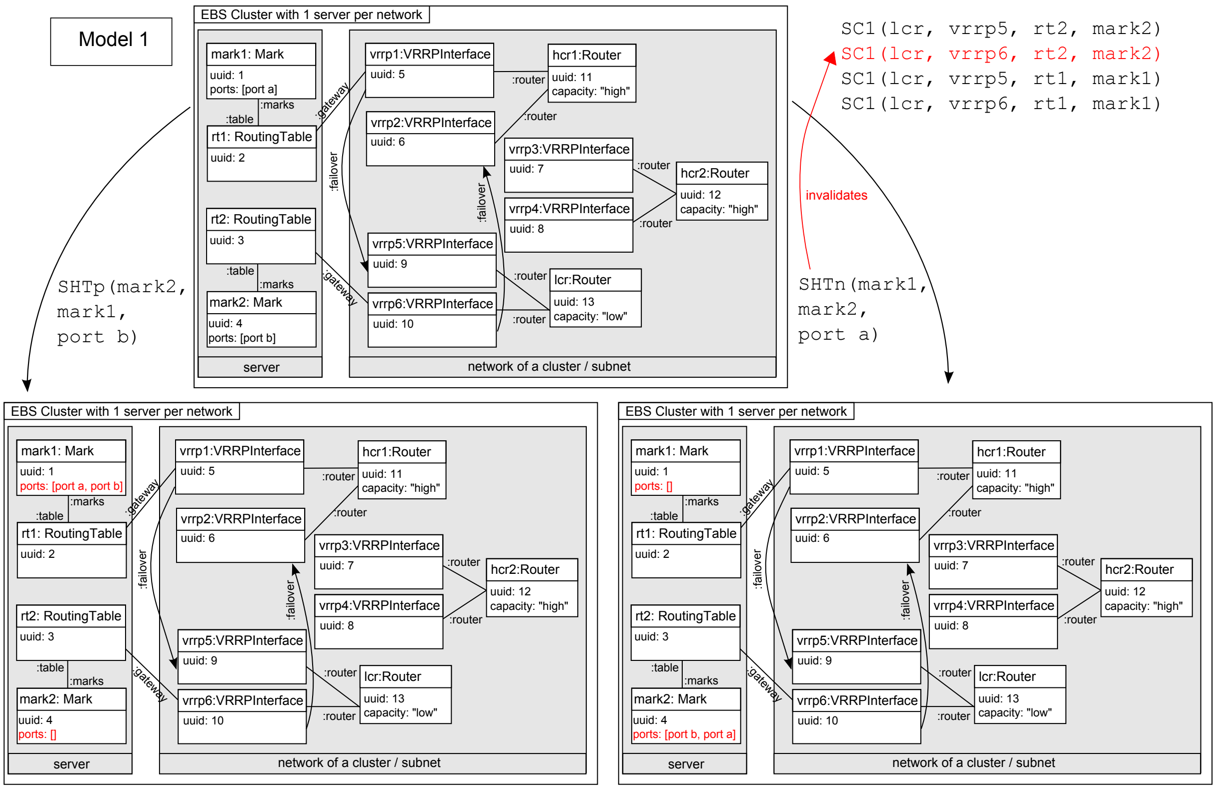


Figure 5.3: *SHTp* and *SHTn* change activities and safety constraints in a one-server EBS cluster in Model$_1$.

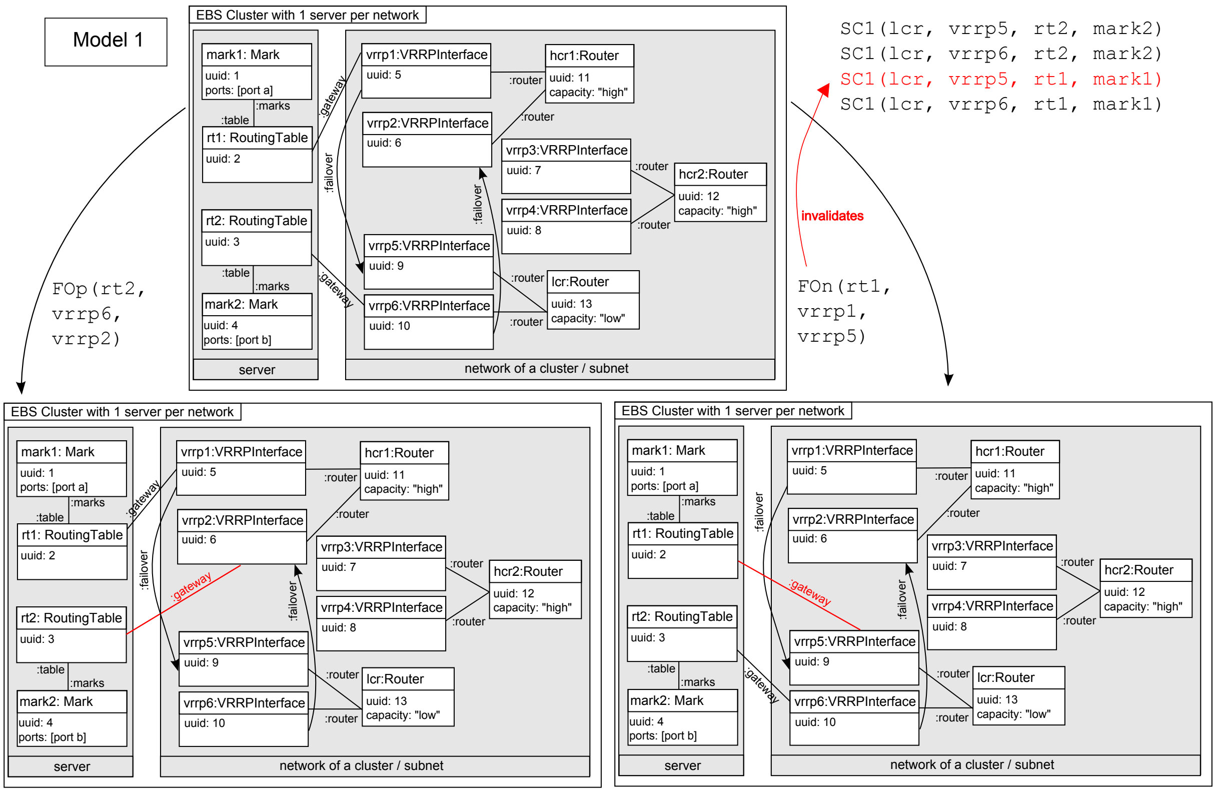

Model 1
EBS Cluster with 1 server per network
SC1(lcr, vrrp5, rt2, mark2)
SC1(lcr, vrrp6, rt2, mark2)
SC1(lcr, vrrp5, rt1, mark1)
SC1(lcr, vrrp6, rt1, mark1)
invalidates
FOp(rt2, vrrp6, vrrp2)
FOn(rt1, vrrp1, vrrp5)
:gateway
:failover
server
network of a cluster / subnet

Figure 5.4: *FOp* and *FOn* change activities and safety constraints in a one-server EBS cluster in Model$_1$.

SHTp and *SHTn* change activities. `SHT(mark1, mark2, port a)` in Figure 5.3 describes a *SHTn* change activity because high-capacity traffic (`port a`) is shifted from the `mark1` Configuration Item to the `mark2` Configuration Item. This change violates the second instance of safety constraint `SC1` in Figure 5.3 that protects the configuration from an overload.

In Model$_1$ *FOp* and *FOn* change activities are described with the same logical specification as previously introduced in Section 5.1.2:

```
FO(rt: RoutingTable, current: VRRPInterface, failover: VRRPInterface)
pred1:  | rt.gateway.hasValue(current)
pred2:  | current.failover.hasValue(failover)
pre:    | pred1 ∧ pred2
eff1:   | rt.gateway.setValue(failover)
```

Depending on how the parameters of the `FO` change activity are instantiated, we obtain *FOp* or *FOn* change activities. Figure 5.4 depicts the effects of *FOp* and *FOn* change activities on Model$_1$. For example, `FO(rt1, vrrp1, vrrp5)` describes a *FOn* change activity because the new gateway of `rt1` (which is the routing table used to route high-capacity traffic because it is associated with a Mark Configuration Item for high-capacity traffic) is set to `vrrp5` which is a low-capacity router interface. This change violates the third (out of four) instances of safety constraint `SC1` that protects the configuration from an overload. For interested readers we suggest to substitute the instances of change activities and safety constraints in Figures 5.3 and 5.4 by their logical specification to comprehend the graphical model changes and the invalidation of safety constraints.

Model 2: A Minimal Model for SHTp and SHTn Change Activities

Model$_2$ is a highly-optimized version of Model$_1$ that only supports change workloads comprising *SHTp* and *SHTn* change activities. Different to Model$_1$, this model cannot be used to verify *FOp* and *FOn* change activities. *SHTp* and *SHTn* change activities have the same logical specification as in Model$_1$, i.e., they only modify Mark Configuration Items. Thus, for workloads only comprising *SHTp* and *SHTn* change activities it suffices to use models that only comprise Mark Configuration Items. All other Configuration Items can be neglected as their configuration remains unchanged and the safety constraint can be simplified to reflect this. Thus, parts of the safety constraint of Model$_1$ become obsolete yielding `SC2`.

```
SC2(mark: Mark)
pred1:  | mark.port.!contains(port a)
pre:    | pred1
```

`SC2` is the safety constraint of Model$_2$. It is sound if it is instantiated for every Mark Configuration Item that is statically associated with a routing table that statically routes towards a low-capacity interface. These associations can be assumed to be static because no other change activities to change them are supported by Model$_2$. We call Model$_2$ the minimal model to verify a workload of *SHTp* and *SHTn* change activities because it only models Mark Configuration Items with a single property to solve the verification problem.

For interested readers we highlight that substituting the logical descriptions of change activities and safety constraints in Figure 5.5 invalidates the logical specification of the safety

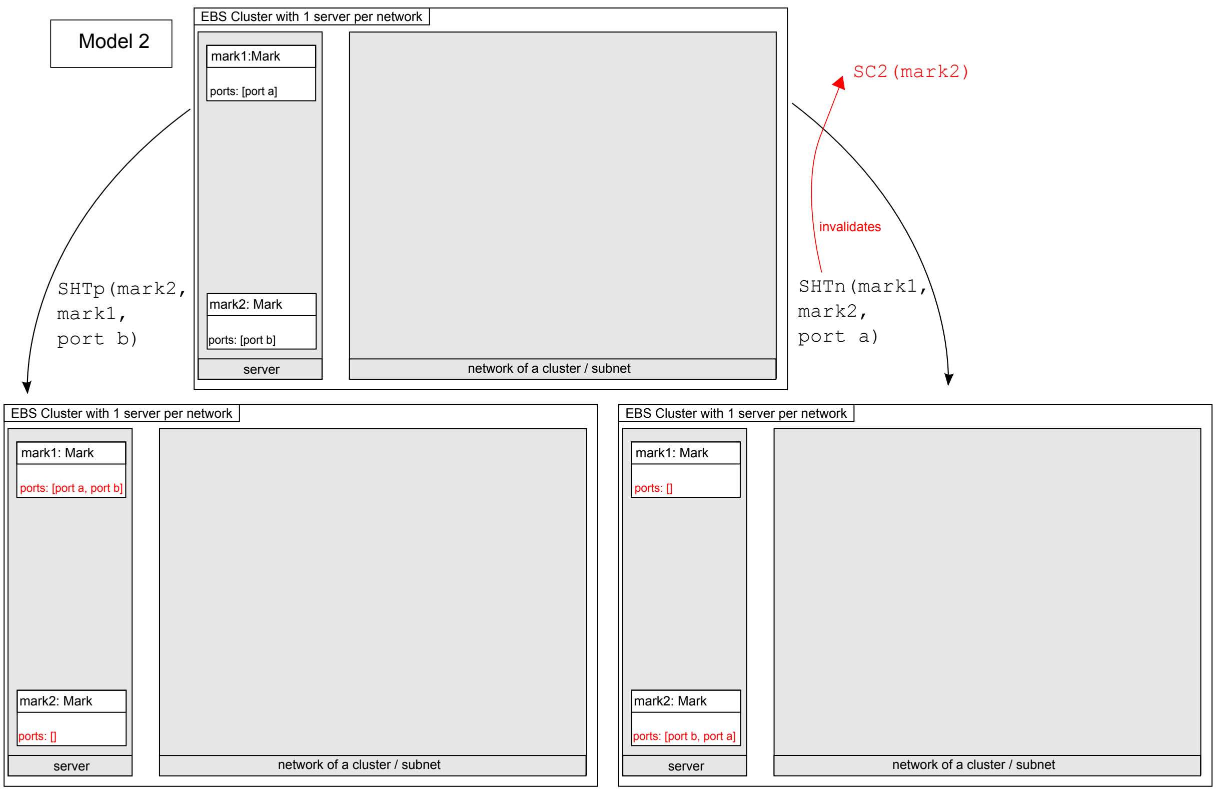

Model 2
EBS Cluster with 1 server per network
mark1:Mark
ports: [port a]
mark2: Mark
ports: [port b]
server
network of a cluster / subnet
SC2(mark2)
invalidates
SHTp(mark2, mark1, port b)
SHTn(mark1, mark2, port a)
EBS Cluster with 1 server per network
mark1: Mark
ports: [port a, port b]
mark2: Mark
ports: []
server
network of a cluster / subnet
EBS Cluster with 1 server per network
mark1: Mark
ports: []
mark2: Mark
ports: [port b, port a]
server
network of a cluster / subnet

Figure 5.5: *SHTp* and *SHTn* change activities and safety constraints in a one-server EBS cluster in Model_2.

constraint as depicted. The logical description of *SHTp* and *SHTn* change activities equals that of Model_1.

Model 3: A Minimal Model for FOp and FOn Change Activities

Model_3 (see Figure 5.6 for an instance in the one-server EBS cluster) is a highly-optimized version of Model_1 that only supports change workloads comprising *FOp* and *FOn* change activities. *FOp* and *FOn* change activities have the same logical specification as in Model_1, i.e., they change the gateway association between routing tables and interfaces to reflect the default gateway of a routing table. Consequently, for pure *FOp* & *FOn* workloads it suffices to only use models that comprise RoutingTable and VRRPInterface Configuration Items. All other Configuration Items can be neglected as their configuration remains unchanged. Thus, parts of the safety constraint of Model_1 become obsolete yielding `SC3`.

```
SC3(rt:RoutingTable, interface:VRRPInterface)
pred₁:  | rt.gateway.!hasValue(interface)
pre:    | pred₁
```

`SC3` is the safety constraint of Model_3. The safety constraint is sound if it is instantiated for every combination of routing table that routes high-capacity traffic (rt1 in our case) and low-capacity router interface (vrrp5 and vrrp6). Formalized in change verification logic, the safety constraint checks whether a routing table that routes high-capacity traffic (`rt`) has a low-capacity router interface as default gateway (`vrrp`). As this model does not allow/support other change activities, `SC3` is sound. Model_3 is the minimal model to verify a workload of *FOp* and *FOn*

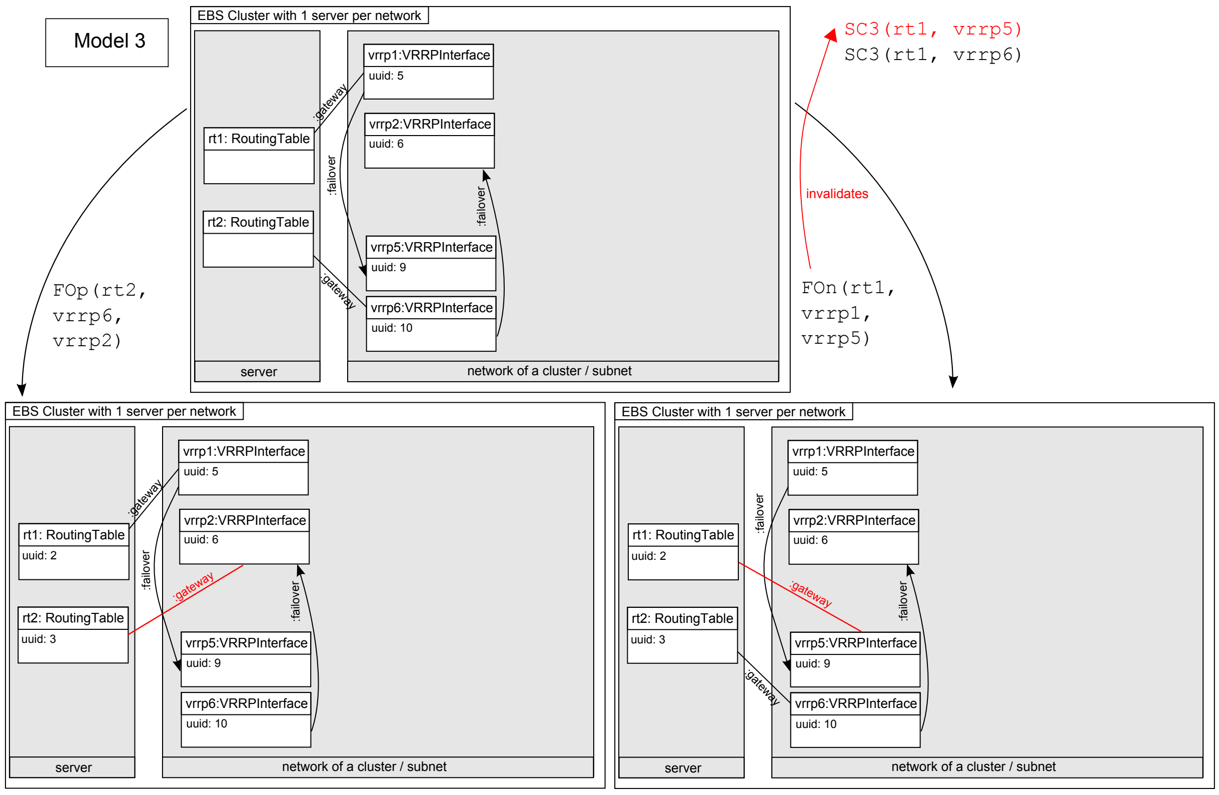


Figure 5.6: *FOp* and *FOn* change activities and safety constraints in a one-server EBS cluster in Model$_3$

change activities because it only comprises RoutingTable and VRRPInterface Configuration Items, which are the minimal Configuration Items affected by the logical description of the *FOp* and *FOn* change activities.

For interested readers we highlight that substituting the logical descriptions of change activities and safety constraints in Figure 5.6 invalidates the logical specification of the safety constraints as depicted. The logical specification of *FOp* and *FOn* change activities equals that of Model$_1$.

Model 4: A Minimal Model for Static Routing Change Activities

Model$_4$ (see Figures 5.7 and 5.8 for an instance in the one-server EBS cluster) is a highly-optimized version of Model$_1$ that supports any static routing change. In Model$_4$ all change activities have the same specification as in Model$_1$. However, a couple of Configuration Items can be neglected compared to Model$_1$ as they remain unchanged by static routing changes. Thus, parts of the safety constraint of Model$_1$ become obsolete yielding SC4.

```
SC4(mark: Mark, rt: RoutingTable, vrrp: VRRPInterface)
pred₁:  |  mark.ports.contains(port a)
pred₂:  |  rt.gateway.hasValue(vrrp)
pre:    |  pred₁ → ¬ pred₂ ≡ ¬pred₁ ∨ ¬pred₂
```

SC4 is the safety constraint of Model$_4$. The safety constraint is sound if it is instantiated for every combination of a Mark Configuration Item, its statically associated routing table, and any

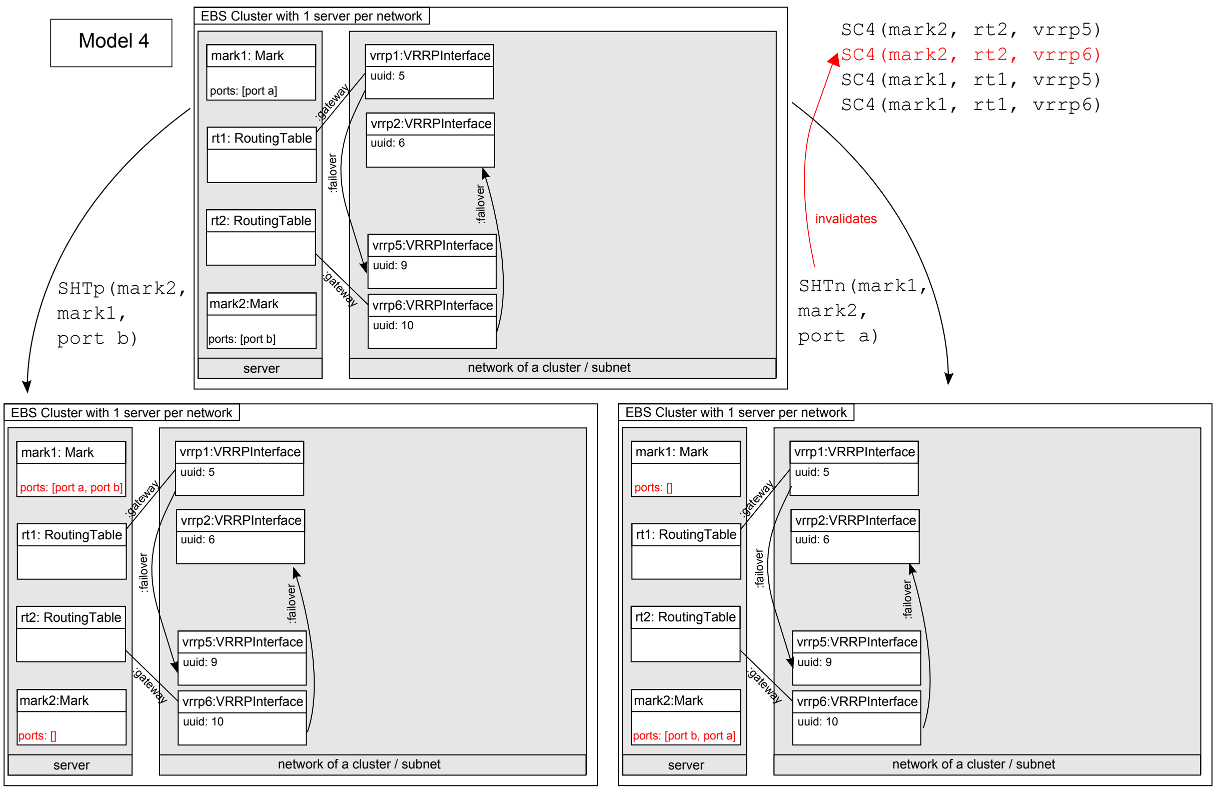


Figure 5.7: *SHTp* and *SHTn* change activities and safety constraints in a one-server EBS cluster in Model$_4$.

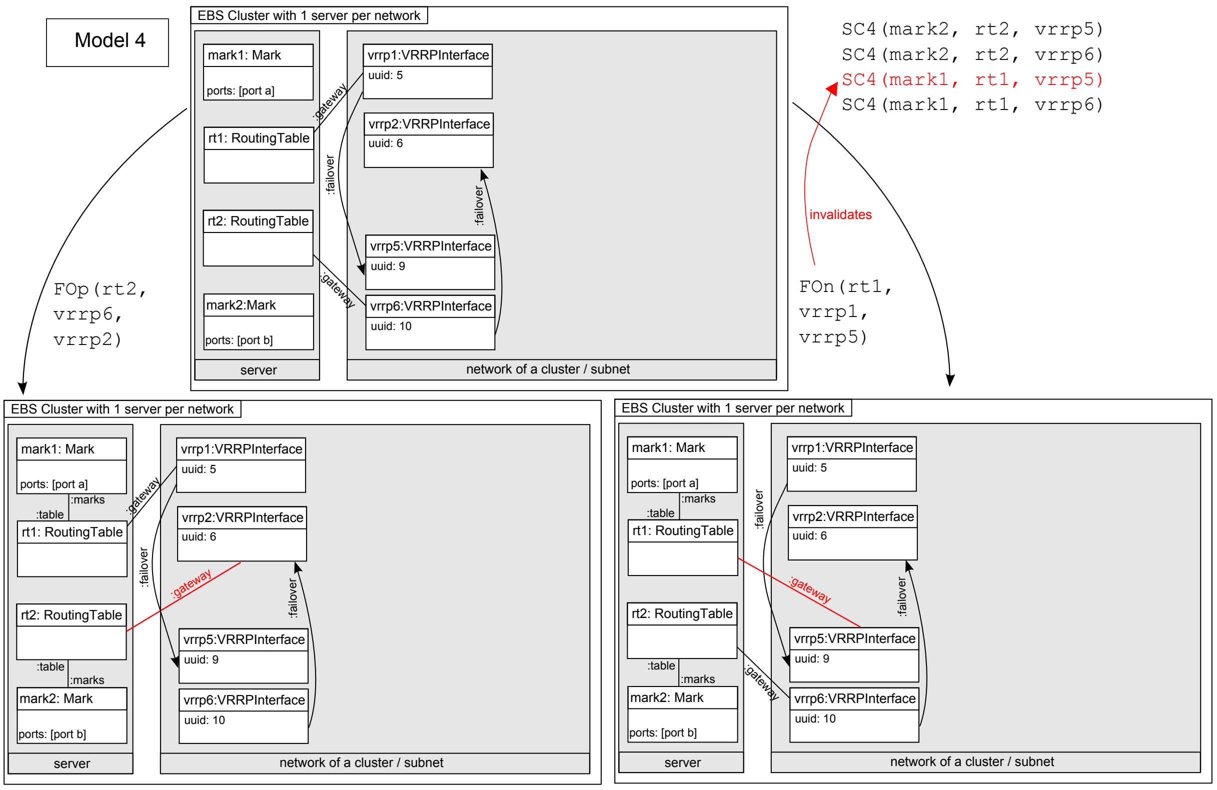


Figure 5.8: *FOp* and *FOn* change activities and safety constraints in a one-server EBS cluster in Model$_4$.

low-capacity router interface. Then, the safety constraint is violated if a `mark` that routes high-capacity traffic (`port a`) is associated to a routing table (`rt`) that routes towards a low-capacity router interface (`vrrp`). The logical description of static routing change activities remains the same as in Model$_1$.

Notice that the graphical model changes and the violations of safety constraints can be comprehended by substituting the logical descriptions of change activities and safety constraints in Figures 5.7 and 5.8.

Model 5: A Minimal Model for Dynamic Routing Changes with Manual Metrics

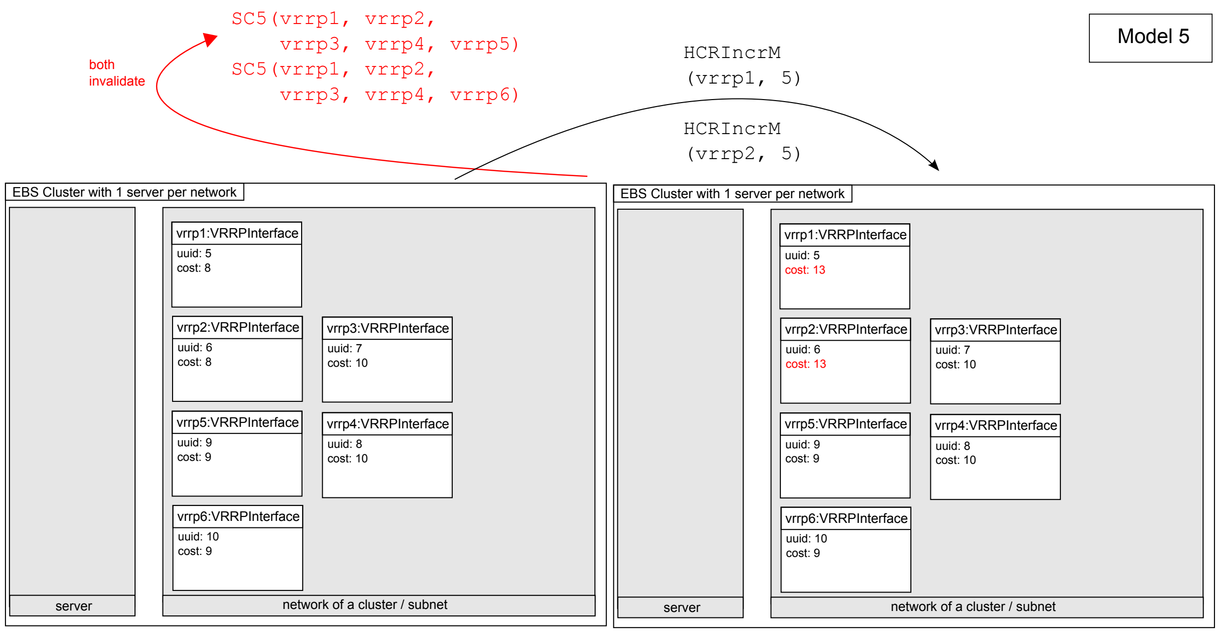


Figure 5.9: *HCRIncrM* change activity and safety constraints in a one-server EBS cluster in Model$_5$.

Model$_5$ is the minimal model to verify all types of dynamic routing changes when manual metrics are used, i.e., for *HCRIncrM* and *LCRDecrM* change activities. Figure 5.9 depicts an instance of Model$_5$. Alternatively, Table 5.2 describes Model$_5$ in a tabular fashion. Model$_5$ only covers instances of VRRPInterface Configuration Items that only hold one property, the manually configured cost/metric of the interface.

Safety constraint `SC5/SC5'` protects the network of Model$_5$ from an overload caused by workloads comprising *HCRIncrM* or *LCRDecrM* change activities.

`SC5/SC5'(vrrp1: VRRPInterface, vrrp2: VRRPInterface, vrrp3: VRRPInterface, vrrp4: VRRPInterface, vrrp: VRRPInterface)`

`pred`$_1$: | `vrrp1.cost < vrrp.cost`

`pred`$_2$: | `vrrp2.cost < vrrp.cost`

`pred`$_3$: | `vrrp3.cost < vrrp.cost`

`pred`$_4$: | `vrrp4.cost < vrrp.cost`

`pre SC5 :` | $\vee_{i=1\ldots4}$ `pred`$_i$ (for workloads not comprising *LCRDecrM* or *LCRDecrOSPF* change activities to interface `vrrp`)

`pre SC5':` | $\wedge_{i=1\ldots4}$ `pred`$_i$ (for workloads comprising *LCRDecrM* or *LCRDecrOSPF* change activities to interface `vrrp`)

`SC5/SC5'` needs to be instantiated for any combination of the four high-capacity router interfaces of a network and all low-capacity interfaces of a network. All in all, two instances (per network) are needed to protect the configuration from an overload. Both instances of `SC5` are depicted in the upper left corner of Figure 5.9. If properly instantiated, `SC5` reads as follows. The cost/metric of at least one of the four high-capacity VRRPInterfaces (parameters `vrrp1` to `vrrp4`) needs to be cheaper than the cost of low-capacity router interface `vrrp`. Notice that if we allow the occurrence of *LCRDecrM* change activities, the Boolean or ($\vee$) in the precondition of `SC5` has to be changed to a logical and ($\wedge$). The rationale behind this modification has previously been given in Section 5.1.3 where `SC5/SC5'` has been discussed in more detail. An example involving Model$_5$, safety constraint `SC5'`, and an *LCRDecrM* change activity can be found in Appendix C, Figure C.1. The change activities to increase or decrease the manual cost/metric of an interface match the logical description of *HCRIncrM* and *LCRDecrM* change activities previously given in Section 5.1.3:

```
HCRIncrM(interface:VRRPInterface, int:delta)
eff1:   | interface.cost.inc(delta)
```

```
LCRDecrM(interface:VRRPInterface, int:delta)
eff1:   | interface.cost.dec(delta)
```

According to the previously discussed scenarios for a network outage in a static routing environment with manual metrics (see Section 5.1.2), we consider two scenarios how a network overload can appear in Model$_5$: (1) Two *HCRIncrM* change activities increase the costs of the high-capacity router interfaces currently being used for routing such that the route via a low-capacity router interface becomes cheaper. See the two instances of *HCRIncrM* in Figure 5.9. (2) The metric of a low-capacity router interface is decreased such that the interface becomes the newest cheapest interface of the network. Figure C.1 (Appendix C) depicts such an instance of *LCRDecrM*.

Notice that the execution of the change activities depicted in Figure 5.9 cause the graphical changes to the model and the logical violation of the red instances of safety constraint `SC5`.

Model 6: A Minimal Model for Dynamic Routing Changes with OSPF Metrics

Model$_6$ is the minimal model used to verify *HCRIncrOSPF* and *LCRDecrOSPF* change activities. Figure 5.10 depicts an instance of Model$_6$. Similar to Model$_5$, this model only comprises interfaces of a subnet. Different to Model$_5$, properties for the reference and interface bandwidth need to be added to the VRRPInterface Configuration Items to take into account the additional configuration parameters for automatically computed OSPF metrics.

The safety constraint to protect the network from an overload caused by OSPF routing change activities is exactly the same as for Model$_5$. A specification of the *HCRIncrOSPF* and *LCRDecrOSPF* change activities in change verification logic has been previously given in Section 5.1.3:

```
HCRIncrOSPF(interface:VRRPInterface, int:delta)
eff1:   | interface.refbw.inc(delta)
eff2:   | interface.cost.inc(delta / interface.intfbw)
```

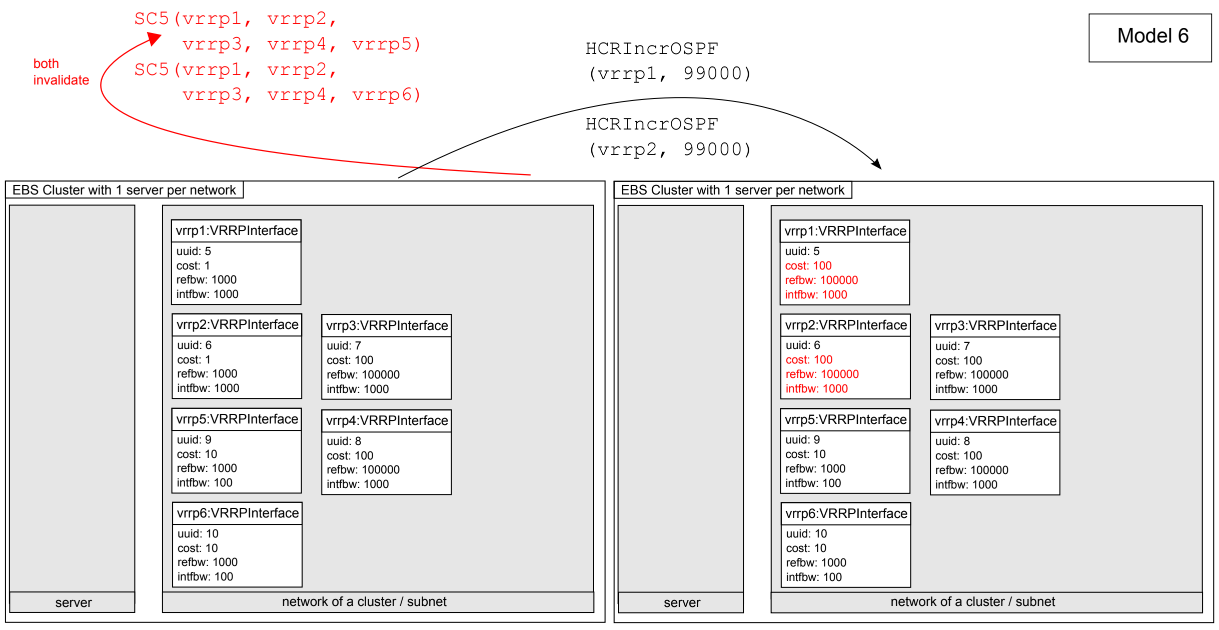


Figure 5.10: *HCRIncrOSPF* change activity and safety constraints in a one-server EBS cluster in Model$_6$.

```
LCRDecrOSPF(interface:VRRPInterface, int:delta)
eff₁:   | interface.refbw.dec(delta)
eff₂:   | interface.cost.dec(delta / interface.intfbw)
```

According to the scenarios (see Section 5.1.3) there are two ways how a network overload can appear in Model$_6$: (1) Two *HCRIncrOSPF* change activities increase the reference bandwidth of the high-capacity interfaces such that a route via a low-capacity router interface becomes cheaper. See the two instances of *HCRIncrOSPF* in Figure 5.10 for an example. (2) The reference bandwidth of a low-capacity router interface is decreased (and thus the cost of the interface) such that the interface becomes the newest cheapest interface on the network. For this case the slightly more restrictive safety constraint `SC5'` has to be used (similar to Model$_5$) to preserve the soundness of the verification algorithm. Figure C.2 in Appendix C depicts such a configuration and its modification by an *LCRDecrOSPF* change activity.

Model 7: A Detailed Model to Describe Dynamic Routing Changes with Manual Metrics

Model$_7$ is a detailed model to verify *HCRIncrM* and *LCRDecrM* change activities. Different to Model$_5$, it is not minimal because it also describes router and server Configuration Items. Figure 5.11 depicts an instance of Model$_7$ that describes the configuration of a one-server EBS cluster. Alternatively, refer to Table 5.2, which describes the model as well. The safety constraints (`SC5` or `SC5'`) to protect the network from an overload are the same as in the other dynamic routing models (Model$_5$, Model$_6$, and Model$_8$). The specification of *HCRIncrM* and *LCRDecrM* change activities remains unchanged. Similar to the other models, `SC5'` needs to be used once the workload comprises *LCRDecrM* change activities. See Figure C.3 in Appendix C for an example configuration and its modification by an *LCRDecrOSPF* change activity.

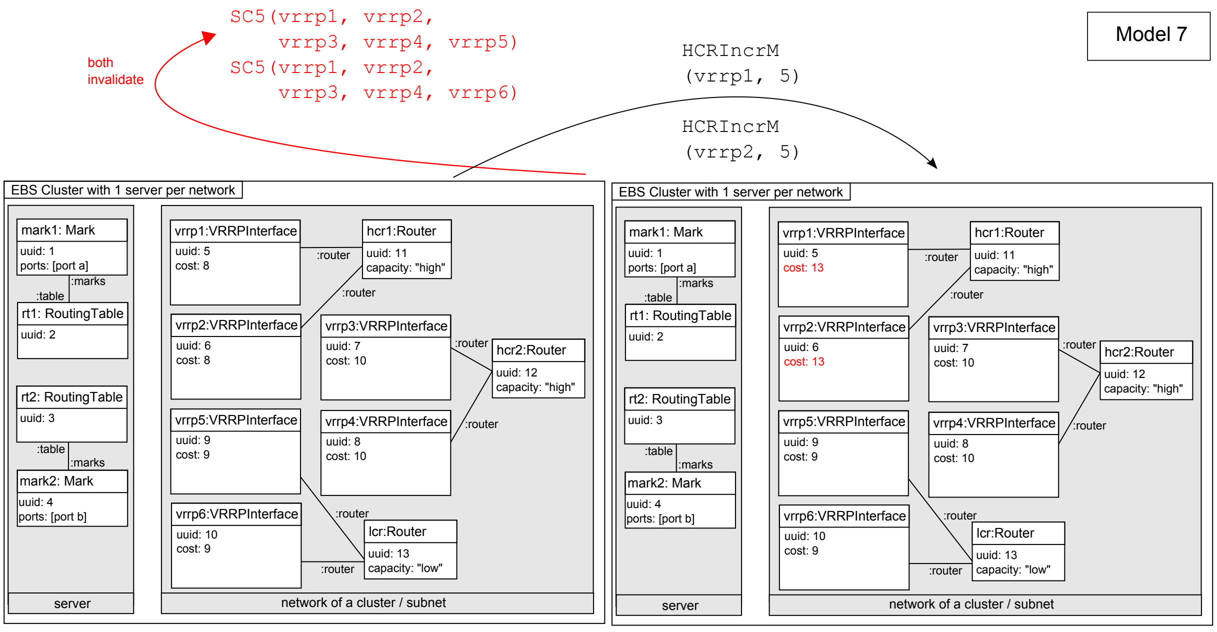

Model 7
both invalidate
SC5(vrrp1, vrrp2, vrrp3, vrrp4, vrrp5)
SC5(vrrp1, vrrp2, vrrp3, vrrp4, vrrp6)
HCRIncrM (vrrp1, 5)
HCRIncrM (vrrp2, 5)
EBS Cluster with 1 server per network
server
network of a cluster / subnet

Figure 5.11: *HCRIncrM* change activity and safety constraints in a one-server EBS cluster in $Model_7$.

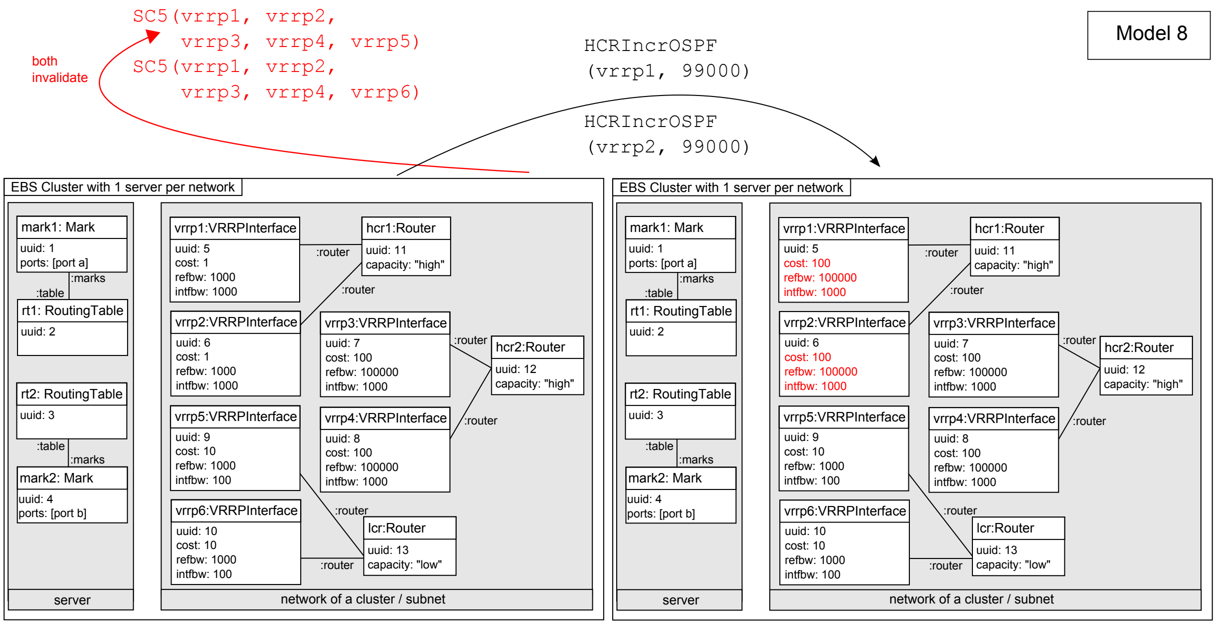

Model 8
both invalidate
SC5(vrrp1, vrrp2, vrrp3, vrrp4, vrrp5)
SC5(vrrp1, vrrp2, vrrp3, vrrp4, vrrp6)
HCRIncrOSPF (vrrp1, 99000)
HCRIncrOSPF (vrrp2, 99000)
EBS Cluster with 1 server per network
server
network of a cluster / subnet

Figure 5.12: *HCRIncrOSPF* change activity and safety constraints in a one-server EBS cluster in $Model_8$.

Model 8: A Detailed Model to Describe Dynamic Routing Changes with OSPF Metrics

$Model_8$ is a detailed model to verify *HCRIncrOSPF* and *LCRDecrOSPF* change activities. Figure 5.12 depicts an instance of $Model_8$ that describes the configuration of a one-server EBS cluster. The safety constraints (`SC5` or `SC5'`) to protect the network from an overload are the same as in the other dynamic routing models ($Model_5$, $Model_6$, and $Model_7$). The specification of *HCRIncrOSPF* and *LCRDecrOSPF* change activities remains unchanged. Similar to the other models, `SC5'` needs to be used once the workload comprises *LCRDecrOSPF* change activities. See Figure C.4 in Appendix C for an example configuration and its modification by an *LCRDecrOSPF* change activity.

5.4 Experimental Evaluation

This section evaluates the extended partial-order reduction model checker against NuSMV and SPiN using the Amazon case study. The experimental setup is described in Section 5.4.1. Section 5.4.2 compares the runtime complexity and performance of the special purpose model checker against NuSMV and SPiN. After that, we compare the maximum solvable problem sizes in a memory- and time-constrained environment depending on the model checkers, the CMDB models, and change workloads in Section 5.4.3. The robustness of the runtime performance of the model checkers is analyzed in Section 5.4.4. Finally, Section 5.4.5 compares the worst-case verification performance of the special purpose model checker against the best-case performance of the NuSMV and SPiN model checker.

5.4.1 Benchmarks

In Section 5.3.2 we introduced eight different models that can each verify different types of workloads (see Table 5.1). All in all, there are 32 combinations of models and workloads supported by them. We call these 32 combinations a benchmark. Table 5.3 depicts all 32 benchmarks (first multicolumn) and the Configuration Items, change activities, and safety constraints comprised in each benchmark for a server and network of a subnet. For example, consider the first row of Table 5.3, which describes the characteristics of the *SHTp* workload benchmark on Model$_1$. For this benchmark, the per server and per network multicolumns describe:

- The Configuration Items (for each server and each network) that are comprised in the model. For example, Model$_1$ comprises two Mark and two RoutingTable Configuration Items for every EBS server and vrrp1-vrrp6 (router interfaces), hcr1-2 (high-capacity routers), and lcr (low-capacity router) Configuration Items for every subnet of a cluster.
- The change activities that are comprised in the workload and are instantiated over every network or server in the CMDB. For example, for the combination of Model$_1$ and the *SHTp* workload one *SHTp* change activity is instantiated on every server and no changes are instantiated over the network Configuration Items.
- The safety constraints that are instantiated to protect the configuration described by a model from an overload. For example, when a pure *SHTp* workload appears on Model$_1$, four safety constraint instances of SC1 need to be instantiated over every server to protect the configuration.

Thus, if we benchmark a *SHTp* workload on a CMDB comprising 10,000 servers, then the CMDB comprises 40,000 Configuration Items describing servers (4 CIs per server, see Table 5.3) and 360 network CIs (9 CIs per subnet, 40 subnets assuming 255 servers per subnet). Furthermore, as it can be observed in Table 5.3, verifying the *SHTp* workload on Model$_1$ means to verify 10,000 *SHTp* change activities (one *SHTp* change per server) against 40,000 instances of safety constraint SC1 (4 instances of SC1 per server).

The logical specification of change activities and safety constraints as well as the Configuration Items modeled in each model have previously been introduced in Section 5.3.2. Each

Benchmark		per Server			per Network		
Model	Workload	CIs	Instantiated changes	SCs	CIs	Instantiated changes	SCs
Model$_1$	*SHTp*	2x mark, 2x rt	1x *SHTp*	4x SC1	vrrp1-6, hcr1-2, lcr		
↑	*SHTn*	↑	1x *SHTn*	↑	↑		
↑	*SHTp* & *SHTn*	↑	1x *SHTp*, 1x *SHTn*	↑	↑		
↑	*FOp*	↑	1x *FOp*	↑	↑		
↑	*FOn*	↑	1x *FOn*	↑	↑		
↑	*FOp* & *FOn*	↑	1x *FOp*, 1x *FOn*	↑	↑		
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	↑	1x *SHTp*, 1x *SHTn*, 1x *FOp*, 1x *FOn*	↑	↑		
Model$_2$	*SHTp*	2x mark	1x *SHTp*	1x SC2	none		
↑	*SHTn*	↑	1x *SHTn*	↑	↑		
↑	*SHTp* & *SHTn*	↑	1x *SHTp*, 1x *SHTn*	↑	↑		
Model$_3$	*FOp*	2x rt	1x *FOp*	2x SC3	vrrp1-2, vrrp5-6		
↑	*FOn*	↑	1x *FOn*	↑	↑		
↑	*FOp* & *FOn*	↑	1x *FOp*, 1x *FOn*	↑	↑		
Model$_4$	*SHTp*	2x mark, 2x rt	1x *SHTp*	4x SC4	vrrp1-2, vrrp5-6		
↑	*SHTn*	↑	1x *SHTn*	↑	↑		
↑	*SHTp* & *SHTn*	↑	1x *SHTp*, 1x *SHTn*	↑	↑		
↑	*FOp*	↑	1x *FOp*	↑	↑		
↑	*FOn*	↑	1x *FOn*	↑	↑		
↑	*FOp* & *FOn*	↑	1x *FOp*, 1x *FOn*	↑	↑		
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	↑	1x *SHTp*, 1x *SHTn*, 1x *FOp*, 1x *FOn*	↑	↑		
Model$_5$	*HCRIncrM*				vrrp1-6	2x *HCRIncrM*	2x SC5
↑	*LCRDecrM*				↑	1x *LCRDecrM*	2x SC5’
↑	*HCRIncrM* & *LCRDecrM*				↑	2x *HCRIncrM*, 1x *LCRDecrM*	2x SC5’
Model$_6$	*HCRIncrOSPF*				vrrp1-6	2x *HCRIncrOSPF*	2x SC5
↑	*LCRDecrOSPF*				↑	1x *LCRDecrOSPF*	2x SC5’
↑	*HCRIncrOSPF* & *LCRDecrOSPF*				↑	2x *HCRIncrOSPF*, 1x *LCRDecrOSPF*	2x SC5’
Model$_7$	*HCRIncrM*	2x mark, 2x rt			vrrp1-6, hcr1-2, lcr	2x *HCRIncrM*	2x SC5
↑	*LCRDecrM*	↑			↑	1x *LCRDecrM*	2x SC5’
↑	*HCRIncrM* & *LCRDecrM*	↑			↑	2x *HCRIncrM*, 1x *LCRDecrM*	2x SC5’
Model$_7$	*HCRIncrOSPF*	2x mark, 2x rt			vrrp1-6, hcr1-2, lcr	2x *HCRIncrOSPF*	2x SC5
↑	*LCRDecrOSPF*	↑			↑	1x *LCRDecrOSPF*	2x SC5’
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	↑			↑	2x *HCRIncrOSPF*, 1x *LCRDecrOSPF*	2x SC5’

Table 5.3: Characteristics of the 32 change verification benchmarks. See Section 5.3.2 for a description of the models, their Configuration Items, and the logical specification of safety constraints and change activities.

benchmark is then run on every model checker taking into account the specific optimization techniques available to the model checker (see Section 5.2). With each three optimization techniques for the NuSMV and SPiN model checker and a total of two optimization techniques for the special purpose model checker, we obtain $32 * 8 = 256$ different combinations (called configuration of a benchmark) that have been benchmarked depending on the size of the CMDB. Much care and caution has been invested in the specification of the inputs to the different model checkers to guarantee that all model checkers have to verify the same logical formulas, that models are the same for each model checker, that the same workload is verified, and that change activities have the same logical specification.

All benchmarks have been performed on an Intel Xeon Processor with 2.8Ghz and 4 GB of RAM. Each model checker has been restricted to a maximum memory of 1GB and a maximum processing duration of 12 minutes to ensure that runtime complexity is not achieved at the cost of memory complexity or the other way round. All benchmark results presented herein comprise all problem sizes that can be solved within these time and memory constraints if not noted otherwise.

5.4.2 Runtime Complexity and Performance

This section summarizes the runtime complexity and performance of each model checker for the configurations of the 32 benchmarks. Table 5.4 consolidates the complexity results and Tables 5.5 through 5.6 summarize the relative runtime performance of the model checkers and optimization techniques on small and large problem instances. Figures B.1 through B.10 in Appendix B.1 provide the graphs of the benchmarks that are summarized in the tables.

Due to the numerous benchmarks and optimization techniques covered, we only summarize the results herein and exemplary refer to the *SHTn* on Model$_4$ benchmark (see Figure 5.13) to explain the results.

Performance Results of the Special Purpose Model Checker

This paragraph summarizes the complexity and performance results of the extended partial-order reduction model checker for the configurations of the 32 benchmarks. An overview of the results is given in Tables 5.4 through 5.6.

- **The special purpose model checker has linear runtime complexity:**
  The extended partial-order reduction model checker has linear runtime complexity in each of the benchmarks independent of the optimization technique. This experimentally confirms the results of the complexity analysis previously presented in Section 4.2. There we concluded that the special purpose model checker solves all three IT change verification problems in time linear in the number of change activities, safety constraints, and the size of the CMDB assuming that changes and constraints are equally distributed over the Configuration Items of the CMDB. The linear runtime complexity can be observed in Figure 5.13 (and for all other benchmarks in Appendix B.1), which depicts the log-log graph of the verification time of the *SHTn* workload on Model$_4$ depending on the size of the CMDB. In log-log graphs linear runtime complexity can be observed in curves with a slope of one. Thus, the special purpose model checker has linear runtime complexity for the *SHTn* workload on Model$_4$. This holds for any other benchmark (see Figures B.1a through B.10c in Appendix B.1) as well.

Benchmark		OWN configurations		NuSMV configurations			SPIN configurations			
Model	Workload	JITC	noJITC	BDD CTL	BDD LTL	SAT LTL	NCP	CP	DMA	Figure
Model$_1$	*SHTp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.1a
↑	*SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.1b
↑	*FOp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.2a
↑	*FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.2b
↑	*SHTp* & *SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.2c
↑	*FOp* & *FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.2d
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.2e
Model$_2$	*SHTp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.3a
↑	*SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.3b
↑	*SHTp* & *SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.3c
Model$_3$	*FOp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	[**polyn** : **exp**]	Fig. B.4a
↑	*FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	[**polyn** : **exp**]	Fig. B.4b
↑	*FOp* & *FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.4c
Model$_4$	*SHTp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.5a
↑	*SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.5b
↑	*FOp*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	[**polyn** : **exp**]	Fig. B.5c
↑	*FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	[**polyn** : **exp**]	Fig. B.5d
↑	*SHTp* & *SHTn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.5e
↑	*FOp* & *FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.5f
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.6
Model$_5$	*HCRIncrM*	**linear**	**linear**	[**polyn** : **exp**]	[**polyn** : **exp**]	[**polyn** : **exp**]	**exp**	**exp**	**exp**	Fig. B.7a
↑	*LCRDecrM*	**linear**	**linear**	[**polyn** : **exp**]	[**polyn** : **exp**]	[**polyn** : **exp**]	**exp**	**exp**	**exp**	Fig. B.7b
↑	*HCRIncrM* & *LCRDecrM*	**linear**	**linear**	[**polyn** : **exp**]	[**polyn** : **exp**]	[**polyn** : **exp**]	**exp**	**exp**	**exp**	Fig. B.7c
Model$_6$	*HCRIncrOSPF*	**linear**	**linear**	[**polyn** : **exp**]	[**polyn** : **exp**]	[**polyn** : **exp**]	**exp**	**exp**	**exp**	Fig. B.8a
↑	*LCRDecrOSPF*	**linear**	**linear**	[**polyn** : **exp**]	[**polyn** : **exp**]	[**polyn** : **exp**]	**exp**	**exp**	**exp**	Fig. B.8b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**exp**	Fig. B.8c
Model$_7$	*HCRIncrM*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**indefinable**	Fig. B.9a
↑	*LCRDecrM*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**indefinable**	Fig. B.9b
↑	*HCRIncrM* & *LCRDecrM*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	[**polyn** : **exp**]	[**polyn** : **exp**]	**indefinable**	Fig. B.9c
Model$_8$	*HCRIncrOSPF*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**indefinable**	Fig. B.10a
↑	*LCRDecrOSPF*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	**exp**	**exp**	**indefinable**	Fig. B.10b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**linear**	**linear**	**polyn**	**polyn**	**polyn**	[**polyn** : **exp**]	[**polyn** : **exp**]	**indefinable**	Fig. B.10c

Table 5.4: Runtime complexity of verification depending on model, workload, model checker, and optimization technique on small and large CMDBs. See graphs in Section B.1, Appendix B.

Benchmark		OWN configurations		NuSMV configurations			SPIN configurations			
Model	Workload	JITC	noJITC	BDD CTL	BDD LTL	SAT LTL	NCP	CP	DMA	Figure
Model$_1$	*SHTp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.1a
$\uparrow$	*SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.1b
$\uparrow$	*FOp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.2a
$\uparrow$	*FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.2b
$\uparrow$	*SHTp* & *SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.2c
$\uparrow$	*FOp* & *FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.2d
$\uparrow$	*SHTp* & *SHTn* & *FOp* & *FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.2e
Model$_2$	*SHTp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.3a
$\uparrow$	*SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.3b
$\uparrow$	*SHTp* & *SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.3c
Model$_3$	*FOp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.4a
$\uparrow$	*FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.4b
$\uparrow$	*FOp* & *FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.4c
Model$_4$	*SHTp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5a
$\uparrow$	*SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5b
$\uparrow$	*FOp*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5c
$\uparrow$	*FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5d
$\uparrow$	*SHTp* & *SHTn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5e
$\uparrow$	*FOp* & *FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.5f
$\uparrow$	*SHTp* & *SHTn* & *FOp* & *FOn*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.6
Model$_5$	*HCRIncrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.7a
$\uparrow$	*LCRDecrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.7b
$\uparrow$	*HCRIncrM* & *LCRDecrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.7c
Model$_6$	*HCRIncrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**fastest**	**slowest**	Fig. B.8a
$\uparrow$	*LCRDecrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**fastest**	**slowest**	Fig. B.8b
$\uparrow$	*HCRIncrOSPF* & *LCRDecrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**fastest**	**fastest**	**slowest**	Fig. B.8c
Model$_7$	*HCRIncrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9a
$\uparrow$	*LCRDecrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9b
$\uparrow$	*HCRIncrM* & *LCRDecrM*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9c
Model$_8$	*HCRIncrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10a
$\uparrow$	*LCRDecrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10b
$\uparrow$	*HCRIncrOSPF* & *LCRDecrOSPF*	**slowest**	**fastest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10c
	percentage fastest	0%	**100%**	**100%**	0%	0%	**81.25%**	28.13%	0%	
	percentage slowest	100%	**0%**	**0%**	100%	0%	**0%**	0%	100%	
	percentage intermediate	0%	**0%**	**0%**	0%	100%	**18.75%**	71.87%	0%	

Table 5.5: Runtime performance of verification depending on model, workload, model checker, and optimization technique on small CMDBs (< 20 server). See graphs in Section B.1, Appendix B.

Benchmark		OWN configurations		NuSMV configurations			SPIN configurations			
Model	Workload	JITC	noJITC	BDD CTL	BDD LTL	SAT LTL	NCP	CP	DMA	Figure
Model$_1$	*SHTp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.1a
↑	*SHTn*	**fastest**	**slowest**	**intermediate**	**fastest**	**slowest**	**intermediate**	**fastest**	**slowest**	Fig. B.1b
↑	*FOp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.2a
↑	*FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.2b
↑	*SHTp* & *SHTn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.2c
↑	*FOp* & *FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.2d
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.2e
Model$_2$	*SHTp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.3a
↑	*SHTn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.3b
↑	*SHTp* & *SHTn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.3c
Model$_3$	*FOp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**fastest**	**intermediate**	**slowest**	Fig. B.4a
↑	*FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.4b
↑	*FOp* & *FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.4c
Model$_4$	*SHTp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.5a
↑	*SHTn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.5b
↑	*FOp*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.5c
↑	*FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.5d
↑	*SHTp* & *SHTn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.5e
↑	*FOp* & *FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.5f
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**fastest**	**intermediate**	**slowest**	Fig. B.6
Model$_5$	*HCRIncrM*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.7a
↑	*LCRDecrM*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.7b
↑	*HCRIncrM* & *LCRDecrM*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.7c
Model$_6$	*HCRIncrOSPF*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.8a
↑	*LCRDecrOSPF*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.8b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**fastest**	**slowest**	**intermediate**	**slowest**	**fastest**	**intermediate**	**fastest**	**slowest**	Fig. B.8c
Model$_7$	*HCRIncrM*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9a
↑	*LCRDecrM*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9b
↑	*HCRIncrM* & *LCRDecrM*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.9c
Model$_8$	*HCRIncrOSPF*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10a
↑	*LCRDecrOSPF*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**fastest**	**slowest**	**fastest**	**slowest**	**intermediate**	**intermediate**	**fastest**	**slowest**	Fig. B.10c
	percentage fastest	**100%**	0%	37.5%	0%	**62.5%**	25%	**75%**	0%	
	percentage slowest	**0%**	100%	0%	100%	**0%**	0%	**0%**	100%	
	percentage intermediate	**0%**	0%	62.5%	0%	**37.5%**	75%	**25%**	0%	

Table 5.6: Runtime performance of verification depending on model, workload, model checker, and optimization technique on large CMDBs. See graphs in Section B.1, Appendix B.

Benchmark		OWN configurations		NuSMV configurations			SPIN configurations		
CMDB size	Position	JITC	noJITC	BDD CTL	BDD LTL	SAT LTL	NCP	CP	DMA
large	fastest	**100%**	0%	37.5%	0%	**62.5%**	25%	**75%**	0%
large	slowest	**0%**	100%	0%	100%	**0%**	0%	**0%**	100%
large	intermediate	**0%**	0%	62.5%	0%	**37.5%**	75%	**25%**	0%
small	fastest	0%	**100%**	**100%**	0%	0%	**81.25%**	28.13%	0%
small	slowest	100%	**0%**	**0%**	100%	0%	**0%**	0%	100%
small	intermediate	0%	**0%**	**0%**	0%	100%	**18.75%**	71.87%	0%
	max size	**81.25%**	68.75%	**56.25%**	18.75%	59.38%	25%	**96.88%**	28.13%
	min size	**18.75%**	31.25%	**9.38%**	81.25%	18.75%	62.5%	**3.13%**	40.63%
	middle size	**0%**	0%	**34.38%**	0%	21.88%	12.5%	**0%**	31.25%

Table 5.7: Average performance of each model checker in the benchmarks depending on the optimization technique chosen. For example, SAT LTL is the best choice in terms of performance on large CMDBs in 62.5% of all cacses for the NuSMV model checker.

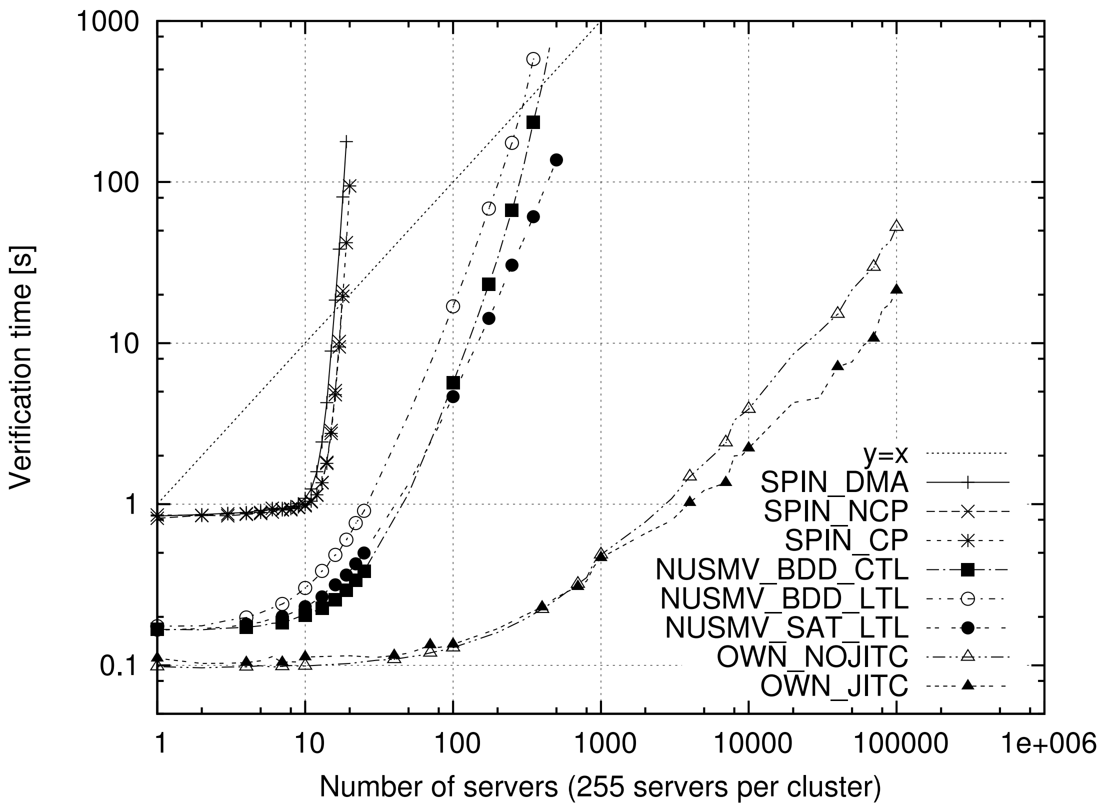


Figure 5.13: Time needed to verify a *SHTn* workload on Model$_4$ depending on the number of servers, the model checker, and optimization technique used.

- **The special purpose model checker always delivers the best runtime performance for a given model:**
  For a fixed model the special purpose model checker always outperforms NuSMV and SPiN - even when the worst-case optimization technique is chosen for the special purpose model checker. For example, this can be observed in Figure 5.13 because the JITC and noJITC measurements always lie well below all curves of the NuSMV and SPiN model checker. This holds for all 32 benchmarks (see Figures B.1a through B.10c in Appendix B.1) as well. We later show in Section 5.4.5 that this even holds among all models, i.e., that the worst-case choice of model and optimization technique made for the

extended partial-order reduction model checker is always better than the best-case choice that can be made for NuSMV and SPiN.

- **On large domains just-in-time compilation outperforms code interpretation:**
  Throughout each benchmark we can observe that just-in-time compilation only improves runtime performance on large CMDB sizes while it imposes a runtime penalty on small CMDBs. The smaller the configuration of the CMDB, the larger the penalty caused by just-in-time compilation. This is the case because the time to compile code does not outweigh the faster execution performance of native code on small problem instances. For example, in Figure 5.13 noJITC is faster than JITC for problem sizes no larger than 600 servers. After that, the runtime improves when just-in-time compilation is activated. This behavior can be observed in any benchmark (see Figures B.1a through B.10c in Appendix B.1).

Performance Results of NuSMV Model Checker

This paragraph summarizes the complexity and performance results of the NuSMV model checker for the configurations of the 32 benchmarks. An overview of the results is given in Tables 5.4 through 5.6.

- **NuSMV mostly has polynomial runtime complexity:**
  NuSMV solves 84% of the 32 verification benchmarks in polynomial time and in the remaining benchmarks runtime complexity varies between polynomial and exponential. The runtime complexity reaches phases of exponential complexity for certain workloads verified with Model_5 and Model_6 (see Table 5.4). The polynomial runtime complexity or worse can be observed in Figure 5.13 as the BDD CTL, BDD LTL, and SAT LTL curves have a slope larger than one, which corresponds to polynomial or exponential complexity. Polynomial complexity was distinguished from exponential complexity by observing the corresponding semi-log graphs. In these graphs most curves grew slower than a straight line which confirms polynomial complexity[1]. Similar observations like for the *SHTn* workload on Model_4 can be made for the other benchmarks (see Figures B.1a-B.10c in Appendix B.1).

- **NuSMV performs better than SPiN and worse than the special purpose model checker in any benchmark:**
  For any benchmark even the worst-case optimization technique for NuSMV outperforms the best-case choice for the SPiN model checker. This can be easily observed in Figure 5.13, where the BDD CTL, BDD LTL, and SAT LTL curves always lie well below all measurements made for the SPiN model checker. This can also be observed in Figures B.1a through B.10c in Appendix B.1 that depict the results of the remaining benchmarks.

- **On large domains SAT LTL performs best most of the time. On small domains BDD CTL model checking performs best most of the time:**
  Table 5.6 consolidates the runtime performance results of the benchmarks on large CMDB sizes. In 62.5% of all benchmarks SAT LTL model checking provides the best runtime

[1]All conclusions about the runtime complexity of NuSMV presented herein have been drawn from the graphical analysis of the log-log and semi-log plots.

performance followed by BDD CTL model checking (best in 37.5% of all cases) and BDD LTL model checking (always providing the worst performance on large CMDBs).

In turn, on small CMDB instances (see Table 5.5) BDD CTL model checking always provides the best performance while the winner for large problem instances (SAT LTL) only provides the second best performance. Similar to large instances, BDD LTL model checking always delivers the worst performance.

Compared to the special purpose model checker, the performance of NuSMV can be less easily predicted. For the special purpose model checker JITC always performs better on large CMDBs and noJITC is faster on small CMDBs - an obvious recommendation on which optimization technique to prefer depending on the size of the CMDB. In turn, for NuSMV an obvious recommendation can only be provided for small CMDB instances (BDD CTL model checking) while SAT LTL - the best choice for large CMDB instances - will only achieve the second best performance in 37.5% of all benchmarks.

Notice that Figure 5.13 provides one of the examples where BDD CTL model checking initially performs better than SAT LTL model checking on small CMDB instances but is then overtaken by SAT LTL model checking when the CMDB size increases.

Performance Results of SPiN Model Checker

This paragraph summarizes the complexity and performance results of the SPiN model checker for the configurations of the 32 benchmarks. An overview of the results is given in Tables 5.4 through 5.6.

- **SPiN mostly has exponential runtime complexity:**
  The SPiN model checker shows exponential runtime complexity in most of the 32 benchmarks. For a few cases runtime complexity varies between polynomial and exponential complexity. For some benchmarks involving Model$_7$ and Model$_8$ in combination with the DMA optimization technique we were unable to graphically determine runtime complexity because only two problem instances could be solved. Exponential runtime complexity can be identified in Figure 5.13 because the slopes of the NCP, CP, and DMA curves are larger than one and the curves appear as straight lines in the corresponding semi-log graphs.[1] For the sake of brevity the semi-log graphs of the benchmarks are not provided, but the log-log graphs can be found in Figures B.1a-B.10c in Appendix B.1.

- **SPiN always performs worse than NuSMV in every benchmark:**
  In every benchmark, even the best-case optimization technique for SPiN yields longer verification times than the worst-case choice of the NuSMV model checker. This can be easily observed in Figure 5.13, where all NuSMV curves lie well below all measurements made for the SPiN model checker. This is also the case in Figures B.1a through B.10c in Appendix B.1.

- **On large domains compression performs best on average. On small domains no compression performs best on average:**
  On large domains state compression (CP) provides the best runtime performance in 75% of all benchmarks. In the remaining 25% of all cases no compression (NCP) is faster than compression (CP) (see Table 5.6 for results).

[1] All conclusions about the runtime complexity of SPiN presented herein have been drawn from the graphical analysis of the log-log and semi-log plots.

In turn, on small domains no compression provides the best performance in 81% of all cases. In the remaining cases compression is faster on small CMDB instances. The deterministic minimal automaton strategy always has the worst-case runtime performance on small and large CMDB instances.

SPiN mainly profits from compression on large CMDBs. On small CMDBs it causes an overhead that slows down the verification performance such that the NCP strategy becomes faster. However, there are exceptions to the rule. Consequently, the performance of SPiN is more difficult to predict than the performance of the extended partial-order reduction model checker. For the latter JITC is always faster on large CMDBs and noJITC is faster on small CMDBs - a clear statement on which optimization technique to prefer depending on the size of the problem instance. In turn, optimal recommendations can only be made in 75% (on large CMDBs) or 81% (on small CMDBs) of all examined cases for the SPiN model checker.

5.4.3 Maximum Solvable Problem Instances in Time- and Memory-constrained Environments

Besides the time needed for verification, the maximum size of problems solvable in a time- and memory-constrained environment is of importance as well because verification problems of significant size can appear for IT change verification due to the scale of IT infrastructures. Table 5.8 depicts the results of the maximum problems sizes solvable[1] depending on the model, the workload, and optimization technique used. The results summarized in Table 5.8 are based on Figures B.1a-B.10c in Appendix B.1.

- **The special purpose model checker solves the largest problem instances:**
  In Figures B.1a through B.10c it can be observed that the special purpose model checker always solves the largest problem instances in the time- and memory-constrained environment - even if the best-case optimization technique is chosen for NuSMV and SPiN.

- **The NuSMV model checker solves larger problem instances than the SPiN model checker:**
  In all 32 benchmarks the NuSMV model checker always solves larger problem instances than the SPiN model checker - even if the most beneficial optimization technique is chosen for SPiN.

- **The best optimization techniques to solve the largest problem instances are:**

  - **SPiN model checker: Collapse compression (CP) solves the largest problem instances most of the time:**
    Not surprisingly compression enables the SPiN model checker to solve the largest problem instances in approx. 96.8% of all cases. State compression effectively reduces memory consumption. In addition to that, it only has a small impact on the performance because it is also the fastest strategy in 75% of all cases on large CMDBs (see Section 5.4.2). Consequently, CP is the SPiN strategy that most frequently solves the largest problems in the shortest time. Following CP, DMA solves the second largest problem instances and NCP the smallest instances.

[1]within 1GB of RAM and 12 minutes.

Benchmark		OWN configurations		NuSMV configurations			SPIN configurations			
Model	Workload	JITC	noJITC	BDD CTL	BDD LTL	SAT LTL	NCP	CP	DMA	Figure
Model$_1$	*SHTp*	**max**	**max**	**max**	**min**	middle	**min**	**max**	middle	Fig. B.1a
↑	*SHTn*	**min**	**max**	**max**	**min**	**max**	**min**	**max**	middle	Fig. B.1b
↑	*FOp*	**max**	**max**	**max**	**min**	middle	**min**	**max**	middle	Fig. B.2a
↑	*FOn*	**max**	**max**	**max**	**min**	**max**	**min**	**max**	middle	Fig. B.2b
↑	*SHTp* & *SHTn*	**min**	**max**	middle	**min**	**max**	**min**	**max**	**max**	Fig. B.2c
↑	*FOp* & *FOn*	**max**	**max**	**max**	**min**	middle	**min**	**max**	**min**	Fig. B.2d
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**min**	**max**	middle	**min**	**max**	**max**	**max**	**max**	Fig. B.2e
Model$_2$	*SHTp*	**max**	**min**	**max**	**min**	middle	**min**	**max**	**max**	Fig. B.3a
↑	*SHTn*	**max**	**min**	middle	**min**	**max**	**min**	**max**	**max**	Fig. B.3b
↑	*SHTp* & *SHTn*	**min**	**max**	middle	**min**	**max**	**min**	**max**	**max**	Fig. B.3c
Model$_3$	*FOp*	**max**	**min**	**max**	**min**	middle	middle	**max**	**min**	Fig. B.4a
↑	*FOn*	**max**	**max**	**max**	**min**	**max**	middle	**max**	**min**	Fig. B.4b
↑	*FOp* & *FOn*	**max**	**max**	middle	**min**	**max**	**max**	**max**	**min**	Fig. B.4c
Model$_4$	*SHTp*	**max**	**min**	**max**	**min**	middle	**min**	**max**	middle	Fig. B.5a
↑	*SHTn*	**max**	**max**	middle	**min**	**max**	**min**	**max**	middle	Fig. B.5b
↑	*FOp*	**max**	**min**	**max**	**min**	middle	**min**	**max**	**min**	Fig. B.5c
↑	*FOn*	**max**	**min**	**max**	**min**	**max**	**min**	**max**	**min**	Fig. B.5d
↑	*SHTp* & *SHTn*	**max**	**max**	middle	**min**	**max**	**min**	**max**	**max**	Fig. B.5e
↑	*FOp* & *FOn*	**max**	**max**	**max**	**min**	**max**	**min**	**max**	**min**	Fig. B.5f
↑	*SHTp* & *SHTn* & *FOp* & *FOn*	**max**	**max**	middle	**min**	**max**	**min**	**min**	**max**	Fig. B.6
Model$_5$	*HCRIncrM*	**max**	**min**	middle	**min**	**max**	**min**	**max**	middle	Fig. B.7a
↑	*LCRDecrM*	**max**	**min**	**min**	**min**	**max**	**min**	**max**	middle	Fig. B.7b
↑	*HCRIncrM* & *LCRDecrM*	**max**	**max**	**min**	**min**	**max**	**max**	**max**	**max**	Fig. B.7c
Model$_6$	*HCRIncrOSPF*	**max**	**min**	**min**	**min**	**max**	**min**	**max**	middle	Fig. B.8a
↑	*LCRDecrOSPF*	**max**	**min**	middle	**min**	**max**	**min**	**max**	middle	Fig. B.8b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**max**	**max**	middle	**min**	**max**	**max**	**max**	**max**	Fig. B.8c
Model$_7$	*HCRIncrM*	**min**	**max**	**max**	**max**	**min**	middle	**max**	**min**	Fig. B.9a
↑	*LCRDecrM*	**max**	**max**	**max**	**max**	**min**	**max**	**max**	**min**	Fig. B.9b
↑	*HCRIncrM* & *LCRDecrM*	**max**	**max**	**max**	**max**	**min**	**max**	**max**	**min**	Fig. B.9c
Model$_8$	*HCRIncrOSPF*	**max**	**max**	**max**	**max**	**min**	middle	**max**	**min**	Fig. B.10a
↑	*LCRDecrOSPF*	**max**	**max**	**max**	**max**	**min**	**max**	**max**	**min**	Fig. B.10b
↑	*HCRIncrOSPF* & *LCRDecrOSPF*	**min**	**max**	**max**	**max**	**min**	**max**	**max**	**min**	Fig. B.10c
	percentage max	**81.25%**	68.75%	**56.25%**	18.75%	59.38%	25%	**96.88%**	28.13%	
	percentage min	**18.75%**	31.25%	**9.38%**	81.25%	18.75%	62.5%	**3.13%**	40.63%	
	percentage middle	**0%**	0%	**34.38%**	0%	21.88%	12.5%	**0%**	31.25%	

Table 5.8: Size of the models verifiable within 12 minutes and 1GB of RAM depending on the model, workload, model checker, and optimization technique. See graphs in Appendix B.1.

- **NuSMV model checker: BDD CTL and SAT LTL solve roughly equally large problem instances:**
  BDD CTL (SAT LTL) solves the largest problem instances in 56% (59%) of all cases. Both can be regarded as roughly equally good when it comes to solve the largest problem instances. However, in 34% of all benchmarks BDD CTL solves the second largest problem instances while SAT LTL only solves 22% of all cases placed second. Consequently, considering first and second places BDD CTL provides the better strategy although it solves slightly less benchmarks placed first. However, BDD CTL is not the best recommendation in terms of performance on large CMDB sizes. Consequently, SAT LTL might still be a good choice. It has slightly worse scalability in terms of problem sizes but shows better runtime performance on large CMDBs (see Table 5.7).
- **The special purpose model checker: Just-in-time compilation (JITC) solves roughly equally large problem instances as noJITC:**
  We were unable to determine which optimization technique solves the larger problem instances as the same configurations of a benchmark led to slightly different memory consumptions. The fact that there is roughly a gap of 12% for which JITC was observed to solve larger problem instances than noJITC and the fact that there is only a small difference between the maximum and minimum problem sizes makes it impossible to give a recommendation on which strategy solves larger problem instances.

5.4.4 Robustness of Runtime Performance

In this section we show that the special purpose model checker is more robust than the NuSMV and SPiN model checker in terms of verification runtime for any of the benchmarks. Robustness of change verification is very important because we cannot expect a change manager to be trained to choose the most performant model and optimization technique for a change verification problem. Consequently, for IT change verification a model checker is required whose performance is as independent as possible in respect to the description of the IT infrastructure and the specification of change activities and safety constraints. To discuss robustness, we introduce a metric for its measurement.

For a workload w, i.e., a combination of change activities to be verified and a model checker m, we introduce the terms of worst- and best-case verification time as follows. The best-case verification time of a workload w using model checker m is the shortest runtime for verification that can be achieved with model checker m by selecting the combination of model (see Section 5.3.2) and optimization technique (see Section 5.2) that verifies the workload in the shortest time. Analogously, the worst-case verification time of a workload w using model checker m is the longest runtime for verification that is needed when the most adverse combination of model and optimization technique is used for model checker m. The difference between the worst-case and best-case verification performance of a workload w using a model checker m is called the maximum penalty of m for workload w. The maximum penalty describes the maximum additional verification runtime that occurs when an inexperienced change manager chooses the most unfavorable model and optimization technique instead of the optimum. Finally, we say that a model checker m_1 is more robust than a model checker m_2 with respect to a workload w if and only if the maximum penalty of m_1 (for workload w) is lower than that of m_2 (for workload w).

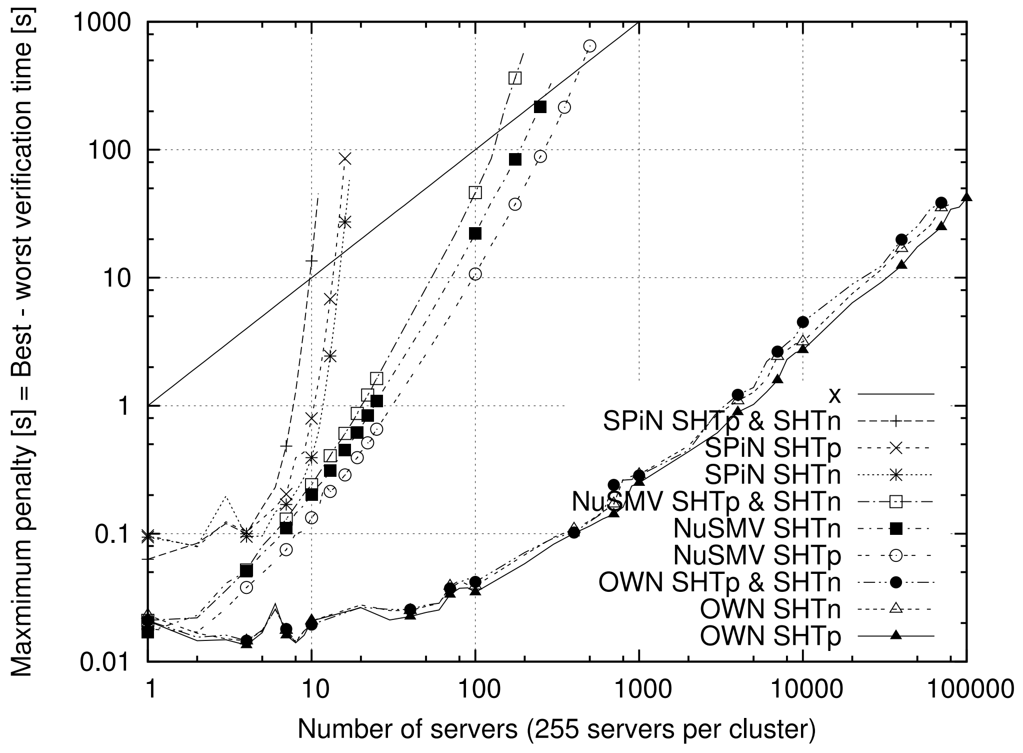


Figure 5.14: Maximum penalty, i.e., difference between worst- and best-case verification time, for the different model checkers and *SHTp*, *SHTn*, and *SHTp & SHTn* workloads.

Figure 5.14 depicts the maximum penalties for all three model checkers when verifying a *SHTp*, *SHTn*, or *SHTp & SHTn* workload. In this section we only discuss the results of these workloads. For the sake of completeness the benchmark results for all other workloads can be found in Appendix B.2 (Figures B.11a - B.11e).

In Figure 5.14 we observe that the maximum penalty of the special purpose model checker increases linearly with the size of the CMDB as the curves converge to a slope of one in the log-log graph. Thus, should a change manager - due to lack of knowledge in the design of efficient verification domains and models - choose an adverse model, he or she will not be penalized more than linearly in the size of the CMDB compared to the best model that could have been chosen.

In turn, if the NuSMV model checker is used, the penalty is polynomial in the size of the CMDB as the penalty graphs of NuSMV have a slope larger than one but still do not appear as a straight line in the corresponding semi-log graphs. Although phases of exponential runtime complexity can be observed for NuSMV for dynamic routing workloads (see Model$_5$ and Model$_6$ in Table 5.4), the maximum penalty of these workloads remains polynomial in the size of the CMDB. This is the case because the maximum penalty is only calculated for model sizes that are solvable by any combination of model and optimization technique. As the phases of exponential runtime complexity appear for CMDB sizes not solvable with all models and optimization techniques, the maximum penalty has polynomial complexity.

Robustness is mostly exponential in the size of the CMDB for the SPiN model checker. However, the lack of measurements did not allow us to determine the complexity of the maximum penalty for some workloads. For example, the maximum penalty cannot be determined for *HCRIncrOSPF*, *LCRDecrOSPF*, and *HCRIncrOSPF & LCRDecrOSPF* workloads (see Figure B.11d) because SPiN scales badly for this workload making it impossible to determine runtime complexity.

A complete coverage of the maximum penalties of the remaining workloads can be found in Figures B.11a - B.11e in Appendix B.2. All in all, the special purpose model checker is not only faster but also more robust than the SPiN and NuSMV model checker when it comes to verify the IT change workloads that could have caused Amazon's data center outage.

5.4.5 Worst- and Best-case Performance

In this section we show that the special purpose model checker always performs better than the NuSMV and SPiN model checker - independent of the models and optimization techniques chosen for each model checker. A related but slightly weaker statement has previously been shown in Section 5.4.2: The special purpose model checker is always the fastest model checker independent of the optimization technique chosen if verification is performed with the same model. The extension to an arbitrary model made herein is a strong robustness statement: No matter which of the examined models or optimization techniques is chosen, the runtime performance of the special purpose model checker is always better than the best choice that can be made by an experienced user of the NuSMV or SPiN model checker. This observation might already have been drawn from Table 5.4 that depicts the superior runtime complexity of the special purpose model checker compared to the NuSMV and the SPiN model checker. However, for now it has only been obvious that this would hold on large CMDB instances. A polynomial or exponential algorithm might still outperform a linear algorithm on small CMDB instances if the constant scaling factor of the linear algorithm is high enough.

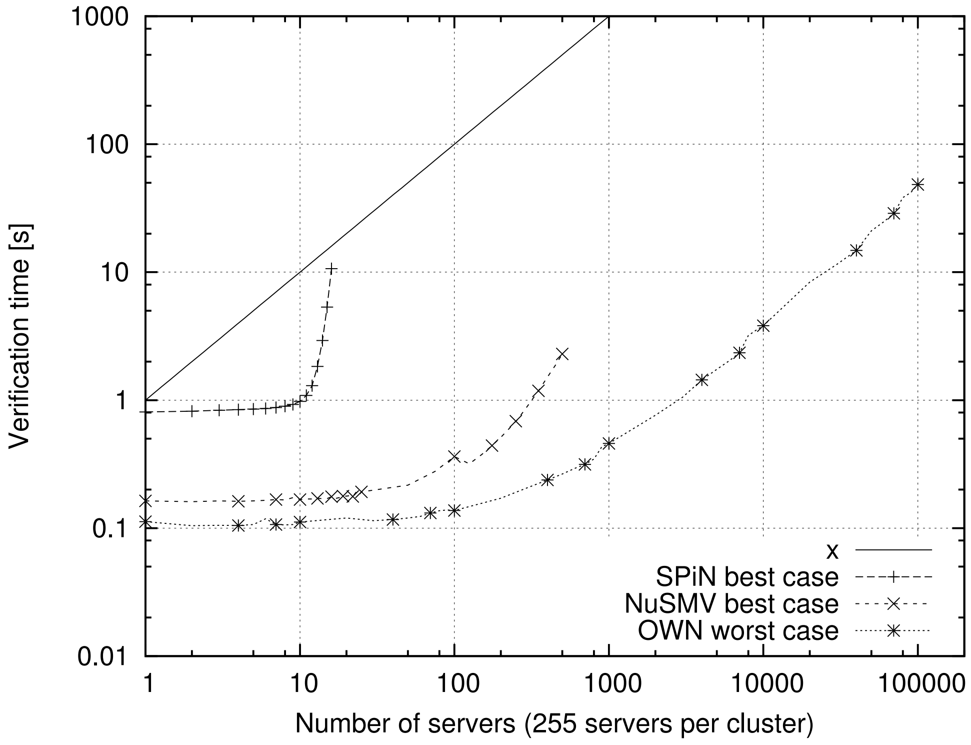


Figure 5.15: Worst- vs. best-case verification time of the various model checkers for the *SHTp* workload.

For the sake of brevity we only discuss the best- and worst-case performance analysis for the

SHTp workload herein. The benchmarks of the other workloads yield similar results and can be found in Figures B.12a-B.14 in Section B.3 of Appendix B. Figure 5.15 depicts the worst-case verification time of the special purpose model checker and the best-case verification time of the NuSMV and the SPiN model checker for the *SHTp* workload. The best- and worst-case have been determined among all models that support the *SHTp* workload (Model$_1$, Model$_2$, and Model$_4$) and all model checker specific optimization techniques. Notice that measurements are only considered for CMDB sizes that can be solved independent of the configuration of a benchmark, i.e., independent of the optimization technique chosen for every model checker. Thus, we ensure that the analysis only considers problem sizes that are solvable by every model checker using any of its optimization techniques. In Figure 5.15 it can be observed that, even in the worst-case, the special purpose model checker outperforms NuSMV and SPiN for the *SHTp* workload. The same holds for all other workloads (see graphs in Section B.3 of Appendix B).

5.5 Related Work

This section provides an overview of work related to IT change verification. Section 5.5.1 discusses related work in the context of IT change verification. After that, we broaden the scope and provide an overview of conflict detection in the configuration of security critical software systems in Section 5.5.2. In Section 5.5.3 we discuss policy conflict detection in the area of systems management. Finally, Section 5.5.4 discusses related work that aims to improve the reliability of systems and Change Management in general.

5.5.1 IT Change Verification

The solution presented in [50] is our preliminary work on IT change verification. In this work we propose an object-oriented reasoning algorithm to detect conflicting IT changes. Different to the algorithms discussed herein, the algorithm's runtime is factorial in the number of conflicting changes but allows for a more expressive set of effects and predicates. Thus, the algorithm has very limited scalability and cannot be applied to verify realistically sized Configuration Management Databases.

In a subsequent work [51] we present the preliminary ideas of the extended partial-order reduction approach and the efficient verification algorithm presented herein. However, the work lacks the formal proves and theory of extended partial-order reduction and does not evaluate extended partial-order reduction against the NuSMV and the SPiN model checker. To the best of our knowledge, these investigations are the only ones to address the problem of IT change verification in the context of IT Service Management.

5.5.2 Security Related Conflict Detection

There has been strong interest in the detection of conflicts or anomalies in the configuration of security critical software systems. For example, Al-Shaer et al. [5, 6, 7] propose an algorithm for the automated detection of conflicts in firewall policies. The notion of conflicts in these investigations is restricted to specific anomalies in the configuration of firewall policies. Different to the work presented herein, conflict detection is performed over a static set of policies while our solution takes into account the dynamic influence of changes on the configuration of the infrastructure. In addition to that, our solution detects conflicts based on the preconditions and effects of change activities while Al-Shaer et al. [5, 6, 7] detect conflicts from a previously

given, fixed set of firewall policies.
Investigations that aim to detect similar conflicts are, for example, Yuan et al. [100], Gligor et al. [43], Jajodia et al. [57] and Ferraresi et al. [40]. Because IT change verification focuses on the detection of non-security related conflicts in the systems management domain, specific solutions to detect conflicts in firewall configurations do not provide the logical expressiveness and the dynamics necessary to describe a broader set of IT changes.

Hu et al. [55, 56] apply model checking to verify access control models, which restrict the access of users to resources. The authors use the NuSMV model checker to verify the properties of access control models. Thus, the work suffers from similar scalability issues as observed in our analysis.

The authors of [19, 93] implement an algorithm to detect policy conflicts that incorporate authorizations (i.e., constraints that allow or permit an action) and obligations (i.e., constraints that require some action to be performed). The algorithm makes use of a theorem prover and subsumption reasoning [92]. The strong focus of the solution on policies only involving authorizations and obligations prevents its application to IT change conflict detection.

Another application domain for which conflict detection has been discussed is the domain of call control in telecommunication systems [17]. The authors address the detection of conflicts in a telecommunications domain. For example, two call forwarding policies that match for the same call but specify to forward the call to different numbers.

Common to all these solutions is that they have been specifically tailored for an application domain and its specific conflicts. It is important to mention that static policy conflict detection can be achieved by solely analyzing the parameters of policies. In turn, this is not the case for IT change conflict detection, which has to take into account the effects of changes on the configuration and the satisfiability of safety constraints. Due to the solutions proximity to security-related conflict detection, we will not further discuss them herein and focus on work that aims to detect conflicts among different policies in the broader area of systems management.

5.5.3 Conflict Detection for Systems Management Policies

Early work to detect conflicts in systems management policies was done by Lupu et al. [68]. The authors identify a set of policy conflicts and provide algorithms for their detection. Different to our approach, conflicts cannot be inferred from the logical specification of preconditions and effects of change activities, but conflict detection has to be explicitly specified for every policy. In addition to that, the methodology introduced does not take into account the dynamic aspects of change activities, i.e., that the state of the configuration changes over time by the application of IT change operations.

Agrawal et al. [4] discuss conflict detection for systems management policies as well. In their work a pair of policies conflicts if they are executable at the same time but issue different directives that cannot be achieved simultaneously. This notion of conflict differs significantly from our notion of conflicting change activities. IT change verification allows change activities to have contradicting effects as long as the safety constraints remain satisfied. The validation of policies specific for SAN systems is also discussed by the same author in [3].

Samak et al. [86] address conflict detection for policies that define the treatment of traffic flows on different networked systems. The work focuses on conflicts in the QoS configuration of network flows caused by the configuration of QoS parameters such as drop method, queue sizes, or bandwidth allocation. Consequently, this work specifically aims to detect QoS conflicts for network flows and cannot be used to detect conflicts among a broader range of IT change activities and safety constraints. However, the change verification logic enables the verification of configuration parameters across several networked Configuration Items, e.g., to determine whether queue sizes of several networked systems adhere to minimal size requirements.

Different to Lupu et al. [68], Dunlop et al. [37] propose a solution that supports the detection of policy conflicts at runtime. Different to our approach, their solution relies on capturing conflict profiles first to determine conflicts. Thus, conflicts cannot be inferred from the specification of preconditions and effects. Furthermore, the solution targets a restricted set of known policy conflicts for which the authors propose efficient algorithms.

The authors of [13, 23, 24, 25] make use of the Event Calculus [63] to formalize policy specifications and to detect conflicts among policies for differentiated services (DiffServ) networks. The solution addresses a fixed-set of previously known policy conflicts for DiffServ networks. Compared to the change verification logic, their logic supports a notion of time and variable/function symbols. Thus, their language is less restrictive than ours. However, this comes at increased costs. Charalambides et al. [24] indicate polynomial scalability in the number of policies. Similar to Dunlop et al. [37] and different to our approach, the solution cannot automatically infer conflicts from the specification of preconditions and effects of changes. Instead, rules need to be written beforehand to detect each conflict. Thus, conflicts need to be known in advance.

Most related to our work are the investigations by Kikuchi et al. [61] and Radu et al. [83]. Both authors investigate the use of model checkers to detect policy conflicts in autonomic computing systems. The authors distinguish three different concepts: (1) Operational rules that are comparable to conditionally executable IT change activities, (2) constraints that are comparable to safety constraints, and (3) final state conditions that need to hold in some future state caused by the execution of operational rules. Thus, their execution model is comparable to our notion of IT change verification while additional support for a final state constraint is provided. Similar to the evaluation presented herein, the authors apply the SPiN model checker for verification and report polynomial runtime complexity for their case study, which adds and removes servers (operational rules) from a pool of servers while guaranteeing transaction throughput goal policies and constraints on the number of sparse servers. The authors conclude that, for the verification to be feasible, problem instances need to be small enough to avoid the state-space explosion problem. The work presented herein extends their work by solving the scalability issues on large Configuration Management Databases.

The challenge to develop scalable verification algorithms for the verification of distributed systems and configurations has been noted by Calinescu et al. [22] as well.

5.5.4 Improving Reliability of Systems and Change Management

Similar to IT change verification, prior work on systems management aims to prevent the failure of IT systems as well. Automation, e.g., the Smartfrog framework [44], aims to prevent failures by the automation of application configuration and deployment. Automation makes sense for frequently applied standard changes. However, Amazon's network change is a seldomly executed, non-standard change, which is difficult to automate by deployment tools.

Practical validation, e.g., Tjang et al. [90], moves software systems from a production environment to an isolated validation environment where changes are conducted and validated. If the validation succeeds, the system is moved back to the production site. Similar to our approach, validation requires the specification of safety constraints or test scenarios. Practical validation is difficult to apply to network changes (like in Amazon's case) because a network cannot be easily moved between a production and test environment and overloads might still occur even with network virtualization in place. Instead, logical verification reasons about changes on a logical level without the danger inherent to the actual execution of change activities.

Oliveira et al. [79] propose Barricade, a framework for the mistake aware management of systems that issues and lifts blocks from Configuration Items based on the prediction of tasks and operator mistakes. The main difference is that - due to its prediction model - Barricade cannot prevent false-positives and negatives when issuing blocks. Different to Barricade, we assume complete knowledge about the tasks/changes currently being performed and derive conflicts from their logical description.

Besides the detection of conflicts, there have been other relevant research efforts to improve IT service management and steps of the Change Management process.

For example, Bartolini et al. [14] address the simulation and optimization of the IT incident management process to improve the handling of tickets in a service organization.

Similar to our work, the application of risk management to IT Change Management, among others Sauvé et al. [87] and Wickboldt et al. [97], aims to reduce the risk associated with change operations. Our work profits from risk assessment because it provides important clues about critical change activities worthy to be checked by verification.

Of popular interest is the change scheduling problem, which has been examined in the shape of different optimization problems. Among others, Setzer et al. [88] aim to reduce the risk that comes with the execution/scheduling of change activities.

Another active research area that aims to make the execution of changes more reliable has been that of change planning. Among others, the authors of [45, 53, 59] propose algorithms for the automated planning of IT change operations. Similar to verification, planning aims to make Change Management more reliable by means of automation but cannot guarantee the correctness of changes happening beyond the planning system. Different to automated planners, the approach proposed herein can also be applied to manual changes. Related work on IT change planning is discussed in more detail in Section 6.6 and Section 7.1.

Machado et al. [69, 70] improve Change Management by solutions for the rollback of failed change activities. Although a rollback was quickly initiated in Amazon's case [1], it failed to prevent the destabilization of the cluster and the outage [1].

5.6 Summary and Conclusions

This section summarizes the results of the evaluation experiments carried out in this chapter.

- **The special purpose model checker outperforms the NuSMV and the SPiN model checker in runtime complexity:**
  Independent of the model, workload, and optimization technique used, the special purpose model checker always outperforms NuSMV with linear instead of polynomial and SPiN with linear instead of exponential runtime complexity. See Section 5.4.2 and Table 5.4 for more details.

- **The special purpose model checker solves larger problem instances in time- and memory-constrained environments:**

In a time- and memory-constrained verification environment the special purpose model checker solves larger problem instances than the NuSMV and the SPiN model checker independent of the optimization technique or model used. Consequently, even if an inexperienced change manager chooses the worst model and optimization technique for the special purpose model checker, it still solves larger instances than the best choice that can be made for the NuSMV or SPiN model checker. See Section 5.4.3 for details.

- **The special purpose model checker is more robust than NuSMV and SPiN:** A bad choice made in the model and optimization technique of the special purpose model checker causes a runtime increase linear in the size of the CMDB compared to the best-case choice. In turn, the maximum penalty increases polynomially (NuSMV) or exponentially (SPiN) when a bad model and optimization technique are chosen for both general purpose model checkers. See Section 5.4.4 for details.
- **Worst-case runtime performance of the special purpose model checker better than best-case performance of NuSMV and SPiN:** Even if an inexperienced change manager chooses the worst-case combination of infrastructure model and optimization technique for the special purpose model checker, this choice is still faster than the best-case choice that can be made for the NuSMV or the SPiN model checker. See Section 5.4.5 for details.

Finally, we conclude that the special purpose model checker shows superior performance over NuSMV and SPiN when it comes to verify the different workloads to have caused Amazon's network outage. The special purpose model checker was proven to be superior in runtime complexity, practical runtime performance, robustness in respect to infrastructure models and optimization techniques, and scalability to large problem instances for various workloads of change activities to have caused Amazon's data center outage.

CHAPTER 6

A Comparison of Automated Planners for IT Change Planning

In Chapters 2 through 5 we discussed the efficient verification of atomic IT change activities, which can be used to increase the reliability of Change Management by automatically detecting conflicts among pending atomic change activities and hosting safety constraints. However, before atomic change activities can be verified, IT change plans need to be generated based on *Request for Changes* (RFCs), which specify on an abstract level the goal to be achieved by a major change operation. Similar to change verification, the automated generation of IT change plans aims to make Change Management more reliable by using automated and logically correct algorithms to improve an important step of the Change Management process. The generation of IT change plans has been subject to many investigations [32, 33, 46, 48, 53, 59, 71, 91]. Some investigations [32, 33, 48, 59] on this topic propose domain specific planning approaches that have not been previously discussed in the automated planning community [41] as general purpose planning algorithms. Others [46, 71, 91] use general purpose planning algorithms and sometimes add slight modifications to them to make them more applicable to IT change planning. Despite these investigations, it remains unknown as to how existing planning algorithms compare to each other when applied to IT change planning. In particular, the following two questions remain unanswered:

- How well do the different algorithms scale with the number of Configuration Items (software and hardware IT infrastructure components) in the Configuration Management Database to plan on?
- How difficult is it for a change manager to specify IT changes and to write efficient search control for a planner, which is crucial to enable planning on realistically sized CMDBs? Which way to specify search control most naturally matches to the domain of IT change planning?

To investigate which planners are most suitable for IT change planning in respect to scalability and usability, we compare four different domain independent automated planners (Graphlan

[18], Prodigy [94], TLPlan [10], and SHOP2 [76]) that cover a wide range of planning paradigms in a case study. The case study asks each planner to generate a high-level deployment plan of a three-tier application using an Infrastructure as a Service (IaaS) cloud. We examine the planners scalability when it comes to plan on large *Configuration Management Databases* (CMDBs) comprising thousands of *Configuration Items* (CIs) under a metric that measures the percentage of available resources in the Configuration Management Database. In addition to that, we discuss how well the different planners can be used by a logic- and algorithm-agnostic IT change manager, in particular when it comes to specify search control knowledge to guide each planner. To the best of our knowledge, this is the first work to examine different Artificial Intelligence planning algorithms [41] in respect to their applicability to IT change planning.

We find that Hierarchical Task Network (HTN) planners are the ones to offer the best performance as they scale to much larger CMDB sizes (up to 20,000 CIs) compared to the other algorithms ($\leq$ 1000 CIs) in our case study. This extends previous work [46, 91] that only argues in favor of Hierarchical Task Network planning due to its natural fit to change planning but does not show its performance benefits on large CMDBs. Similar to previous work [46, 91], we claim that HTN algorithms naturally match to IT change planning. However, this is the first work to support this claim by comparing alternatives to specify search control (among them, *linear temporal logic* (LTL) [10] and algorithmic specific control rules [94]) in a change planning case study with the goal to gain experience about their usability for change planning.

The remainder of this chapter is organized as follows. Section 6.1 sets the basic terminology and introduces the case study. Section 6.2 evaluates planning through planning graph analysis followed by an analysis of a forward chaining planner using temporal control knowledge in Section 6.3. Section 6.4 addresses planning through means-end analysis. Hierarchical Task Network planning is evaluated in Section 6.5. We discuss related work in Section 6.6. Finally, Section 6.7 summarizes our findings and concludes the chapter.[1]

6.1 Automated Planning and Case Study

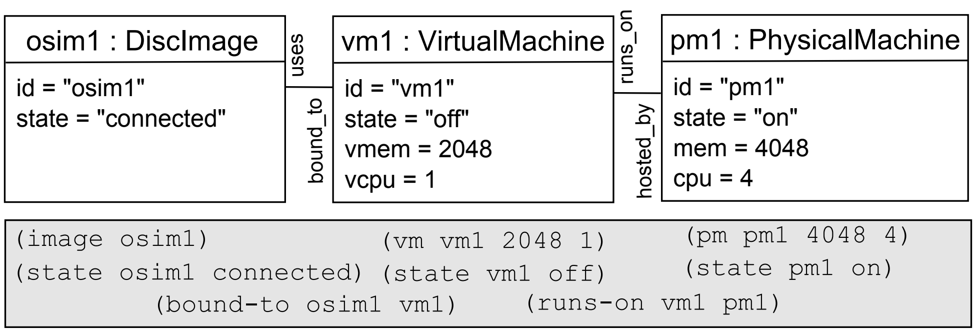
6.1.1 Automated Planning

The Problem of Plan Generation

Automated planning [41] has been an active research area within the Artificial Intelligence community for decades. The problem of plan generation (adapted to the IT change planning domain) is as follows. Given the current configuration of a data center (i.e., the current state of the hardware and software assets), a description of the goal to achieve, a description of the preconditions and effects of atomic IT change activities/planning actions, find a set of atomic IT change activities and a partial (sometimes total) order among them that achieve the goal when executed in any topological order of the partial-order. We use actions and atomic IT change activities synonymously, depending on whether we want to highlight the IT change planning view (atomic change activities) or the planning view (actions). Among other criteria, planners can be categorized in two distinct categories:

- **Domain-specific planners** are made, tuned, and tested for a specific domain and constraints. They do not perform well or at all in a domain they were not engineered for.

[1]Parts of this chapter previously appeared in [49].



```
(image osim1)              (vm vm1 2048 1)     (pm pm1 4048 4)
(state osim1 connected) (state vm1 off)     (state pm1 on)
         (bound-to osim1 vm1)    (runs-on vm1 pm1)
```

Figure 6.1: Object-oriented model instance of a Configuration Management Database and isomorphic set of ground atoms used by automated planners.

```
(:action start-vm
 :parameters  (?vmid ?pmid)
 :precondition  (and
  (vm ?vmid ?vmem ?vcpu) (state ?vmid off)            atoms of
  (runs-on ?vmid ?pmid) (state ?pmid on)              precondition
  (bound-to ?osid ?vmid) (state ?osid connected)
 )
 :effect (and                                         atoms added
  (not (state ?vmid off)) (state ?vmid on)            and deleted
 )
)
```

Figure 6.2: Description of an atomic change activity/ planning action to start a virtual machine written in the *Planning Domain Definition Language* (PDDL).

Most previous investigations on IT change planning [32, 33, 35, 46, 48, 59] fall into that category. Comparing these planners is difficult due to several reasons: (1) They are engineered for specific cases and have different expressiveness. Thus, finding a common case study that works with all of them and does not penalize a planner is very difficult. (2) Their source code/implementations are not publicly available.

- **Domain-independent planners** are generic planners that can be used for many planning domains. They are not engineered for a specific domain, but much work was invested [41] to optimize the performance of the algorithms over a variety of different domains. Domain-independent planners take a definition of the preconditions and effects of actions for planning. Thus, when atomic IT change activities are matched to planning actions, domain independent planners can be applied to IT change planning. Different to domain-specific IT change planners, domain-independent planners are easily comparable because their logical expressiveness is much closer and they are publicly available.

All planners examined herein (except Graphlan [18]) offer means to describe search control, which is specific to the planning domain and guides the planners search. However, the planners differ significantly on how easily search control can be written by a change manager.

Transformation of Object-oriented CMDBs

Traditionally object-oriented models have been used to describe the current state of infrastructure and software hosted in a data center, e.g., in commercial CMDBs [60] or in the *Common*

Information Model (CIM) [21]. Automated planners [41] do not plan on object-oriented models, but the state of the world is described in a function-free first-order language. Such a language comprises the following concepts:

- **Constant symbols**: Constants are used to describe the configuration of a data center. For example, there are constants that describe unique identifiers of Configuration Items, e.g., `vm` for a virtual machine or `pm` for a physical machine. Furthermore, there are constants to describe states of Configuration Items, e.g., `on`, `off`, `installed`, or Integer constants to describe resources available to Configuration Items.

- **Variables**: Variables can hold the values of constant symbols. If a variable is assigned the value of a constant, it is called bound, otherwise unbound. For an unbound variable x we write `?x` not to confuse with `x`, which describes constant x.

- **Atoms**: Atoms are n-ary predicates whose variables are either bound or not. For example, atom `state` is a 2-ary predicate that describes the current state of a Configuration Item. An atom is called ground if all of its variables are bound to constants. For example, `(state vm on)` is a ground instance of atom `state` describing the fact that `vm` is currently in state `on`.

To be used by automated planners, the information stored in an object-oriented CMDB needs to be translated to a set of ground atoms. Figure 6.1 depicts a part of the object-oriented CMDB and the analogous ground atoms view used by all planners. To transform the object-oriented CMDB, the properties of an object (i.e., Configuration Item) can be described as the bound variables of a ground atom. For example, ground atoms `(vm vm1 2048 1)` and `(state vm1 off)` describe all attributes of Configuration Item vm_1 in Figure 6.1. References are translated to binary predicates. For example `(runs-on vm1 pm1)` describes that `vm1` is hosted by `pm1` using a reference between `vm1` and `pm2`. Maghraoui et al. [71] have previously described this transformation process in more detail.

Actions and IT Change Activities:

An action is an atom together with a precondition and effects. Precondition and effects are described as a set of partially ground atoms. To apply an action, all of its unbound variables need to be bound such that all atoms in the precondition become grounded and hold in the current state of the knowledge base. At each point in time there might be multiple consistent bindings among all variables of an action to instantiate it. For example, consider the atomic change activity `start-vm` in Figure 6.2. The action has two variables / parameters `?vmid` and `?pmid` that need to be bound in order to apply the action (see Line 2 in Figure 6.2). If we choose `?vmid:=vm1`, `?pmid:=pm1`, and `?osid:=osim1`, then `start-vm(vm1,pm1)` is an applicable action in the state depicted in Figure 6.1 because all its ground precondition atoms in Lines 4-6 hold in this state under the substitution. The effects of the action are the deletion of ground atom `(state vm1 off)` from the knowledge base and the addition of `(state vm1 on)` to the knowledge base (see Line 9).

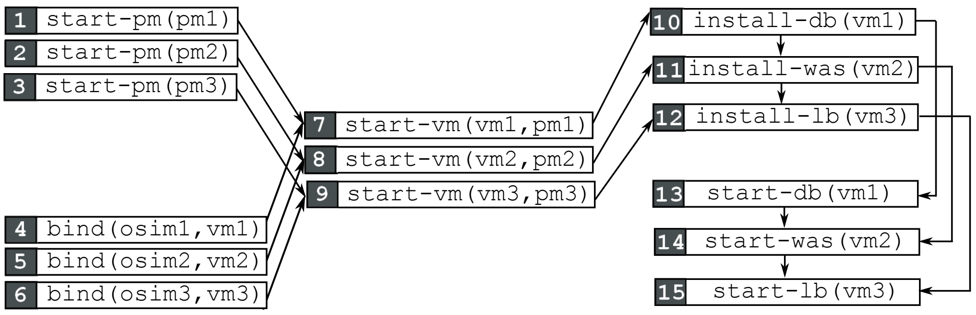


Figure 6.3: Ideal partial-order plan solving the planning case study.

6.1.2 Planning Case Study

Three-tier Application Deployment Case Study

The planning problem to solve by all planners is the generation[1] of a high-level deployment plan of a three-tier business application (database, application server, and load balancer) making use of an IaaS cloud. Figure 6.3 depicts a partial-order plan that solves this problem on a particular instance of a CMDB. It consists of atomic change activities to turn on physical machines (act_{1-3}) and virtual machines (act_{7-9}), to bind OS images to the virtual machines (act_{4-6}), and to install (act_{10-12}) and start (act_{13-15}) software. Although this plan omits more complex configuration details inherent to applications and networks, its generation already causes serious trouble for several planners when CMDBs reach a few hundred or thousand Configuration Items. All planners need to take the following constraints into account:

- Disc images can only be bound to a single virtual machine.
- Virtual machines need to run exclusively on a physical machine.
- Database, application server, and load balancer need to run exclusively on a virtual machine.
- A service that depends on another software, e.g., application server on database, cannot be installed/started without the other one already being installed/running due to configuration and runtime constraints.

Although the constraints are quite specific, what matters is not the concrete constraint, but the search effort a planner needs to put into finding the proper Configuration Items, i.e., constants, that satisfy the precondition of an action. Consequently, the results presented herein are comparable with different constraints for which up to one property of a Configuration Item needs to be examined to find a match.

Shape of CMDB Used for Evaluation

The algorithms are evaluated on differently sized and shaped configurations of the Configuration Management Databases. A configuration always consists to $\frac{1}{4th}$ of physical machines, virtual machines, OS images, and services no matter how large it is. Services consist to $\frac{1}{3rd}$ of database, application server, and load balancer services. For example, a CMDB comprising

[1]All performance measurements are conducted on an Intel Xeon x86 CPU with 2.8Ghz and a maximum of 1024 MB RAM available to each planner. Planners run single threaded.

8,000 Configuration Items comprises 2,000 physical machines, virtual machines, OS images, and each 666 instances of database, application server, and load balancer services. The goal to achieve is to create a valid three-tier application deployment plan (see Figure 6.3) in this environment.

During the experiments we vary the *selectivity* (sel) of Configuration Items in the CMDB. *Selectivity* describes the fraction of Configuration Items (physical machines, virtual machines, OS images, services) that can be used to instantiate an atomic change activity/planning action in such a way that its precondition is satisfied. For example, if 100 out of 300 OS images have not yet been bound to a virtual machine, then OS images have a selectivity of 33% because 33% of all OS images qualify for an atomic change activity that requires an OS image to be unbound in order to bind it to a virtual machine. Thus, selectivity is a metric describing the fraction of Configuration Items that satisfy the precondition of atomic change activities that have to choose among resources.

Representativeness of Case Study

The case study presented herein is realistic and representative for IT change planning because:

- The case study requires the planners to plan for several atomic change activities that need to be properly instantiated with resources, i.e., Configuration Items of the CMDB, such that their preconditions are satisfied when they are about to be executed in the final plan. For example, the case study comprises atomic change activities that bind virtual machines to physical machines (the planner needs to decide which physical machine to chose) or atomic change activities to place services on virtual machines (the planner needs to decide on which virtual machine to place the service). These types of change activities are typical for deployment scenarios and require a planner to efficiently select the resources according to constraints.

- The case study comprises state-related change activities among dependent Configuration Items [46, 48], e.g., the requirement for a database server to be in state *on* to start an application server. State-related dependencies are typical for many distributed systems when it comes to deployment, undeployment, migration, and operational changes.

- The decomposition of abstract change activities into finer-grained change activities [33, 35, 46, 91] has been previously addressed in the literature and has been noted to be typical for IT change planning. The deployment case study can be expressed as the abstract task to deploy a three-tier application, which requires the further decomposition of the abstract change activity into finer-grained change activities. Thus, the deployment case study is suitable to assess how well the planners cope with the decomposition concept often present in IT change planning.

Thus, we can expect the results presented herein to hold for similar planning problems that are dominated by atomic change activities that need to select resources over large CMDBs and that need to take into account state-related dependencies.

6.2 Planning Through Planning Graphs

6.2.1 The Graphlan Algorithm

The *Graphlan* algorithm [18] is a forward chaining search algorithm, i.e., it starts searching from the initial state towards the goal state by applying actions. Graphlan takes as input a description of each action/atomic change activity to consider for planning (see Figure 6.2 for an example), a list of ground atom instances holding in the initial state, and a list of ground atom instances that need to hold in the goal state. Graphlan creates a planning graph that consists of several levels, so called proposition levels (comprising ground atoms) and action levels (comprising instances of atomic change activities/actions). The i-th proposition level comprises all ground atoms that can theoretically be obtained from the initial state by applying i or less sets of parallel atomic change activities/actions. Similarly, action level i comprises all atomic change activities/actions that can theoretically be applied based on the ground atoms of the preceding proposition level. Thus, action levels are generated from proposition levels and proposition levels from action levels.

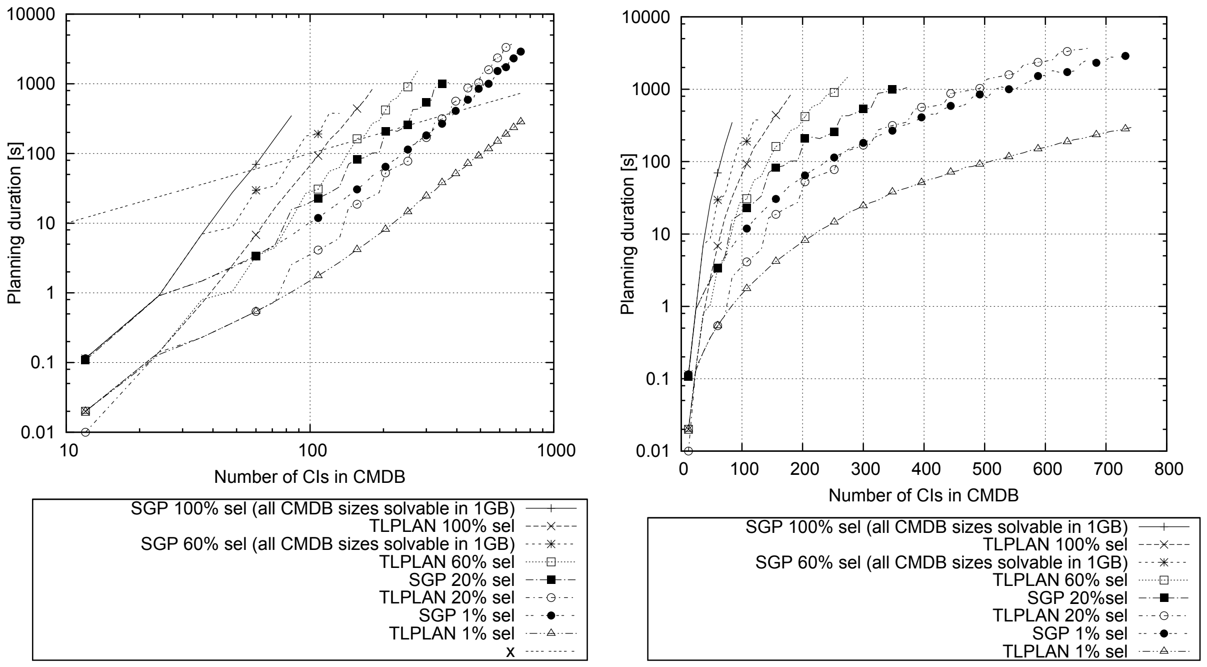
The algorithm starts with the initial proposition level that matches the initial state of the planning problem (specified as a set of ground atoms). To generate the action level, Graphlan creates all possible ground instances of all actions that are applicable based on all ground atoms in the proposition level. Thus, the action level comprises all possible instances of atomic change activities/actions that can be applied given all the atoms of the proposition level. This means that Graphlan instantiates every applicable version of every atomic change activity in each action level. The proposition level, which follows the action level, comprises all atoms of the effects of the atomic change activities/planning actions of the preceding action level (to take into account the execution of each change activity) and all atoms that belong to the proposition level that was preceding the action level (to take into account that no action of the action level might be executed). Thus, the i-th proposition level comprises all atoms that might theoretically be obtained from the initial state by the execution of i sets of atomic change activities. Graphlan creates proposition and action levels until a proposition level is obtained that satisfies the goal formula. Graphlan then starts a backward search to verify whether a valid plan exists. To create a plan, the search picks from each action level atomic change activities/actions that are free of conflicts. Actions from the same action level can be executed in parallel. For a more detailed introduction to the Graphlan planning algorithm see [18].

6.2.2 Evaluation

Usability

With only 200 *lines of code* (LOCs) we found Graphlan's action descriptions to be easily writable because the change manager only needs to think about the preconditions and effects of atomic change activities as shown in Figure 6.2. Furthermore, we found the idea of atoms, which basically are tuples being added and deleted from a tuple store, to describe the current state of the world, intuitive. However, as change managers are not trained in logics, they most likely feel different about the logical representation. We do not consider this to be a big hurdle in the adoption of automated planning approaches because the object-oriented and predicate-based representation are isomorphic. This enables *Domain Specific Languages* (DSL) to describe IT changes [46] that are closer to a change manager's domain.

Performance



(a) Log-log graph **(b)** Semi-log graph

Figure 6.4: Planning performance of Sensory Graphlan (SGP) and TLPlan (with LTL search control) depending on the size of the CMDB and selectivity.

Figure 6.4 depicts the log-log and semi-log graphs of the planning time of *Sensory Graphlan* (SGP) [95], a LISP extension of Blum's original algorithm [18], when solving the planning case study. SGP is preferred over Blum's original implementation because in our experiments it solved larger problem instances when being memory constrained. From the log-log graph (Figure 6.4a) it can be observed that all curves are straight lines with a slope larger than one. Thus, runtime complexity is polynomial in the number of Configuration Items. This is also confirmed by the corresponding semi-log graph in Figure 6.4b. In this graph, curves grow slower than a straight line indicating that runtime complexity is less than exponential.

For an upper bound of 100s on the planning duration and a selectivity of 100%, a domain cannot be larger than 60 Configuration Items, i.e., 15 physical machines, virtual machines, OS images, and services to remain solvable (see Figure 6.4). SGP runs out of memory for domains comprising more than 84 Configuration Items and 100% selectivity. However, if selectivity decreases to 1%, a planning domain comprising 240 Configuration Items becomes feasible within 100s. We conclude that the lower the selectivity and the smaller the CMDB, the faster SGP becomes and the more likely it is that SGP can solve the problem when constrained to 1024 MB of memory. When the selectivity decreases, there are less valid choices that can be made by the planner to instantiate actions. Thus, the number of actions instantiated in every action level decreases and less atoms are added to the subsequent proposition level. This is the main reason why Graphlan performs better if many resources are not available, i.e., when selectivity is small.

All in all, the Graphlan algorithm shows an unsatisfactory performance for our case study and can at best be used to solve small problem instances. This is the case because the algorithm applies all instances of all atomic change activities/planning actions, independently of their contribution to the goal, in a breadth-first manner at each action level. When the CMDB size or the selectivity increases, a larger number of Configuration Items qualify as valid bindings to instantiate actions inflating the action and proposition levels. Decreased selectivities and smaller CMDB sizes lead to less ways to instantiate an action and can make the difference between solvable and unsolvable problems.

6.3 Planning Through Forward Chaining and Temporal Control Knowledge

6.3.1 The TLPlan Algorithm

Similar to Graphlan, *TLPlan* [10] is a forward chaining planner, but it uses domain specific search control knowledge specified in a first-order version of linear temporal logic [11] to prune plans causing undesired sequences of intermediate configurations of the CMDB. TLPlan starts with the initial configuration of the CMDB and determines all applicable ground instances of actions/atomic change activities that can be applied to the current CMDB. Based on these actions all successor CMDBs are created. TLPlan then continues in a depth-first search with the exploration of a successor CMDB. However, TLPlan does only further explore a CMDB configuration if the atomic change activity/state last added to the plan does not violate the temporal logic control formula. Nevertheless, all successor worlds are created. TLPlan also offers the option to directly discard CMDB states violating the formula (causing less memory consumption), but we were unable, due to stability issues of the planner, to derive a working LTL formula for this early pruning option.

6.3.2 Evaluation

Usability

For TLPlan to solve the IT change planning case study, a change manager has to specify a formula in linear temporal logic to guide the search. Let's assume $act_i, i \in \{1, \ldots, n\}$ are atomic IT change activities, such that the sequence of atomic change activities $\langle act_1, act_2, \ldots, act_n \rangle$ forms a plan to achieve the deployment of a three-tier application. Then, let $\langle s_0, \ldots, s_n \rangle$ be the sequence of corresponding configurations s_i of the CMDB induced by the execution of the plan (s_0 initial state, s_n valid goal state, $s_i, i \in \{1, \ldots, n\}$, the state after the execution of act_i).

A first-order linear temporal logic formula, which is evaluated on $\langle s_0, \ldots, s_n \rangle$, can be specified by the change manager to prevent undesired plans. Two of the four LTL operators [10] used in our control rules are $\Box\phi$[1] and $\bigcirc\phi$ (next ϕ operator)[2]. To prune portions of the search space, atomic change activities that do not contribute to the goal state need to be disallowed by a LTL formula. For example, it makes sense to only turn on virtual machines (action `start-vm`, Figure 6.2) that are not running and are meant to be on in the goal state. Figure 6.5 depicts the LTL formula of this constraint.

The formula (Figure 6.5) only permits sequences of intermediate CMDB configurations ($\Box$) such that among subsequent configurations ($\bigcirc$) only virtual machines are allowed to change their state from *off* to *on* (left side of implication) if the virtual machine is explicitly specified as *on* in the goal state (goal expression).

[1] $\Box\phi$ means that ϕ holds for the current and all future configurations (states) of the CMDB.

[2] $\bigcirc\phi$ means that ϕ holds in the subsequent configuration (state) of the CMDB.

```
(□  (∀ ?vmid, ∀ ?mem, ∀ ?cpu : (vm ?vmid ?mem ?cpu)
       (  (state ?vmid off) ∧
          (◯ (state ?vmid on))
       ) → (goal (state ?vmid on))
)   )
```

Figure 6.5: Example LTL formula to guide the search of the TLPlan planner.

There are several problems with LTL formulas as a mean to formalize search control for IT change planning:

- For some planning problems it can be impossible to determine whether an atomic change activity contributes to the goal state or not. This is the case if effects of an atomic change activity of a change plan are not visible in the goal state anymore. For example, assume a software is not installed in the initial state and supposed to be running in the goal state. A simple change plan would first install and then start the software. The effect of the first atomic change activity to install the software (effect: software is in state installed) does not match to the final state where the software is supposed to be in state running. Thus, as the effect of the install software change activity only contributes indirectly to the goal state (it enables the start action which achieves the goal state), we cannot tell that the install action needs to be part of a plan by simply looking at the logical description of the goal state as we did in the LTL formula in Figure 6.5.

- The LTL formula used in the change planning case study comprises 170 LOCs. A linear temporal logic formula of that size is too large and complex to be practically used and derived by an IT change manager.

- It is very unlikely that a change manager is willing and capable to cope with LTL and its subtle semantics.

- We found it very difficult to write LTL formulas over a sequence of intermediate CMDB states. Instead, given our experience, it seems more natural to specify LTL formulas over valid and invalid sequences of atomic change activities because intermediate states are more difficult to grasp than the constraints on sequences of atomic IT change activities.

Performance

The performance of TLPlan strongly depends on the usage of LTL search control.

Without LTL search control and a selectivity of 100%, TLPlan takes 10ms to solve the change planning case study for 12 Configuration Items and 833s for a CMDB comprising 24 CIs. The problem becomes unsolvable within 12h for larger CMDBs and 1 GB of RAM. Thus, TLPlan is impracticable to solve even small problems of the case study without LTL search control.

With LTL search control (see Figure 6.4) TLPlan performs better than SGP/Graphlan for every selectivity and every size of the CMDB. Different to SGP, which can only solve very small instances of the case study when constrained to 1 GB of main memory at selectivities between 40% and 100%, TLPlan does not show this limitation. Similar to SGP, TLPlan struggles to maintain performance with larger CMDBs. The time to derive a plan increases polynomially with the size of the CMDB and is slightly subdued by smaller selectivities because less successor CMDBs need to be managed by the planner. Different to Graphlan, TLPlan produces totally ordered plans instead of partially ordered plans. Notice that the performance improvement is achieved at the cost of complex LTL formulas that are difficult to specify.

6.4 Planning through Means-end Analysis

6.4.1 The Prodigy Algorithm

Different to the forward chaining approach pursued by Graphlan and TLPlan, *Prodigy* [94] searches backwards from the goal (backward chaining). Prodigy uses means-end analysis, i.e., it chooses a ground atom of the goal state that has not yet been achieved and attempts to instantiate an action in such a way that its effects produce that atom. The atoms in the action's precondition that do not yet hold become goal atoms that need to be subsequently achieved by applying further actions. For example, consider the ground goal atom `(state vm1 on)`. Action `start-vm(?vmid, ?pmid)` in Figure 6.2 can be partially instantiated (choose `?vmid:=vm1`) such that it produces that atom. If atoms in the precondition of the action should not yet hold, Prodigy keeps on working on these open goals. During planning Prodigy has several choices: (1) which goal to achieve first, (2) which action to use to achieve the goal and how to instantiate the action, e.g., how to instantiate variable `?pmid` in `start-vm(vm1, ?pmid)`, and (3) when to apply an action. To ensure that the planner makes good choices (in terms of state-space exploration), an IT change manager can specify *control rules* that tell the planner on how to make these decisions.

6.4.2 Evaluation

Usability

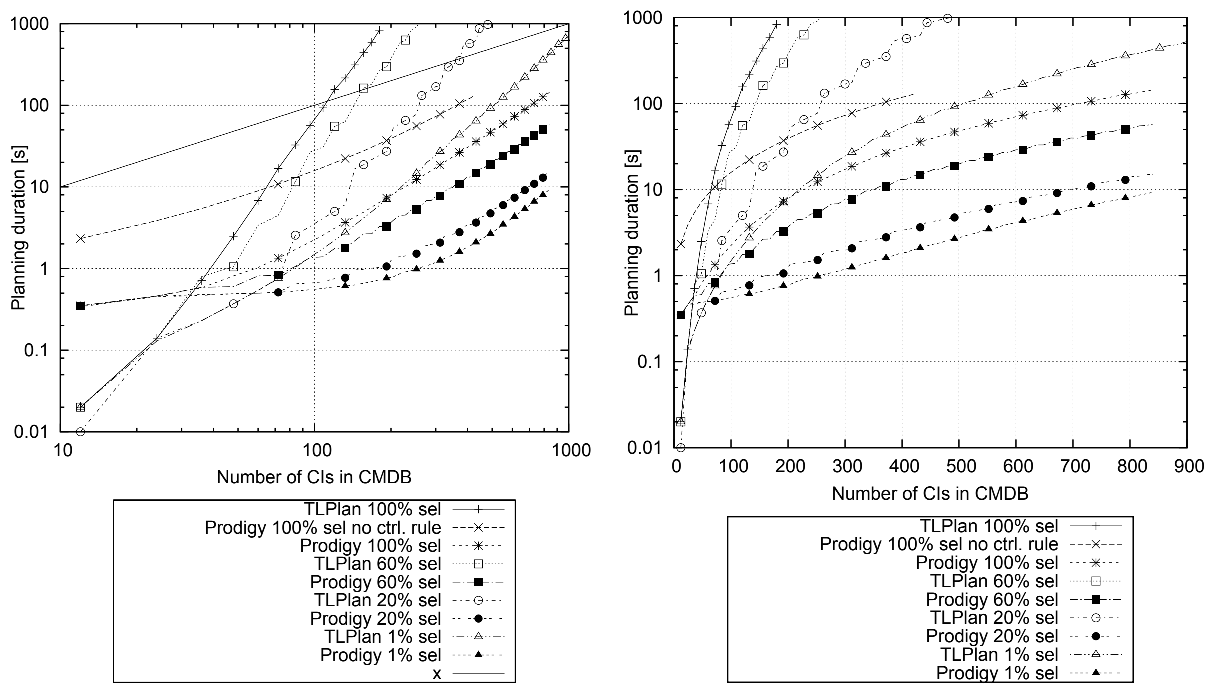
Prodigy uses the same[1] domain description as SGP and TLPlan for input. Thus, Prodigy's actions can be as easily engineered as Graphlan's and TLPlan's. For Prodigy to work efficiently, control rules need to be specified. Control rules are *if-then* rules. For example, one of four control rules used in the case study reads as follows. If the planner is working on the goal atom `(state ?vmid on)` and tries to achieve it by instantiating action `start-vm(?vmid,?pmid)` (see Figure 6.2 to comprehend why this achieves the goal), then variable `?pmid` needs to be bound to a physical machine that has no virtual machine running on it. The reason for this rule is that when Prodigy later plans for a subsequent action, this placement constraint is checked leading to costly backtracking if the wrong decision is made in the prior step.

Change managers need to look at the debug output and need to have a precise understanding of how the algorithm works to write efficient control rules. With search control so closely related to the algorithm, i.e., the need to examine the debug output of a planner, we conclude that this approach is of no practical use to a change manager. However, if actions, problems, and control rules have been carefully tuned, Prodigy's control rules offer effective speedup.

Performance

Figure 6.6 depicts the planning performance of Prodigy and TLPlan (with LTL search control) depending on the size of the CMDB, the use of control rules, and the selectivity.

[1] besides some syntactical differences



(a) Log-log graph (b) Semi-log graph

Figure 6.6: Planning performance of Prodigy (with control rules if not mentioned otherwise) depending on (1) size of the CMDB, (2) selectivity, and (3) use of control rules compared to TLPlan (with LTL search control).

Prodigy performs better with control rules. For example, within a time boundary of 100s and 100% selectivity, Prodigy can solve the case study on a CMDB comprising 350 Configuration Items without control rules compared to 700 Configuration Items with control rules. On small CMDB sizes, TLPlan outperforms Prodigy even when control rules are used. Beyond 24 - 36 Configuration Items (depending on the selectivity), Prodigy becomes faster than TLPlan. Thus, for realistic CMDB sizes Prodigy always outperforms TLPlan and SGP/Graphlan. Without control rules TLPlan manages to stay ahead of Prodigy on slightly larger CMDBs. For example, at 100% selectivity the CMDB must be larger than 60 Configuration Items for Prodigy (without control rules) to outperform TLPlan.

From the log-log graph in Figure 6.6a we can observe that the measurements of Prodigy converge towards a straight line with a slope larger than one with increasing CMDB size. Thus, Prodigy has polynomial runtime complexity for the case study. Even without control rules Prodigy performs significantly better than TLPlan (on larger CMDBs) for the deployment case study (350 Configuration Items vs. 100 Configuration Items within 100s at 100% selectivity) and thus better than Graphlan. Similar to TLPlan and SGP, we can observe better planning performance with decreasing selectivity. This is the case because for every not yet achieved goal atom that Prodigy decides to work on, it computes all ground instances of all actions that can achieve this atom before it continues planning with one instance. Thus, the smaller the CMDB and the smaller the selectivity, the less Configuration Items qualify to properly instantiate an atomic change activity. This decreases the computational effort to compute the bindings of applicable, atomic IT change activities.

The main performance improvement of Prodigy over TLPlan and SGP is due to the fact that Prodigy only considers the application of actions that contribute to a goal atom. As the atoms

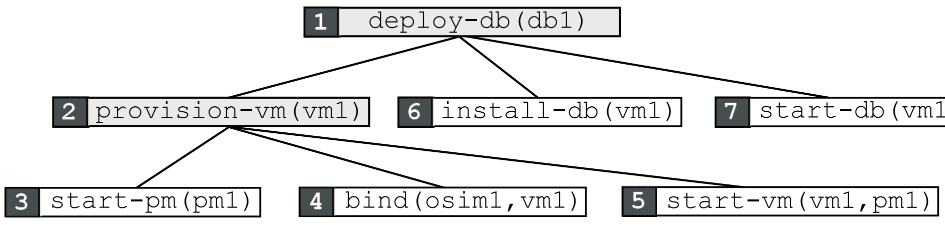


Figure 6.7: Simplified HTN decomposition tree created by an HTN algorithm for the deployment of a database.

of the goal state are already given, lots of choices on how to instantiate actions to produce these atoms can be pruned. Notice that the results presented herein differ to the results presented by Blum et al. [18]. That work shows that Graphlan outperforms Prodigy in two artificial planning domains (2-Rockets domain and Link-repeat domain). However, for the IT change planning problem, Prodigy outperforms Graphlan because it carefully chooses the actions that contribute to the goal state.

6.5 Hierarchical Task Network Planning (HTN)

6.5.1 The SHOP2 Algorithm

Hierarchical Task Network (HTN) planners [42] differ from the previously examined planners in a significant way: The goal to achieve is not specified as a set of ground atoms that need to be satisfied in the goal state, but as an abstract task, e.g., to deploy a three-tier application, for which the planner needs to derive a plan. Similar to the previous approaches, atomic IT change activities are described as actions that add and delete ground atoms to and from the knowledge base (see Figure 6.2 for an example). The domain specific search control is described by HTN *methods*. *Methods* decompose an abstract task, i.e., a high-level, abstract IT change activity, into finer-grained activities until atomic change activities are reached that can be directly mapped to the planning actions. Figure 6.7 depicts an exemplary decomposition tree created by an HTN planner for the abstract change activity to deploy a database on a specific CMDB configuration. To generate an IT change plan that solves the high-level change `deploy-db(db1)`, SHOP2 [76], a forward chaining partial-order HTN planner, searches for a method to decompose the abstract change activity. A method specified by a change manager instructs the planner to decompose act_1 by trying to achieve the finer-grained change activities `provision-vm(vm1)` (act_2), `install-db(vm1)` (act_6), and `start-db(vm1)` (act_7) in sequence (see Figure 6.7). Methods capture best practice problem solving strategies inherent to the domain. Planning is done in a depth-first search according to the order on the subtasks (see numbers of change activities in Figure 6.7). Notice that the decomposition stops at the leaf nodes of the decomposition tree that cannot be further decomposed and directly map to the planning actions used by the other algorithms as well. An HTN planner only tries to apply an action if it is directed to do so by the decomposition tree. For example, the planner will never try to start a virtual machine (act_5) before an image has been bound (act_4) because the decomposition tree directs the planner to do it in the opposite order. The plan returned by an HTN planner only comprises the leaf nodes of the decomposition tree. Notice that the leaf nodes match to the changes in Figure 6.3 and their order of exploration in the depth-first search tree is a topological sort of the partial-order plan in Figure 6.3.

6.5.2 Evaluation

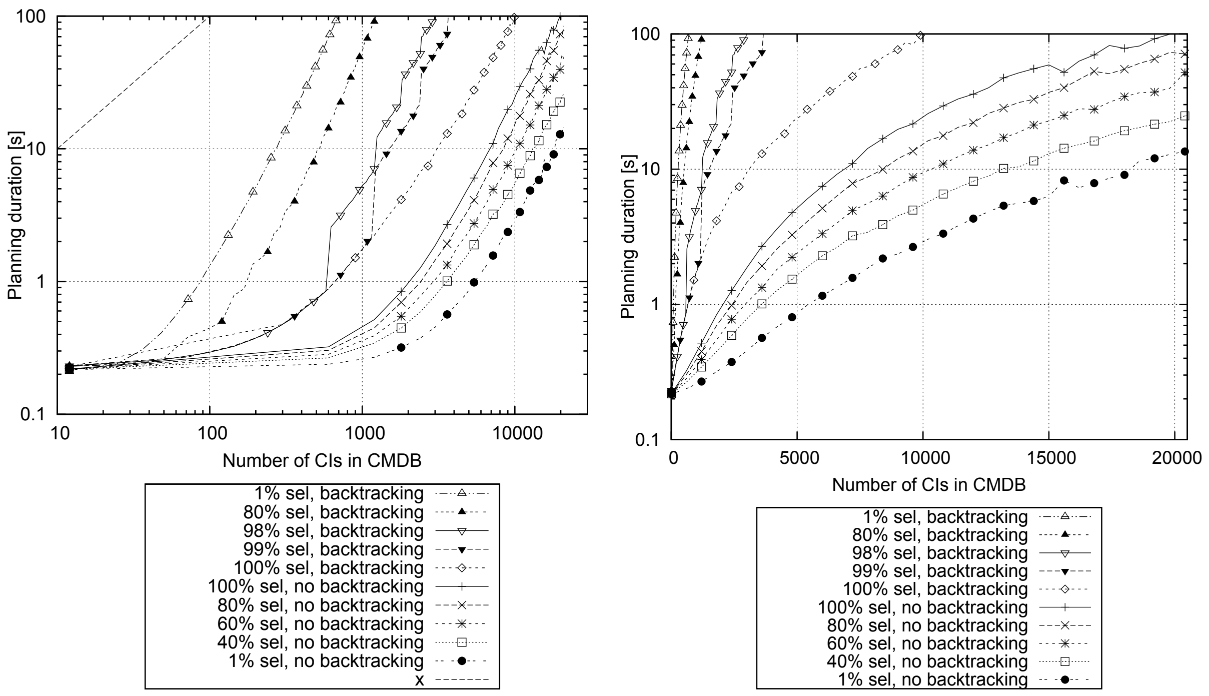
Usability

The SHOP2 domain description comprises around 700 LOCs, which is roughly 3.5-times the size of the other planning domains. In addition to the specification of the preconditions and effects of atomic IT change activities (needed for all other approaches as well, see Figure 6.2), decomposition rules need to be specified to tell the planner how to decompose abstract change activities into finer-grained change activities. Thus, more effort is needed to write and debug such a domain. Once engineered, an HTN domain has several advantages: (1) HTN domains are very reusable because newly added abstract change activities can make use of the logical specification of already existing change activities and the decomposition rules written for them. For example, an abstract change activity to deploy a three-tier application can, for instance, rely on the specification of another abstract change activity to deploy a database system (see Figure 6.7). (2) The idea of abstract task decomposition matches precisely to the IT change planning problem that, given an abstract Request for Change [65], asks the change manager to propose a plan of atomic change activities to implement that change. The change manager stays within his/her domain of thinking (decomposition of abstract IT change activities) without being distracted by LTL formulas or algorithm specific control rules. All in all, the specification of decomposition rules seems to be the easiest and most natural way for a change manager to describe domain specific search control knowledge. This conclusion is supported by Cordeiro et al. [33, 35] who argue in favor of reusable plan templates that can be implemented using HTN methods. Although the proximity of IT change planning to HTN planning has been noted before [46, 91], we argue in this work that alternatives to describe search control, such as linear temporal logic and algorithm specific control rules, are not a better solution to specify search control.

Performance

Figure 6.8 depicts the planning performance of SHOP2 for two different domain descriptions, one that has been carefully engineered to solve the problem without backtracking [31] and another one that solves the problem with backtracking. The difference between both domains lies in the way how constraints are enforced. Consider `bind(osim1,vm1)` (act_4) in Figure 6.7. When planning for the case study problem, a method is applied to decompose `provision-vm(vm1)` (act_2) into act_3, act_4, and act_5. This means that an appropriate OS image needs to be determined to instantiate act_4. In order for act_4 to be executable, its precondition checks whether the image is unbound. In the domain that does not prevent backtracking the planner binds parameter `?osid` in act_4 to an already bound image when decomposing act_2, causing the planner to backtrack over act_4 because its precondition is infeasible (see measurements involving backtracking in Figure 6.8). Instead, the constraint that the image needs to be unbound can be added to the method that decomposes act_2. This prevents backtracking because then the method only chooses an image that satisfies the precondition of act_4 as well. This yields the no backtracking measurements in Figure 6.8.

Similar to the other algorithms, the planning time increases polynomially with the size of the CMDB because all measurements converge towards a straight line with a slope larger than one in the log-log graph (see Figure 6.8a). However, the polynomial complexity is much more modest compared to the other planners for the domain that solves the problem without backtracking. Thus, problem sizes of up to 20,000 Configuration Items - 28 times the size limit of Prodigy and 333 times that of SGP - can be solved within 100s at 100% selectivity (worst-case). The



(a) Log-log graph **(b)** Semi-log graph

Figure 6.8: Planning performance of SHOP2 depending on the size of the CMDB, the selectivity, and the occurrence of backtracking.

performance is superior because when refining an abstract change activity, SHOP2 computes all bindings of variables in the subtasks but only applies one instance of an action or method during planning. Thus, SHOP2 does not inflate its search space by applying all unifications of an IT change activity as Graphlan and TLPlan do. In addition to that, the decomposition tree tells the planner precisely when to try an abstract or atomic IT change activity pruning large portions of the search space.

If backtracking is not avoided, planning takes significantly longer than without backtracking because backtracking is more costly than enforcing binding constraints early. In case of backtracking, the runtime increases when the selectivity decreases because the smaller the selectivity, the more frequent the planner has to backtrack as it more often commits to unsuitable Configuration Items to instantiate an action or method. In our benchmark we assumed the worst-case scenario in which SHOP2 has to backtrack over all unqualified resources (whose number is defined by the selectivity). If the planning domain without backtracking is used, the runtime is dominated by computing all ground instances of actions and methods, which is lower for smaller selectivities (exactly opposite to the backtracking case).

The observation that SHOP2 performs poorly for backtracking domains yields the question whether a carefully engineered Prodigy domain with control rules outperforms a poorly designed HTN domain that does not prevent backtracking?

This is indeed the case on large CMDBs. On small CMDBs a naively written SHOP2 domain (not avoiding backtracking) performs better than a carefully tuned Prodigy domain with control rules for the case study (see Figure 6.9). However, with increasing CMDB size Prodigy starts to outperform SHOP2 when SHOP2 makes use of backtracking. The CMDB size where Prodigy overtakes SHOP2 is reached the later the higher the selectivity (for 10% selectivity from 90 Configuration Items onwards, for 40% selectivity from 200 Configuration Items onwards,

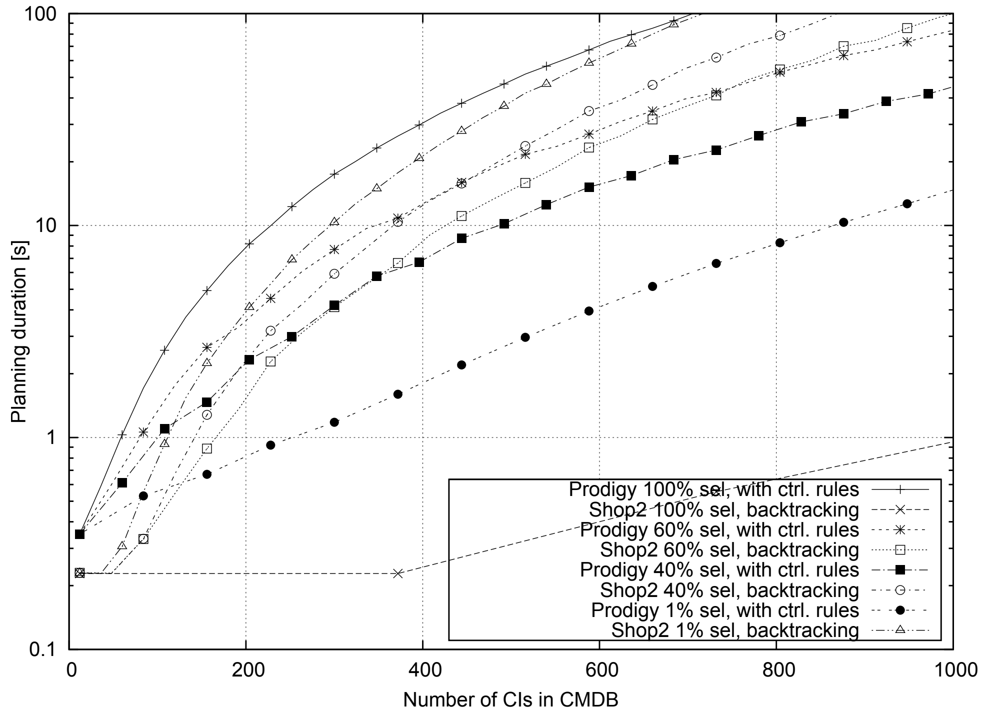


Figure 6.9: Planning performance of SHOP2 (backtracking domain) compared to Prodigy (with control rules).

for 60% selectivity from 750 Configuration Items onwards, see Figure 6.9) because the lower the selectivity, the more SHOP2 has to backtrack making it easier for Prodigy to catch up earlier. Prodigy fails to catch up with SHOP2 for 100% selectivity because backtracking does not appear for the SHOP2 domain.

6.6 Related Work

Several solutions [32, 33, 35, 46, 48, 53, 59, 71, 91] have been proposed for the (semi-)automated generation of IT change plans. All investigations propose domain specific algorithms for IT change planning but do not analyze existing general purpose planning algorithms [41] for IT change planning.

Keller et al. [59] propose *CHAMPS*, a system for the planning and scheduling of IT changes that enables a high degree of parallelism among IT changes. The authors propose a planning algorithm that is not based on automated planning algorithms [41]. A performance and usability comparison with other planning algorithms is out of the scope of that paper.

Maghraoui et al. [71] are the first to apply an automated planner, UCPOP [82], to generate IT change plans. Their highly customized version of the UCPOP algorithm is applied to a case study with the size of a few hundred resources. The experience we gained about UCPOP for our case study is that it is even slower than Graphlan without control knowledge.[1] Furthermore, control knowledge in UCPOP is extremely difficult to specify because it is very close to the algorithm. A comparison to other algorithms than UCPOP is out of the scope of Maghraoui's work.

[1]This is also confirmed by the UCPOP project homepage (http://www.cs.washington.edu/ai/ucpop.html), which recommends to use Grphlan instead of UCPOP due to its performance improvements.

Cordeiro et al. [33, 35] propose *ChangeLedge*, a system for the generation of change plans by capturing best practices in IT change design using change templates. Similar to CHAMPS [59], the proposed algorithm does not reason about the preconditions and effects of IT changes on a logical level. It remains unknown as to how the algorithm scales and how it compares to automated planners [41].

In another work, Cordeiro et al. [32] propose a runtime constraint aware solution for the automated decomposition of IT changes. The proposed algorithm captures the basic idea of preconditions and effects of IT changes as known from automated planning [41]. It remains open as to how their approach relates to automated planners in terms of performance and logical soundness/completeness.

In a previous work [46] we propose to apply a hybrid of HTN and state-based planning to solve the IT change planning problem. Similarly, Trastour et al. [91] propose a pure HTN algorithm. Both work do not compare the proposed HTN solutions to other already existing domain independent planners in respect to performance and usability as done in this work. The work herein extends these investigations by providing evidence that HTN planning algorithms are the fastest and most usable algorithms for IT change planning. In another previous work [48] we present a pure state-space planner for IT changes. Similar to [46], a comparison with general purpose planners has been out of the scope of that work.

Herry et al. [53] describe a prototype implementation of a configuration system based on automated planning that generates and execute plans for configuration changes. Their prototype shows that it is possible to combine automated planning and common system configuration tools to automatically deploy change plans. Their system is general enough to be used with any planning algorithm. A comparison of different planning algorithms is out of the scope of their work.

Change plan generation is closely related to the execution of change plans, which can be troublesome due to unpredictable failures. To proactively avoid failures when IT changes are executed, we present in [47] an approach to render IT change plans feasible again if the CMDB changes between planning and execution. The proposed solution is independent of the change planning algorithm and we found it to be applicable to smaller sized IT change plans.

Others argue in favor of risk assessment [16, 96, 97] for IT change plans to proactively treat risks during deployment. Similar to risk assessment, the planners evaluated in this work contribute to the proactive treatment of problems because they guarantee the feasibility of the generated plan on a logical level. If proactive solutions should fail, Machado et al. [69, 70] propose a rollback solution to deal with failures during change implementation in a reactive way by undoing partially executed change plans.

Besides the work mentioned herein, we would like to refer to Section 5.5, which provides a broader comparison of related work from the systems management domain. Because change planning - like change verification - aims to automate and make Change Management more reliable, it similarly relates to these investigations as change verification.

6.7 Conclusions

To conclude this chapter, we return to the research questions asked at the beginning of this chapter:

- **Which algorithm is the best in terms of usability?**
  In terms of usability by a change manager, planners without any search control are eas-

iest to use (see Graphlan, Section 6.2.2) because only preconditions and effects of IT changes need to be specified. Unfortunately, the performance is quite bad in these cases. Among the different ways to specify search control for IT changes, we found that algorithm specific control rules (see Prodigy, Section 6.4.2) cannot be readily used by a change manager because profound algorithmic understanding and debugging skills are necessary. Although the linear temporal logic approach used by TLPlan is independent of algorithmic knowledge, it requires a change manager to undergo training in linear temporal logic. Furthermore, we found LTL formulas over state sequences to be an unnatural way to specify search control for IT change planning (see Section 6.3.2). Instead, the decomposition idea inherent to HTN planning very well matches to the generation of change plans from Request for Changes as proposed by ITIL [65]. Decomposition rules can be written without knowledge of the underlying planning algorithm. However, as evidenced by the comparison between the backtracking and no-backtracking measurements, it does not hurt to have knowledge of the planning algorithm. Furthermore, additional specification effort compared to domains without search control is required.

- **Which algorithm is the best in terms of performance?**
  We found that planners without search control, e.g., Graphlan and TLPlan, can only solve problem instances comprising a few Configuration Items (Section 6.2.2) for our deployment case study. TLPlan performs slightly better than Graphlan when search control is specified. Means-end analysis performs better for the change planning case study (see Prodigy, Section 6.4.2) than an undirected forward chaining planner (Graphlan) and a temporally controlled one (TLPlan) because it does not instantiate all applicable actions and prunes the largest portions of the search space. HTN planning delivers the best performance for the case study because its rigorous decomposition concept prunes large portions of the search space and considers the appropriate IT changes at the right time. A naively written HTN domain for the case study is faster than Prodigy for small CMDBs (the extent depends on the selectivity). However, on larger CMDBs Prodigy becomes better. Nevertheless, we showed that a carefully engineered HTN domain (without backtracking) outperforms all other algorithms by a factor of 28 to 333 at 100% selectivity for the deployment case study.

For automated planning approaches to emerge from research prototypes to commercial service management products, performance on large CMDBs and usability by a change manager are key factors of success. Given the results of the experiments presented in this chapter, we believe that HTN algorithms possess both characteristics. But all that glistens is not gold. The polynomial runtime complexity observed for SHOP2 herein still limits its scalability to a couple of thousand Configuration Items. To address this problem, we discuss in Chapter 7 optimizations to further improve the runtime complexity of decomposition-based IT change planning from polynomial to linear or even constant runtime.

CHAPTER 7

Efficient Generation of IT Change Plans on Large Infrastructures

In Chapter 6 we examined several general purpose planners for IT change planning and concluded that *Hierarchical Task Network* (HTN) algorithms are most suitable for IT change planning in terms of performance and usability by an IT change manager. We observed that even *SHOP2* [76] - an HTN planner and the winner of that comparison - struggles to maintain acceptable planning performance on large CMDBs comprising a couple of thousand Configuration Items. However, an IT change planner needs to scale to very large configurations of the CMDB as configurations of infrastructures, e.g., a cloud computing provider's Infrastructure as a Service cloud, can comprise up to millions of Configuration Items. Previous work [32, 33, 35, 46, 48, 53, 59, 71, 91] on IT change planning has concentrated on proving the feasibility of IT change plan generation for small to medium size planning problems and questions of scalability have either been out of their scope or scalability could not be achieved by them.

To overcome the scalability issues of IT change planning, we propose and evaluate in this chapter optimization techniques for decomposition-based IT change planning algorithms that make IT change planning feasible on large infrastructures. The introduced optimizations reduce the runtime complexity of several key operations part of the simple task network planning algorithm that is used by SHOP2 [76] and our IT change planning system. In a sensitivity analysis we examine the influence of several important characteristics of IT changes and the CMDB on the runtime complexity of the proposed optimizations. The analysis shows that our optimizations outperform SHOP2 [76] (the winner of the previous comparison among IT change planners presented in Chapter 6) in terms of runtime complexity for a large percentage of IT changes and CMDBs. In addition to that, the optimizations proposed herein are more robust in respect to the characteristics of IT changes and the CMDB, i.e., the planner's runtime is less influenced by the characteristics of an IT change or the CMDB. A cloud deployment case study of a three-tier business application and a virtual network configuration case study demonstrate the feasibility of the approach and confirm the results from the sensitivity analysis. Our optimizations beneficially influence related Change Management problems as well: A faster planner helps to improve the quality of plans as it can generate more plans within the same time and select the best one among them. Furthermore, more time is left to also schedule IT changes into change windows [84, 88, 102] - a step traditionally kept separately from change planning due to its complexity.

The remainder of this chapter is organized as follows. In Section 7.1 we discuss related work and contributions. Section 7.2 describes the planning algorithm and introduces the optimizations to improve decomposition-based IT change planning. In Section 7.3 we compare both planning systems in a sensitivity and robustness analysis followed by a comparison using case studies in Section 7.4. Finally, we conclude in Section 7.5.[1]

7.1 Related Work and Contributions

The *CHAMPS* system by Keller et al. [59] is the seminal work to address IT change planning. The work proposes a planning algorithm that traverses a dependency tree to derive a plan. Different to the planning algorithm used herein, CHAMPS can only plan for IT changes once dependencies among them have been explicitly specified and cannot infer a plan from preconditions and effects of IT changes. The applicability and optimization of planning over large infrastructures has not been explored by CHAMPS.

Maghraoui et al. [71] apply an optimization of the UCPOP [82] *Artificial Intelligence* (AI) planning algorithm to IT change planning. Although the authors use a different case study, scalability is reported to be a few hundred Configuration Items. Our experience with UCPOP for IT change planning is that its performance is worse than that of all other planners we previously examined in Chapter 6. In addition to that, UCPOP relies on first-order unification of which we show herein that it has serious scalability issues.

Cordeiro et al. [33, 35] propose the *ChangeLedge* system and a constraint aware decomposition approach [32] for IT change planning. The work focuses on the reuse of knowledge in IT change design by capturing best practices and plan generation taking into account the effects IT changes have on one another. Similar to CHAMPS, this early research on change plan generation does not address planning in the large.

Trastour et al. [91] propose *ChangeRefinery*, a system for operator-assisted decomposition of IT change plans. SHOP2, ChangeRefinery, and our work share the same task network decomposition algorithm. However, the authors report scalability up to a few hundred Configuration Items. Applicability to very large CMDBs, a comparison with SHOP2, and a complexity analysis has been out of the scope of their work.

Herry et al. [53] describe a prototype implementation of a configuration system based on automated planning that generates and execute plans for configuration changes. Their prototype shows that it is possible to combine automated planning and common system configuration tools to automatically deploy change plans. Their system is general enough to be used to with any planning algorithm. Scaling planning algorithms to very large configurations is not a topic of this work.

In a previous work [46] we propose a hybrid HTN and state-space planning algorithm for IT change planning. Different to Trastour et al. [91], it supports planning according to state-based constraints as well. A performance comparison including SHOP2, runtime complexity optimizations, and scalability to large CMDBs have been out of the scope of that work.

[1]Parts of this chapter previously appeared in [45].

In Hagen et al. [48] we also present a pure state-space planner for IT changes. The approach can only be used to plan for a small subset of IT changes. However, the focus on a smaller set of IT changes enables scalability to several thousand Configuration Items - still significantly less than the numbers reported herein while the expressiveness of the algorithm and the case study presented herein encompasses the constraints previously discussed in [48].

In a recent work [47] we propose an algorithm for the adaptation of infeasible change plans due to unpredictable changes to the IT infrastructure. Although not stressed in this work, the plans constructed by our algorithm are ready for adaptation using the techniques previously presented in [47].

The basic ideas underlying the task-decomposition planning algorithm used in this chapter were developed by Sacerdoti [85] and in the Nonlin planner [89] by Tate. HTN planning systems have been successfully applied to many real-world planning problems [98] and are among the most widely practically used planning techniques. Among other domains, HTN planning has been successfully applied to problems from crisis management [2], evacuation planning [75], robotics [74], planning for spacecrafts [39], and military air campaign planning [99]. There are additional relevant research efforts published in the field of automated planning [41] that explore the generation of plans as well. However, all Artificial Intelligence planners rely on a first-order satisfiability solver, which performs well for smaller problem instances but - as shown in this work - does not scale to domain sizes as they appear for IT change planning. Due to this reason, the fact that these investigations do not address planning for IT changes, and the common sense that knowledge-based planning algorithms perform better than those not making use of search control [98], they are not approached in this work.

Related to IT change planning is the problem of IT change scheduling. IT change scheduling [84, 88, 101, 102] aims to schedule a set of IT changes in accordance with their precedence constraints into change windows while minimizing a given metric. Different to IT change planning, which generates a set of atomic change activities that, when executed, achieve a Request for Change, change scheduling assigns atomic activities to change windows for execution. Thus, planning determines what needs to be done and scheduling decides when to do it.

Several other aspects of IT management have been addressed in recently published work. Among these are approaches to improve the process to manage IT incidents [14] or to staff people [36], and towards connecting risk analysis with IT change prioritization [87]. A more detailed discussion of broader related work in the area of systems and Change Management has previously been given in Section 5.5.4 in the context of change verification. Similar to change verification, change planning is automated and aims to increase the reliability of the Change Management process. Thus, observations made for work related to change verification also hold for change planning.

It is important to emphasize that our work aims at being generic enough for several controlled, well known IT environments. Examples include private data centers and cloud computing. As for the specific management interfaces required for IT changes, we rely on the research efforts that have been carried out to standardize them. Important examples can be found in the context of cloud computing, where such efforts [52, 66] have been driven towards the interoperability across different cloud providers.

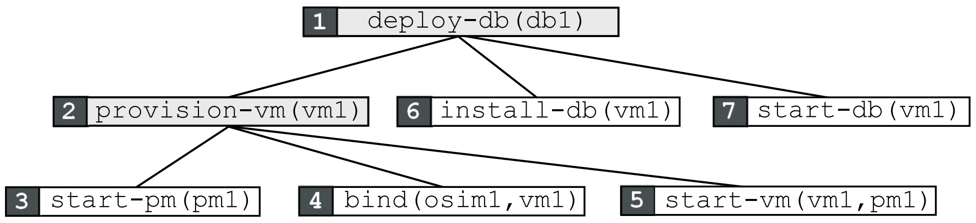


Figure 7.1: HTN decomposition tree of the abstract change activity to deploy a database (gray) into atomic change activities (white).

7.2 Hierarchical Task Network Planning (HTN)

7.2.1 Introduction to HTN planning

For IT change planning based on *Hierarchical Task Network* (HTN) planning [42], a planning problem is given by a *Request for Change* (RFC) for which a sequence of IT activities needs to be computed so as to accomplish the objective of the RFC. Two different types of activities, i.e., IT tasks, are distinguished: *abstract* activities and *atomic* activities. Abstract IT activities are those IT tasks expressed on a high level that are not directly executable. They need to be further (recursively) decomposed into finer-grained activities until a sequence of atomic activities has been determined to implement the abstract activity. The goal to plan for is provided by the RFC, which is directly mapped onto an abstract activity. For example, Figure 7.1 depicts a decomposition tree for the abstract activity (gray) / RFC to deploy a database. The rules to decompose abstract activities into finer-grained change activities are called (HTN) *methods*. Several decomposition rules, representing different problem solving strategies, can exist to decompose an abstract activity. Atomic activities (white) in turn cannot be further decomposed because they are the most basic IT activities out of which abstract activities are composed. An atomic activity carries the specification of a precondition that needs to hold to apply it and a specification of its effects. A *Simple Task Network (STN) Planner*, a simplified form of HTN planning, is a forward-chaining (starting from the current state of the CMDB) ordered, depth-first search tree-decomposition algorithm. The computed plan to implement the RFC only comprises the leaf nodes of the decomposition tree, i.e., the atomic activities. For example, the computed plan to deploy a database for the decomposition tree in Figure 7.1 comprises atomic activities 3, 4, 5, 6, and 7. The planning algorithm guarantees that preconditions and effects of the atomic activities complement each other in such a way that the plan is executable.

7.2.2 Planning Algorithm

Algorithm 1 depicts a simplified version of a simple task network planning algorithm that is the least common multiple of SHOP2 [76] and our algorithm. Key operations of the simple decomposition algorithm have been optimized - often making the difference between constant, linear, and polynomial runtime. The algorithm works in both planners as follows.

As long as the stack of IT activities to plan for is not empty (Line 3) an activity is popped from the stack (Line 4) and planned for (depth-first search). If *act* is an atomic activity (Line 5), its parameters are adapted (Line 8) until its precondition is satisfied (Lines 7-9). For example, when planning for the atomic start-pm activity (activity 3) in Figure 7.1, the physical machine parameter needs to be set to a physical machine that satisfies the precondition that comes with the start-pm activity. For instance, the physical machine chosen needs to be in state off. In

Algorithm 1: SimpleDecomposition(RFC,methods): Simple decomposition planning algorithm

```
Data: RFC: the Request for Change to plan for, methods: list of HTN methods used for
      planning
Result: totally ordered change plan
plan = ⟨⟩ ;                                                                  ▹O( 1 )
stack = ⟨RFC⟩ ;                                                              ▹O( 1 )
while !stack.empty() do                                                      ▹O( 1 )
    act = stack.pop() ;                                                      ▹O( 1 )
    if act is atomic activity then                                           ▹O( 1 )
        it = act.getParameterIterator() ;                                    ▹O( 1 )
        while it.hasNextBinding() ∧ !act.precondition() do   ▹O( see Subsection 7.3.3 )
            act.setNextParameterBinding() ;                  ▹O( see Subsection 7.3.3 )
        end
        if act.precondition() then                           ▹O( see Subsection 7.3.3 )
            act.applyEffects() ;                             ▹O( see Subsection 7.3.5 )
            plan = plan ∘ act ;                                              ▹O( 1 )
        else                                             ▹atomic activity not applicable
            if !backtrack() then                             ▹O( see Subsection 7.3.4 )
                return fail ;                     ▹if backtracking fails, no plan exists
            end
        end
    else                                              ▹act needs to be further decomposed
        choose parameters for act and a method m ∈ methods such that act can be
        decomposed using m ;                                 ▹O( see Subsection 7.3.3 )
        if act.precondition() then                           ▹O( see Subsection 7.3.3 )
            ⟨act_1, ..., act_n⟩ = act.decompose(m) ;                         ▹O( n )
            stack.push(⟨act_1, ..., act_n⟩) ;                                ▹O( n )
        else                                               ▹act cannot be decomposed
            if !backtrack() then                             ▹O( see Subsection 7.3.4 )
                return fail ;                     ▹if backtracking fails, no plan exists
            end
        end
    end
end
return plan ;                                     ▹planning successful, return plan
```

Section 7.3.3 we compare the runtime complexity to determine valid parameter bindings for change activities. If the atomic activity is executable (Line 10), i.e., Configuration Items were found for the parameters of the change such that it became executable, its effects are applied to the CMDB (Line 11, see Section 7.3.5 for complexity) and it is added to the end of the plan (Line 12). The planner has to backtrack [31] if the atomic activity is not applicable, i.e., at a previous decision point an abstract activity has to be decomposed differently or parameters of an activity have to be altered to explore a different plan. For example, assume that in Figure 7.1 activity 6 to install the database on vm1 might not be executable. The planner would have to backtrack over activities 5, 4, and 3 to introduce a different virtual machine for deployment in activity 2. Section 7.3.4 examines the backtracking performance for both planners. Similar to atomic activities, parameters have to be adapted such that a method can be used to decompose an abstract IT activity (Line 19). For example, in Figure 7.1 the virtual machine parameter of activity provision-vm can be restricted to specific virtual machines depending on the precondition of activity 2 and the method used to decompose it. If a method is applicable, the method is used to decompose the abstract activity (Line 21) and the children are pushed on the stack (Line 22). If no decomposition is possible (Line 23), the planner needs to backtrack to choose different parameters or another decomposition alternative for an abstract activity that has been previously planned for. When the stack of change activities to plan for runs empty and backtracking did not fail previously, a valid plan can be returned (Line 30).

7.2.3 Unification vs. Object-oriented Planning

In this section we describe optimizations for several critical operations of Algorithm 1 that make the difference between constant, linear, and polynomial planning complexity compared to SHOP2 - previously found to be the fastest planner in a comparison of IT change planners in Chapter 6. Notice that our optimizations do not depend on a specific programming language. However, some optimizations depend on the availability of pointers / references, which are typically found in object-oriented languages.

Evaluation of Preconditions

To decide whether methods or atomic activities are applicable, preconditions need to be evaluated on Configuration Items. This happens several times in Algorithm 1, e.g., in Lines 7, 10, and 20. To evaluate a precondition, SHOP2 has to unify the precondition. Unification [9, 58] is an algorithmic process that aims to solve a satisfiability problem. The goal of unification is to find a substitution such that two seemingly different terms (the term describing the precondition and the terms stored in the CMDB describing the configuration of the Configuration Items) are equal. Unification is a potentially expensive operation (between linear and polynomial runtime complexity). Readers interested in understanding the complexity, the logical background, and the algorithms for unification may be referred to [9, 58]. Different to SHOP2, we do not make use of unification to evaluate preconditions. Preconditions in our approach are implemented as Boolean methods that are executed over an object-oriented CMDB model. The parameters of atomic change activities are pointers / references to the Configuration Items the precondition is evaluated on. Thus, the properties of the Configuration Items that are necessary to determine the satisfiability of a precondition can be accessed in constant time through references. Consequently, a precondition can be evaluated in constant (e.g., check whether a property holds a value) or linear time (e.g., determine whether an element is comprised in a list). Different to the specific preconditions and effects of the change verification logic, no requirements need to be enforced on the logical expressiveness of preconditions and effects for change planning because the algorithm is a total-order planning algorithm.

Determining Parameters of IT Activities

During planning, parameters of an IT activity need to be bound to Configuration Items, e.g., in Lines 8 and 19, so as to satisfy the activity's precondition. To determine parameter bindings for activities, SHOP2 makes use of an optimized version of the unification algorithm proposed in Nilsson [77]. Different to SHOP2, we do not use unification but determine Configuration Items of parameters by simply iterating over the CMDB until an assignment of Configuration Items to parameters is found that satisfies the precondition. To achieve this, we scan through the CMDB in linear time (to determine one parameter). This is often an improvement over the polynomial complexity caused by Nilsson's algorithm [77].

Application of Effects

During planning, the effects of atomic change activities need to be applied (Line 11, Algorithm 1) to the CMDB to take into account their influence on IT activities being planned for hereafter. SHOP2 describes effects as a list of predicates to be added and deleted from the CMDB that is maintained as a list of predicates. In the worst-case, this operation involves a linear scan over the CMDB to find the predicate to remove and eventually a costly unification step should predicates only be known partially. In turn, our CMDB is described by a set of (Java) objects each matching to a Configuration Item. Our effects are implemented as Java-methods that implement code that change these objects, i.e., the Configuration Items of the CMDB. Using references, the affected Configuration Items can be accessed in constant time and the effects can be applied to them immediately.

Undo of Effects

When backtracking over a previously planned atomic activity (Lines 14 and 24), the effects of the activity need to be undone. SHOP2 needs to conduct a linear scan of all predicates to revert activities by removing predicates previously added by the activity to the CMDB. We optimize this process to constant runtime because our Configuration Items keep a copy of the values of their properties prior to the application of an atomic change activity to quickly restore their previous version should changes need to be undone in our object-oriented knowledge base.

7.3 Sensitivity and Robustness Analysis

7.3.1 Design of Microbenchmarks

Microbenchmarks (MBs) are used to practically analyze the runtime-complexity of the non-trivial parts of Algorithm 1 depending on several characteristics of IT activities and the CMDB. Among these are the complexity of an activity's precondition or effects, the layout of the CMDB, and the proportion of Configuration Items that satisfy preconditions.

Our optimization of the simple task decomposition algorithm (Algorithm 1) and the runs of the microbenchmarks have been carefully engineered such that the same search space is explored by both planners. Thus, SHOP2 and our optimized algorithm decompose and apply activities in the same order, choose parameters in the same order, and backtrack at the same time over the same activities. It is also important to keep in mind that the microbenchmarks do not have side-effects. Thus, their individual results can be combined to assess the runtime when combining different types of IT activities over different instances of a Configuration Management Database in a case study. Consequently, a case study is a weighted, cumulative combination of microbenchmarks because they individually stress a specific operation of Algorithm 1 while solving other operations in constant time.

7.3.2 Parameters of Sensitivity Analysis

Layout of the Configuration Management Database

Let p be a parameter of an IT activity *act* and *pre*(p) a precondition of *act* whose truth value depends on the value assigned to p. Each Configuration Item (CI) can appear in either one of the two roles in respect to precondition *pre* and its parameter p:

- **Qualified CI:** A Configuration Item *ci* is called *qualified CI* in respect to a change activity *act* and a precondition *pre*(p) if *pre* holds when the value of parameter p is set to *ci*, i.e., *pre*(p := *ci*) == *true*. For example, consider the atomic change activity to start a physical machine whose precondition requires the machine to be in state *off*. Thus, all machines that are in state *off* are qualified Configuration Items in respect to that change and precondition.

- **Unqualified CI:** A Configuration Item *ci* is called unqualified CI in respect to a change activity *act* and a precondition *pre*(p) if *pre* does not hold when the value of parameter p is set to *ci*, i.e., *pre*(p := *ci*) == *false*. For example, consider an activity *act* to start a database. A precondition *pre* of *act* is that the database is installed and in state *off*. Thus, all database Configuration Items that are either not installed or running are unqualified Configuration Items for *act*.

Unqualified Configuration Items are a threat to the planner's performance because they cannot be used to instantiate an IT activity such that its precondition is satisfied. Consequently, the existence of unqualified Configuration Items can prolong the search for proper parameter bindings or cause backtracking. Whether this is the case depends on the layout of the CMDB that describes the order Configuration Items are stored. SHOP2 and our approach maintain the CMDB as a list of Configuration Items/predicates. Thus, we distinguish two layouts of a CMDB:

- **Qualified-unqualified (Q-UQ) layouts:** In Q-UQ layouts all qualified Configuration Items appear before unqualified Configuration Items.

- **Unqualified-qualified (UQ-Q) layouts:** In UQ-Q layouts all unqualified Configuration Items appear before qualified Configuration Items.

Both layouts cover the best- and worst-case situations of SHOP2 and our algorithm in terms of the distribution of Configuration Items in the CMDB. Consequently, as we are interested in the best- and worst-case runtime performance of the algorithms, we do not consider statistical distributions for the location of Configuration Items stored in the CMDB.

Selectivity

The layout of the Configuration Management Database describes the order qualified and unqualified Configuration Items appear in the CMDB, but it does not characterize the percentage of qualified or unqualified Configuration Items existing in the CMDB. The selectivity *sel* of a precondition of a change activity denotes the proportion of qualified Configuration Items among all Configuration Items of the CMDB. Thus,

$$sel = \frac{|\text{qualified CIs}|}{|\text{CMDB}|}.$$

Large selectivity means that many Configuration Items can be chosen as valid parameters for IT activities, thus, leaving less wrong choice opportunities for the planner. The benchmarks consider three values for selectivity to assess a performance trend:

- **100% selectivity:** The CMDB only comprises qualified Configuration Items, i.e., all Configuration Items of a certain type are suitable to render the precondition of an IT activity true. Notice that the CMDB layout does not have any influence in this case because Q-UQ and UQ-Q layouts reduce to Q layouts.
- **50% selectivity:** Half of the CMDB's Configuration Items are qualified CIs. Depending on the CMDB layout these are either located at the front (Q-UQ) or the end of the CMDB (UQ-Q).
- **0% selectivity:** The CMDB comprises only the minimal number of qualified Configuration Items for the planning problem to be solvable. The CMDB layout determines whether these Configuration Items appear at the beginning (Q-UQ) or the end of the CMDB (UQ-Q).

Notice that the best- and worst-case runtimes are always covered by 0% and 100% selectivity. Similar to the CMDB layout, we omit a distribution for the selectivity as we are interested in a comparison of best- and worst-case performance.

7.3.3 Influence of Complexity of Preconditions

Microbenchmark 1 (MB1) compares the performance of both planners when it comes to adapt parameters of activities such that their preconditions are satisfied (Lines 7, 8, and 10) or that they can be decomposed by a decomposition method (Lines 19 and 20). Besides the CMDB layout and the selectivity, the following parameters also influence the matching of preconditions: (1) the number of Configuration Items the precondition addresses and (2) whether parameters are chosen in an atomic activity, i.e., in Line 8, or during decomposition of an abstract activity, i.e., in Line 19 of Algorithm 1. Table 7.1 summarizes the complexity results of all 24 configurations of MB1.

Performance

Our optimizations result in constant runtime on qualified-unqualified (Q-UQ) CMDBs as the first Configuration Items chosen for the parameters satisfy the precondition. SHOP2 shows polynomial and linear runtime for Q-UQ CMDBs depending on whether the unification algorithm to compute parameter bindings scales linearly or polynomially with the size of the CMDB. Thus, for Q-UQ CMDBs our planner always outperforms SHOP2 in runtime complexity. Due to the strong results, we do not further discuss the performance on Q-UQ CMDBs herein. Readers interested in the graphs of MB1 on Q-UQ CMDB layouts may be referred to Figure D.2 in

	Microbenchmark 1 parameters				Complexity		Relative Performance	
	CMDB	Pre. in	#CIs	Selectivity	OWN	SHOP2	OWN	SHOP2
(1)	Q-UQ	Method	1	0%	**const**	**linear**	**same**	**fastest**
(2)	Q-UQ	Method	1	50%	**const**	**polyn**	**same**	**intermediate**
(3)	Q-UQ	Method	1	100%	**const**	**polyn**	**same**	**slowest**
(4)	Q-UQ	Method	2	0%	**const**	**polyn**	**same**	**slowest**
(5)	Q-UQ	Method	2	50%	**const**	**polyn**	**same**	**fastest**
(6)	Q-UQ	Method	2	100%	**const**	**polyn**	**same**	**slowest**
(7)	Q-UQ	Activity	1	0%	**const**	[**const**:**linear**]	**same**	**slowest**
(8)	Q-UQ	Activity	1	50%	**const**	[**const**:**linear**]	**same**	**intermediate**
(9)	Q-UQ	Activity	1	100%	**const**	[**const**:**linear**]	**same**	**fastest**
(10)	Q-UQ	Activity	2	0%,	**const**	**polyn**	**same**	**slowest**
(11)	Q-UQ	Activity	2	50%	**const**	**polyn**	**same**	**intermediate**
(12)	Q-UQ	Activity	2	100%	**const**	[**const**:**linear**]	**same**	**fastest**
(13)	UQ-Q	Method	1	0%	**linear**	**linear**	**slowest**	**fastest**
(14)	UQ-Q	Method	1	50%	**linear**	**polyn**	**intermediate**	**intermediate**
(15)	UQ-Q	Method	1	100%	**const**	**polyn**	**fastest**	**slowest**
(16)	UQ-Q	Method	2	0%	**polyn**	**polyn**	**slowest**	**slowest**
(17)	UQ-Q	Method	2	50%	**polyn**	**polyn**	**intermediate**	**fastest**
(18)	UQ-Q	Method	2	100%	**const**	**polyn**	**fastest**	**slowest**
(19)	UQ-Q	Activity	1	0%	**linear**	[**const**:**linear**]	**slowest**	**same**
(20)	UQ-Q	Activity	1	50%	**linear**	[**const**:**linear**]	**intermediate**	**same**
(21)	UQ-Q	Activity	1	100%	**const**	[**const**:**linear**]	**fastest**	**same**
(22)	UQ-Q	Activity	2	0%	**polyn**	[**const**:**linear**]	**slowest**	**same**
(23)	UQ-Q	Activity	2	50%	**polyn**	[**const**:**linear**]	**intermediate**	**same**
(24)	UQ-Q	Activity	2	100%	**const**	[**const**:**linear**]	**fastest**	**same**

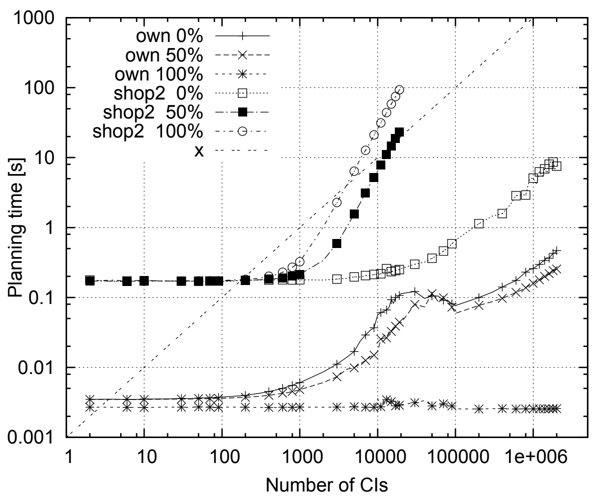
Table 7.1: Runtime complexity of Microbenchmark 1

Appendix D.

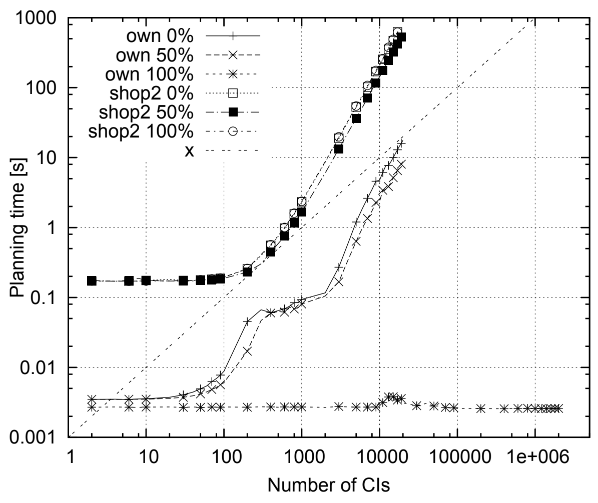
The runtime complexity of both planners is closer to each other on unqualified-qualified (UQ-Q) CMDBs. Figure 7.2 depicts the $log - log$[1] plots of the planning time depending on the CMDB size for the four configurations of MB1 on UQ-Q CMDB layouts.

Figure 7.2a depicts the case of matching over a single Configuration Item inside a method (Line 19, Algorithm 1) on a CMDB where unqualified Configuration Items precede qualified Configuration Items (UQ-Q layout). If the selectivity is 100%, our optimizations have constant runtime because in this case the CMDB only comprises qualified Configuration Items and the first choice for a parameter satisfies the precondition. In turn, SHOP2 has polynomial complexity for 100% selectivity. If the selectivity is 50%, SHOP2 still shows polynomial complexity because of the unification costs and our optimizations only degrade to linear runtime because the planner has to search the prefix of unqualified Configuration Items at the front of the CMDB until a qualifying CI is found. At 0% selectivity SHOP2's runtime becomes linear - the same complexity as our planner. However, our planner outperforms SHOP2 for CMDBs of up to 2 million Configuration Items.

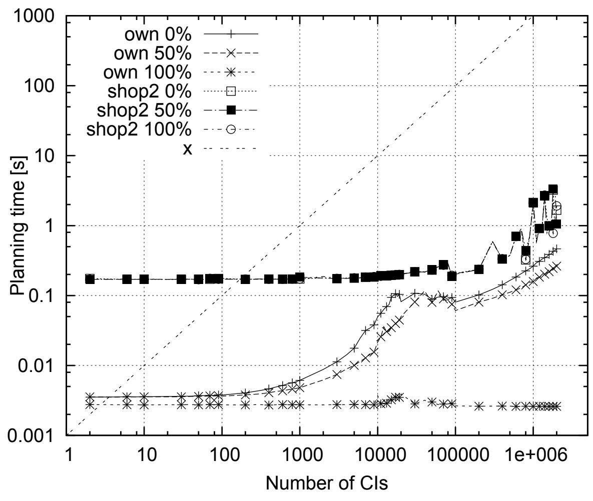
[1]In $log - log$ plots the processing complexity is $O(\alpha n) \leftrightarrow slope == 1$, $O(\alpha n^{slope}) \leftrightarrow slope > 1$ and below linear for $slope < 1$. All measurements were made on an Intel Xeon Processor with 2.8Ghz and 4 GB of RAM.



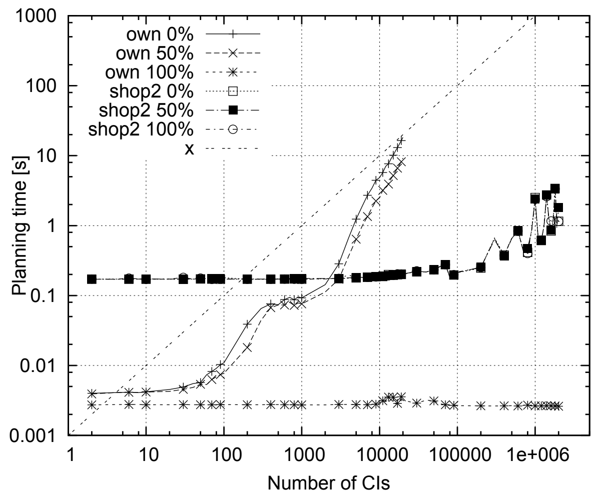
(**a**) MB1 (13,14,15): UQ-Q, Method, 1 CI



(**b**) MB1 (16,17,18): UQ-Q, Method, 2 CIs



(**c**) MB1 (19,20,21): UQ-Q, Activity, 1 CI



(**d**) MB1 (22,23,24): UQ-Q, Activity, 2 CIs

Figure 7.2: Runtime performance of Microbenchmark 1 for UQ-Q CMDB layouts.

Notice that the actual runtime complexity of our algorithm is sometimes disguised by Java's *Just In Time Compiler* (JITC). For example, in Figure 7.2a, for 0% and 50% selectivity we can observe linear growth (slope of curves is one) between 1,000 and 10,000 Configuration Items. Then, the JITC jumps in and compiles frequently used code for native execution instead to interpret it in the Java virtual machine. Performance becomes even faster for larger problem instances (between 10,000 and 100,000) because more code is being compiled with increasing size of the CMDB causing a larger gain in performance than the additional runtime caused by the increasing CMDB size. For even larger problem instances runtime however tends to increase again once the performance advantage of compiled code vs. interpreted code is outweighed by the increasing problem size (beyond 100,000 Configuration Items). However, growth in runtime is less strong than at the beginning because compiled code takes less execution time.

Figure 7.2b depicts a similar configuration of MB1, but the precondition needs to be matched over two Configuration Items. Again, our optimizations result in constant runtime at 100% selectivity because the Configuration Items chosen first satisfy the precondition. SHOP2 has polynomial runtime for all selectivities, similar to our optimized version for 0% and 50% selectivity because it has to search the whole prefix of unqualified Configuration Items of the CMDB until the first pair of qualifying Configuration Items is found. The larger the selectivity, the smaller the UQ part of the CMDB, resulting in faster planning times when selectivity increases. Despite having the same runtime complexity as SHOP2, our approach nevertheless outperforms SHOP2.

Figure 7.2c depicts the case where the precondition of an atomic activity needs to be matched over a single Configuration Item. Similar to the previous cases, our optimizations perform this task in constant time if the selectivity is 100% as the CMDB only comprises qualified Configuration Items. For lower selectivities our optimizations degrade to linear runtime performance as the UQ prefix of the CMDB needs to be searched again. This takes the longer the lower the selectivity. In Figure 7.2c our approach always outperforms SHOP2.

Figure 7.2d depicts the only case of MB1 where SHOP2 has better runtime complexity than our optimizations: matching a precondition over two Configuration Items in an atomic activity on a UQ-Q CMDB. If the selectivity is 100%, the CMDB only comprises qualified Configuration Items immediately yielding a valid choice of Configuration Items for our approach. For lower selectivities a polynomial search through the prefix of unqualified Configuration Items to find the first pair of matching qualified Configuration Items becomes necessary for our approach. However, SHOP2 always shows linear runtime on unqualified-qualified (UQ-Q) CMDBs when matching the precondition in an activity because unification in activities matches from the back of the CMDB leading to an immediate hit because of the UQ-Q layout.

All in all, our approach outperforms SHOP2 in 22 out of 24 configurations of MB1 (92%).

Robustness

To make a statement about the robustness of both approaches, we examine whether each parameter of Microbenchmark 1 influences the runtime performance of both planners. We group the performance results of all configurations of MB1 by every distinct combination of $n - 1$ benchmark parameters to determine whether the n-th parameter influences the runtime performance. For example, Table 7.1 is grouped by every distinct combination of ⟨CMDB layout, precondition, #CIs⟩. Thus, looking at the last two columns of Table 7.1, we can observe that in 50% of all configurations of MB1 the performance of our optimizations is independent of the selectivity, but for SHOP2 this is only the case for 6 (configurations 19 through 24) out of 24 configurations of MB1. Thus, the performance of SHOP2 is only independent of the selectivity in 25% of all cases making our optimizations more robust than SHOP2 in MB1.

Similarly, in 100% vs. 17% of all configurations our planner's performance is the same for matching a precondition in an atomic or abstract activity. In 66% vs. 33% of all configurations of MB1 our optimizations show the same performance when matching one or two preconditions. However, this result is compensated by SHOP2, which is in 66% of all configurations agnostic of the CMDB layout. In turn, this is only the case in 32% of all configurations for our approach.

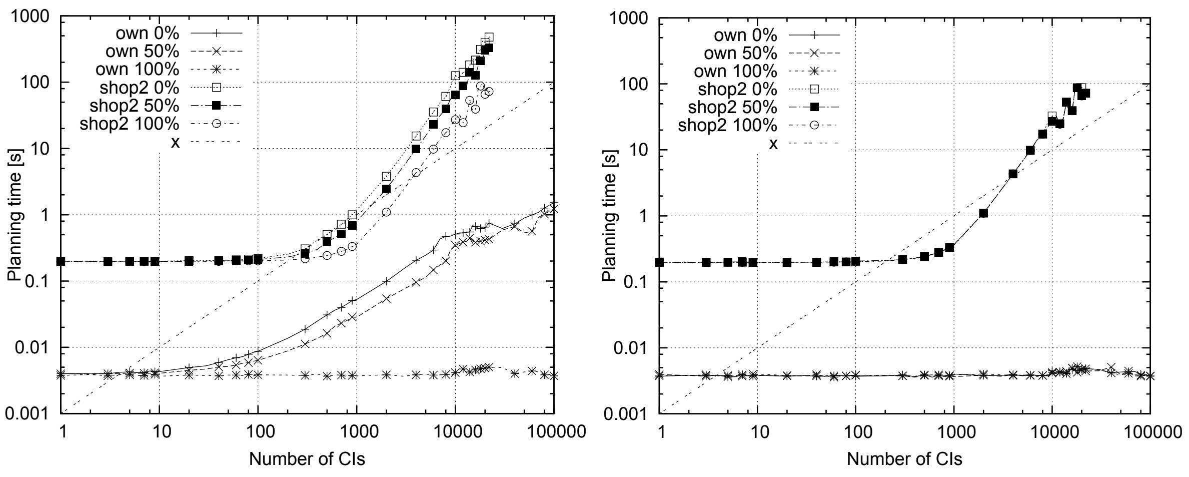
All in all, we conclude that taking into account all parameters of MB1 our approach is less influenced by the specific characteristics of the planning domain.

7.3.4 Influence of Backtracking

Backtracking [31] is an inherent part of the planning algorithm (Algorithm 1, Lines 14 and 24). During backtracking previously planned IT activities need to be undone and a different decomposition choice or choice of parameters needs to be made. *Microbenchmark 2* (MB2) measures the runtime complexity in a backtracking scenario taking into account (1) the layout of the

	Microbenchmark 2 parameters			Complexity		Relative Performance	
	CMDB	effect?	Selectivity	OWN	SHOP2	OWN	SHOP2
(1)	UQ-Q	yes	0%	linear	polyn	**slowest**	**slowest**
(2)	UQ-Q	yes	50%	linear	polyn	**intermediate**	**intermediate**
(3)	UQ-Q	yes	100%	const	polyn	**fastest**	**fastest**
(4)	UQ-Q	no	0%	linear	polyn	**slowest**	**slowest**
(5)	UQ-Q	no	50%	linear	polyn	**intermediate**	**intermediate**
(6)	UQ-Q	no	100%	const	polyn	**fastest**	**fastest**
(7)	Q-UQ	yes	0%	const	polyn	**same**	**same**
(8)	Q-UQ	yes	50%	const	polyn	**same**	**same**
(9)	Q-UQ	yes	100%	const	polyn	**same**	**same**
(10)	Q-UQ	no	0%	const	polyn	**same**	**same**
(11)	Q-UQ	no	50%	const	polyn	**same**	**same**
(12)	Q-UQ	no	100%	const	polyn	**same**	**same**

Table 7.2: Runtime complexity results of Microbenchmark 2



(a) MB2 (1,2,3): UQ-Q, with effect **(b)** MB2 (7,8,9): Q-UQ, with effect

Figure 7.3: Selected cases of the runtime performance of Microbenchmark 2 on UQ-Q and Q-UQ CMDB layouts.

CMDB, which determines whether backtracking occurs, (2) the selectivity, which influences the extent to which backtracking occurs, and (3) whether the planner has to revert effects during backtracking. Table 7.2 depicts the runtime complexity of both planners for all 12 configurations of MB2.

The runtime complexity of SHOP2 is always polynomial because when the size of the CMDB increases linearly, the time spent for first-order unification increases polynomially. We can observe a polynomial increase in the number of calls made to Nilsson's first-order unification algorithm. In 66% of all cases - when qualifying Configuration Items are ordered at the beginning of the CMDB or selectivity equals 100% - our algorithm solves the backtracking benchmark in constant time because backtracking does not occur for these cases as the first choice made for the parameters satisfies the preconditions right away. In 33% of the cases - when unqualified Configuration Items exist at the front of the CMDB - our approach has linear

	Microbenchmark 3 parameters		Runtime complexity		Relative Performance	
	CI at front	#effect?	OWN	SHOP2	OWN	SHOP2
(1)	yes	1	**const**	**linear**	**fastest**	**fastest**
(2)	yes	4	**const**	**linear**	**intermediate**	**intermediate**
(3)	yes	8	**const**	**linear**	**slowest**	**slowest**
(4)	no	1	**const**	**linear**	**fastest**	**fastest**
(5)	no	4	**const**	**linear**	**intermediate**	**intermediate**
(6)	no	8	**const**	**linear**	**slowest**	**slowest**

Table 7.3: Runtime complexity of Microbenchmark 3

runtime complexity because our optimized version of the algorithm has to backtrack through a prefix of unqualified Configuration Items at the front of the CMDB. In these cases, planning time increases linearly because the number of unqualified Configuration Items at the front of the CMDB increases linearly.

Figure 7.3a depicts the runtime performance when backtracking over actions with effects on UQ-Q CMDBs. The biggest gap in performance is visible for 100% selectivity. For example, for 10,000 Configuration Items our approach solves the backtracking microbenchmark within a few ms, while SHOP2, depending on the selectivity, easily reaches 100s. If selectivities are smaller than 100%, our approach degrades to linear runtime performance as it has to backtrack through the UQ prefix of the CMDB. However, this still outperforms the polynomial runtime of SHOP2.

When the layout of the CMDB is changed to Q-UQ (see Figure 7.3b), backtracking does not occur as the first choice is a valid choice. Thus, our approach solves the planning problem in constant time while SHOP2 still shows polynomial runtime. This makes us conclude that the occurrence of backtracking (as long as it increases the search space linearly as it is the case for our benchmark) does not seem to further degrade the performance of SHOP2 beyond polynomial runtime complexity. Nevertheless, the performance penalty caused by backtracking remains clearly visible for SHOP2. Different to our approach, where the performance gap between backtracking and no backtracking increases linearly with the size of the CMDB, it increases polynomially for SHOP2.

The graphs of the remaining cases of MB2 are similar to the ones of Figure 7.3 and can be found in Figure D.1 in Appendix D. All in all, our planner outperforms the SHOP2 planner in all cases of Microbenchmark 2.

Besides planning performance, robustness of planning time is important as well. From Table 7.2 it can be observed that both planners are only independent of the selectivity on Q-UQ CMDBs. Both planners also have the same robustness for the CMDB layout parameter and the parameter that describes whether the planner has to revert effects on backtracking. Thus, both planners have the same robustness for MB2, but our optimizations outperform SHOP2 in terms of runtime performance in all configurations of MB2.

7.3.5 Influence of Complexity of Effects

Microbenchmark 3 (MB3) analyzes the influence that the complexity of effects of atomic activities have on the runtime of the planner (Line 11, Algorithms 1). The following parameters influence the costs to apply effects: (1) the location of a Configuration Item in the CMDB (either

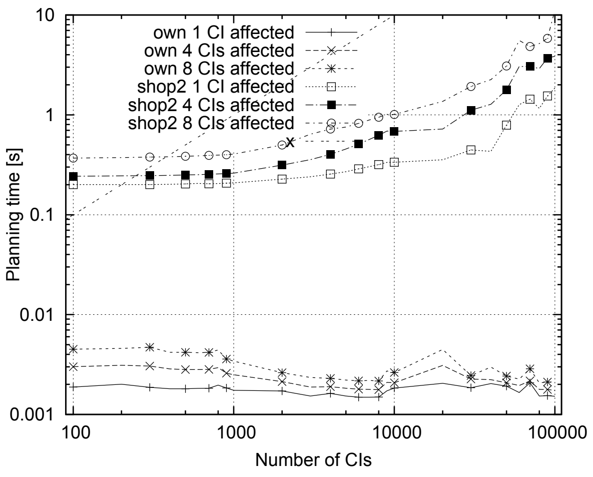


Figure 7.4: MB3 (1,2,3): CI at front

at the front or the back of the CMDB) that is changed by the effects of an atomic activity and (2) the number of Configuration Items affected by the activity (1, 4, or 8). Microbenchmark 3 measures the time to perform 10 IT activities where each activity affects 1, 4, or 8 distinct Configuration Items. A mixture of six blind writes, arithmetic effects, and reference manipulation effects are applied to each affected Configuration Item.

Table 7.3 depicts the complexity results for the six configurations of MB3. Our approach always outperforms SHOP2 by constant instead of linear runtime because our optimizations can access Configuration Items in constant time to directly apply an effect. Figure 7.4 depicts the benchmark results when the Configuration Items affected are ordered at the front of the CMDB. Confirming the general intuition, atomic change activities that affect more Configuration Items yield longer planning durations.

Consequently, our approach is insensitive to the location of a Configuration Item in the CMDB and the size of the CMDB when applying an effect. In contrast, the performance of SHOP2 to apply an effect to a Configuration Item depends on its location and the size of the CMDB: The more to the front of the CMDB the Configuration Item is located or the larger the CMDB, the more expensive the application of effects. This is caused by the different costs to perform unification and to add/remove predicates to/from the CMDB. The planning time increases with the number of Configuration Items affected by the activity for both planners.

All in all, our implementation outperforms SHOP2 in all cases and is more robust regarding the position of a Configuration Item in the CMDB - a benefit as order of Configuration Items in the CMDB cannot be controlled by a change manager.

7.4 Change Planning Case Studies

In this section we evaluate the performance of both planners using case studies instead of isolated characteristics of change activities.

	Case study parameters			Runtime complexity		Relative Performance	
	CMDB	BT domain	Selectivity	OWN	SHOP2	OWN	SHOP2
(1)	Q-UQ	Yes	0%	**const**	**polyn**	**slowest**	**same**
(2)	Q-UQ	Yes	50%	**const**	**polyn**	**intermediate**	**same**
(3)	Q-UQ	Yes	100%	**const**	**polyn**	**fastest**	**same**
(4)	Q-UQ	No	0%	**const**	**polyn**	**slowest**	**slowest**
(5)	Q-UQ	No	50%	**const**	**polyn**	**intermediate**	**intermediate**
(6)	Q-UQ	No	100%	**const**	**linear**	**fastest**	**fastest**
(7)	UQ-Q	Yes	0%	**linear**	**polyn**	**slowest**	**slowest**
(8)	UQ-Q	Yes	50%	**linear**	**polyn**	**intermediate**	**intermediate**
(9)	UQ-Q	Yes	100%	**const**	**polyn**	**fastest**	**fastest**
(10)	UQ-Q	No	0%	**linear**	**polyn**	**slowest**	**slowest**
(11)	UQ-Q	No	50%	**linear**	**polyn**	**intermediate**	**intermediate**
(12)	UQ-Q	No	100%	**const**	**linear**	**fastest**	**fastest**

Table 7.4: Complexity results for three-tier application deployment.

7.4.1 Deployment of a Three-tier Application

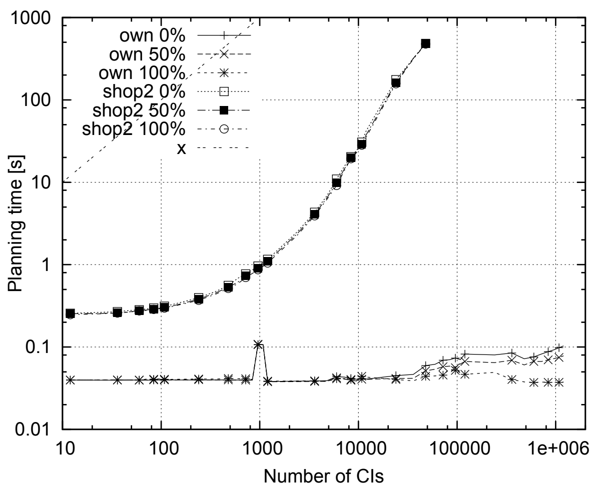
Description of Case Study

This case study evaluates the planning time to deploy a three-tier application (database, application server, and load balancer). The case study comprises 28 abstract IT activities, 48 different decomposition rules, and 16 atomic IT activities. We describe next the constraints according to which a plan is created:

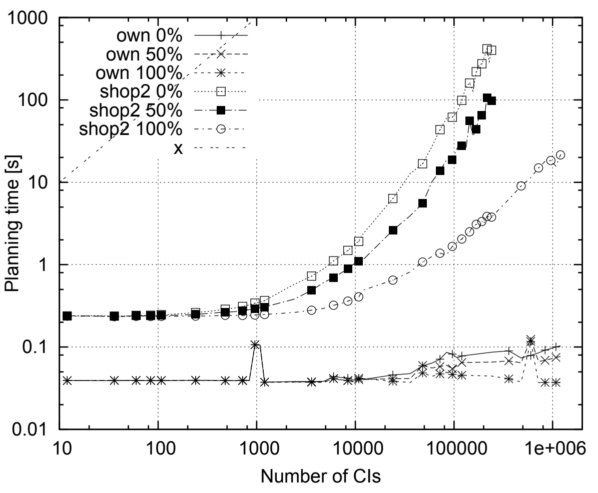
- **Placement constraints:** During planning several placement decisions according to constraints need to be made. For example, virtual machines need to be placed on suitable physical machines, unbound OS images need to be connected to virtual machines, database, application server, and load balancer need to be placed on virtual machines. The concrete placement constraint, e.g., whether enough memory is available on a machine or whether an image has not yet been bound to a machine, does not influence the runtime performance because the time to evaluate constraints on Configuration Items is the same assuming that the same number of properties of Configuration Items need to be matched. However, the CMDB layout and the selectivity of Configuration Items for a placement constraint influence the planning performance.

- **State-based constraints:** During planning, actions need to be planned according to state-related constraints. For example, a physical machine needs to be in state *on* to start a virtual machine on it, an OS image needs to be *mounted* to start a virtual machine, a database needs to be *installed/running* to install/start an application server, or an application server needs to be *installed/running* to install/start a load balancer.

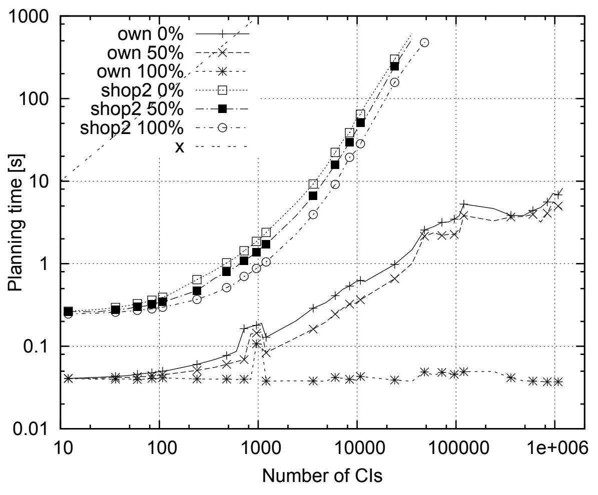
The generated deployment plan comprises atomic activities to bind resources according to the constraints above, activities to mount OS images, change the state of physical machines and virtual machines, and install/start database, application server, and load balancer according to the constraints above. All in all, the case study used herein is comparable to the one in Section 6.1. However, in this chapter we use a highly optimized first-order knowledge base that describes the planning domain with the minimal number of predicates necessary (every Configuration



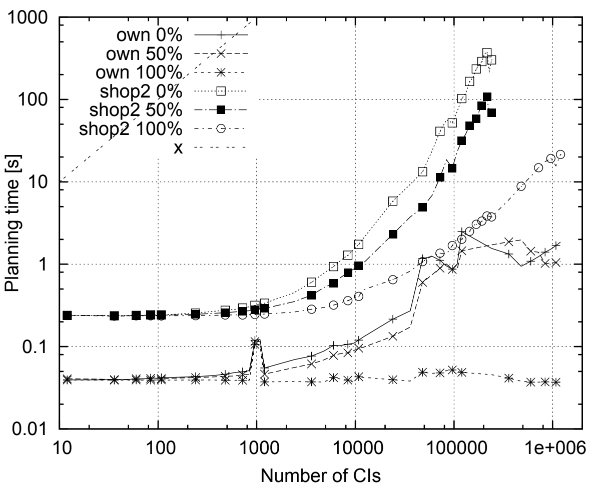
(a) Three-tier application deployment (1,2,3): Q-UQ, backtracking domain



(b) Three-tier application deployment (4,5,6): Q-UQ, no backtracking domain



(c) Three-tier application deployment (7,8,9): UQ-Q, backtracking domain



(d) Three-tier application deployment (10,11,12): UQ-Q, no backtracking domain

Figure 7.5: Runtime performance of three-tier application deployment benchmark for different layouts of the CMDB and the occurrence of backtracking or not.

Item is mapped to one predicate) to reduce the work caused by unification to a minimum. Consequently, although the case studies are comparable, the absolute numbers presented herein for the SHOP2 planner are better than in Chapter 6.

Evaluation

The case study has several change activities for which to select the proper Configuration Items as parameters. Thus, CMDB layout and selectivity have an influence on the performance of the deployment case study because they determine how fast suitable Configuration Items can be found during planning. In addition to that, we examine different planning domains, i.e., problem solving strategies, to solve the case study. One makes use of *backtracking* (BT) [31] (see column named BT domain in Table 7.4) during planning because it commits to parameter bindings of IT changes high up in the decomposition tree that later need to be backtracked over because they do not satisfy the precondition of lower-level atomic activities. Another domain solves the problem without backtracking because placement decisions are made at the latest practicable date when all constraints are known. We consider these two domain descriptions because it mainly depends on the skills and experience of an IT operator how efficiently a planning domain is written: The planning domain without backtracking requires advanced knowledge and experience in formalizing decomposition-based planning domains. Table 7.4 summarizes the complexity results obtained for the three-tier application deployment case study.

On qualified-unqualified (Q-UQ) CMDBs (see Figure 7.5a and 7.5b) our optimizations have constant runtime as the initial choice made for parameters is correct and backtracking does not occur. In turn, SHOP2 shows polynomial runtime behavior on qualified-unqualified (Q-UQ) CMDBs despite the case where selectivity is 100% and the domain is engineered to solve the problem without backtracking. Similar to the microbenchmarks, the runtime of SHOP2 is dominated by the time to perform first-order unification.

Figure 7.5c (Figure 7.5d) depicts the runtime on UQ-Q CMDBs for the domain that solves the three-tier application deployment with backtracking (without backtracking). Different to the constant runtime on Q-UQ CMDBs, our optimizations degrade to linear runtime when selectivity is not 100% on UQ-Q CMDBs because our planner has to backtrack (Figure 7.5c) or search (Figure 7.5d) through a prefix of unqualified Configuration Items at the front of the CMDB. For both approaches backtracking through that prefix is more expensive than just to linearly search it. When the backtracking domain is used, SHOP2 can only derive a plan in polynomial time on UQ-Q CMDBs. When switching to the domain that solves the problem without backtracking, SHOP2 has polynomial runtime despite for 100% selectivity. When backtracking occurs (Figure 7.5c) our planner more significantly outperforms SHOP2. For example, for 0% selectivity SHOP2 can solve a CMDB comprising 3,600 Configuration Items within 10s while our optimizations easily solve problem instances of 1.2 million Configuration Items within 10s (333x the size in 10s). Both planners are closer together (7,200 Configuration Items for SHOP2 vs. 108,000, within 1s, 15x the size) on the optimized domain without backtracking (see Figure 7.5d).

SHOP2 is slightly more robust than our approach when it comes to the deployment case study. The selectivity of the CMDB always influences our optimizations in the deployment case study. In turn, this is only the case in 75% of the benchmark configurations for SHOP2. However, our approach is more robust in respect to the formalization of the domain (backtracking/no backtracking domain). In 66% of all cases our optimizations perform equally for both planning domains while this is never the case for SHOP2. However, SHOP2 is less influenced by the CMDB layout. In 50% of all cases of the deployment benchmark, the performance of SHOP2 is agnostic of the CMDB layout. This is only the case in 33% of all configurations for our optimizations. Taking the influence of all parameters into account, our optimizations are slightly more agnostic (37.5% vs. 25%) to the parameters of the case study than SHOP2.

7.4.2 Virtual Network (Un)configuration Case Study

Description of Case Study

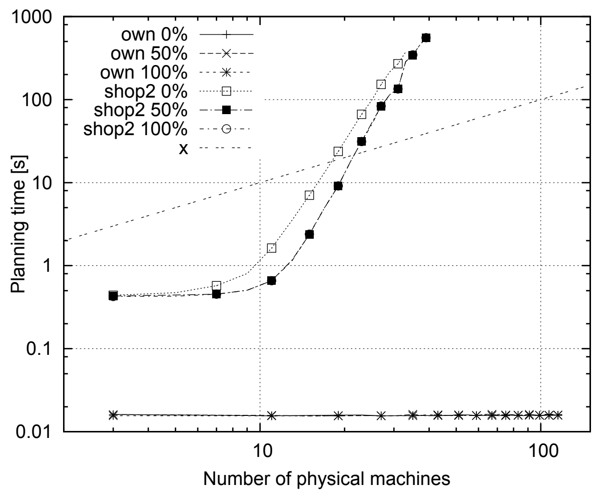
In this case study we evaluate the complexity to configure virtual routers and links on top of a physical network substrate. More specifically, a virtual network is to be configured among three physical machines (pm_1, pm_2, pm_3) such that network connectivity is established between the pairs (pm_1, pm_2) and (pm_2, pm_3). Such a change could be part of a larger deployment change,

	Case study parameters		Runtime complexity		Relative Performance	
	CMDB	Selectivity	OWN	SHOP2	OWN	SHOP2
(1)	UQ-Q	0%	**polyn**	**polyn**	**slowest**	**slowest**
(2)	UQ-Q	50%	**polyn**	**polyn**	**intermediate**	**intermediate**
(3)	UQ-Q	100%	**const**	**polyn**	**fastest**	**fastest**
(4)	Q-UQ	0%	**const**	**polyn**	**same**	**slowest**
(5)	Q-UQ	50%	**const**	**polyn**	**same**	**fastest**
(6)	Q-UQ	100%	**const**	**polyn**	**same**	**fastest**

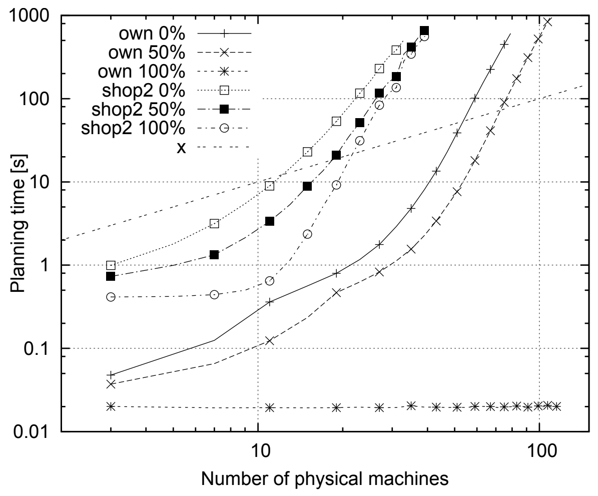
Table 7.5: Complexity results of virtual network configuration case study.

	Case study parameters		Runtime complexity		Relative Performance	
	CMDB	Depth of NW	OWN	SHOP2	OWN	SHOP2
(1)	NW-PMS	3	**const**	**linear**	**fastest**	**fastest**
(2)	NW-PMS	4	**const**	**linear**	**intermediate**	**intermediate**
(3)	NW-PMS	5	**const**	**linear**	**slowest**	**slowest**
(4)	PMS-NW	3	**const**	**linear**	**fastest**	**fastest**
(5)	PMS-NW	4	**const**	**linear**	**intermediate**	**intermediate**
(6)	PMS-NW	5	**const**	**linear**	**slowest**	**slowest**

Table 7.6: Complexity results of virtual network unconfiguration case study.



(a) Network configuration case study (1,2,3): Q-UQ CMDB



(b) Network configuration case study (4,5,6): UQ-Q CMDB

Figure 7.6: Runtime performance of the virtual network configuration case study depending on the layout of the CMDB.

e.g., the three-tier application deployment case study. The constraints taken into account during planning are those that occur when configuring a *Logical Router Service* on Juniper routers as imposed by the JUNOS operating system [62]: (1) a physical router cannot host more than 16 virtual routers and (2) every physical or logical interface can only be used by one virtual router. The planner needs to choose three physical machines such that the interfaces and routers on the network path connecting the physical machines can be properly configured according to

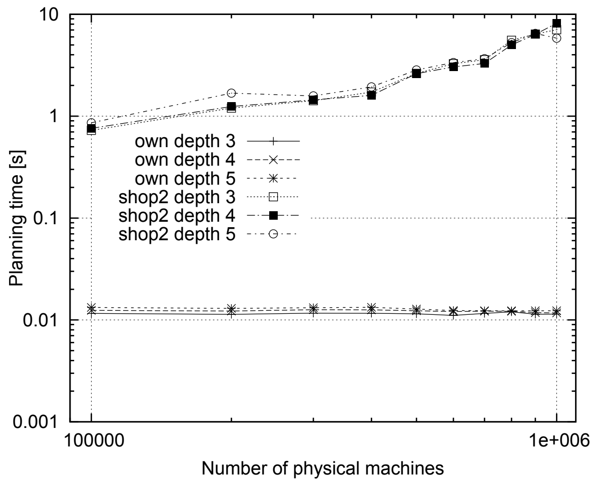


Figure 7.7: Network unconfiguration case study (1,2,3): NW-PMS CMDB layout.

the constraints. Backtracking occurs if the configuration fails among the path. The network layout is of a strict hierarchical nature where the higher-level router aggregates/connects the routers or machines of the lower level. The depth of the network, which is a parameter of the unconfiguration case study, describes the number of hops to the highest level router. Thus, for a depth of n, a total of $2 \times n$ links have to be traversed among any pair of machines. In a second RFC we also evaluate the complexity to unconfigure the virtual network. To plan for both RFCs, the planning domain comprises 14 different abstract IT activities, 22 decomposition rules, and 4 atomic activities (to configure/unconfigure a router/interface).

Evaluation

Table 7.5 depicts the runtime complexity of the virtual network configuration case study. On qualified-unqualified (Q-UQ) CMDBs our optimizations outperform SHOP2 by constant instead of polynomial runtime complexity (see Figure 7.6a) because the first choice made for the physical machines yields a configurable network path (qualified Configuration Items are ordered at the front of the CMDB). Again, SHOP2 spends polynomial time in performing first-order unification. On unqualified-qualified (UQ-Q) CMDBs our approach degrades to polynomial runtime complexity for selectivities below 100%, the same complexity as for SHOP2 (see Figure 7.6b). In this case SHOP2 and our optimizations start backtracking as network configuration fails among all triples of physical machines chosen from the UQ prefix of the CMDB until the first triple of physical machines is chosen from the postfix of qualified Configuration Items such that the network can be configured. Nevertheless, Figure 7.6b shows that - despite the same complexity - our optimizations easily outperform the SHOP2 planner.

In terms of robustness both planners are comparable. Our optimizations are agnostic of the selectivity in 50% of all cases. Different to that, this is never the case for the SHOP2 planner. However, in 30% of all cases SHOP2 is agnostic of the CMDB layout. In turn, this is never the case for our optimizations.

When planning for the virtual network unconfiguration case study (see Table 7.6 and Figure 7.7), our approach always outperforms SHOP2 by constant instead of linear runtime. Undoing the network configuration is easier to handle by both approaches because more parameters of IT changes are already bound to Configuration Items. For example, the physical machines among which to unconfigure the virtual network are already given. Thus, no suitable combination of physical machines needs to be found avoiding the polynomial complexity inherent to the configuration case study. Nilsson's unification algorithm causes linear runtime for SHOP2 in the unconfiguration case study. In turn, our optimizations solve the problem in constant time because we do not use unification but references to quickly derive the parameters of IT activities during decomposition.

Both planners have the same robustness for the network unconfiguration case study. Both are influenced by the depth of the network. The deeper the network, the more routers and interfaces need to be reconfigured causing longer planning durations due to larger networks. Notice that we do not examine selectivity in this case study because it does not have an influence on the runtime performance. This is the case because all changes are completely instantiated in the network unconfiguration case study (it is obvious which routers and interface need to be unconfigured due to the deterministic path among the machines) thus leaving only a few predicates to be considered by unification. The planning performance is influenced by the CMDB layout indifferently. On larger networks we can observe that SHOP2 tends to be penalized by PMS-NW CMDB layouts (all physical machine Configuration Items ordered before network Configuration Items) because more predicates need to be matched in the NW part of the CMDB causing more efforts to scan through the PMS prefix of the CMDB until a match is found. It is difficult to determine robustness as measurements lie closely together on smaller CMDB sizes. All in all, none of the both planners seems to outperform the other one in terms of robustness for the network unconfiguration case study.

7.5 Conclusions

Planning in the large is an imperative feature required of IT change planners in order to cope with the increasing size and complexity of IT infrastructures in most modern organizations. In spite of this, investigations carried out in the context of IT change planning have either left this requirement out of scope, or simply do not deal satisfactorily with CMDBs that contain up to millions of Configuration Items.

To bridge this gap, in this chapter we proposed and discussed optimization techniques for decomposition-based IT change planning over object-oriented CMDBs. The proposed techniques enable a substantial reduction in the runtime complexity of the decomposition process: from polynomial to linear or even constant complexity as evidenced in the various case studies and microbenchmarks presented in this chapter.

Our evaluation has evidenced two important aspects regarding IT change planning. First, object-oriented planning and our optimizations have shown to be a promising approach to deal with planning over very large CMDBs. This is of vital importance to most modern organizations that have to manage up to several change requests a day over large infrastructures in a timely manner. And second, first-order unification suffers from scalability issues. Consequently, typical Artificial Intelligence planners cannot be satisfactorily used for IT change planning in the large as they do not scale for very large CMDBs.

As prospective directions for future research, we suggest research towards algorithms that integrate planning and scheduling while maintaining scalability on large Configuration Management Databases.

CHAPTER 8

Conclusions and Future Work

With businesses relying on the support of IT systems and IT change operations threatening the availability of IT systems, the need to prevent severe outages such as the one caused by a network reconfiguration change in one of Amazon's data centers in April 2011 arises. Although Change Management, a process of the Information Technology Infrastructure Library (ITIL), defines the proper steps and strategies to minimize outages, not much has been done to support the process by automated reasoning techniques. Automated planning and verification of IT change operations are among the most promising steps to be automated in order to guarantee correctness and soundness of IT changes. For that purpose we proposed in this work algorithms for the efficient verification and planning of IT change operations that, compared to previous work on these topics, improve the runtime complexity of change verification and planning from exponential/polynomial to linear or even constant complexity. Thus, both solutions enable the verification and planning of IT change operations on large configurations for the first time.

First, we introduced extended partial-order reduction, a finer-grained version of the partial-order reduction model checking paradigm that enables the efficient verification of safety constraints given a set of concurrent, always-executable effects. Different to partial-order reduction, extended partial-order reduction does not require state commutativity and the stutter criterion of partial-order reduction to hold, but further assumptions about the characteristics of effects and atomic propositions have been introduced for the correctness of partial-order reduction. We showed that extended partial-order reduction is applicable in cases where partial-order reduction fails to further reduce the search space. Extended partial-order reduction was proven to be correct and theorems were provided to efficiently apply extended partial-order reduction to complex, compound formulas. To apply extended partial-order reduction to IT change verification, we introduced a many-sorted logic for IT change verification and proved its compliance with the theory of extended partial-order reduction. We introduced a prototypical implementation of an extended partial-order reduction model checker for IT change verification that solves IT change verification problems in linear time if change activities and safety constraints are roughly equally distributed over the Configuration Items of the CMDB. Finally, we showed that the change verification logic can be used in practice to model several static and dynamic routing network changes that could have caused a recent network outage in one of Amazon's data centers. The extended partial-order reduction model checker was evaluated against the state of the art general purpose model checkers NuSMV and SPiN in a total of 256 configurations of 32 benchmarks that differ in change operations to have caused Amazon's network outage, models describing the configuration of the CMDB, and model checker specific optimization techniques. The results showed that extended partial-order reduction significantly outperforms the model checkers NuSMV and SPiN with linear instead of polynomial (NuSMV) or exponential (SPiN) runtime complexity and that even its worst-case performance surpasses the best-case performance of NuSMV and SPiN. Furthermore, extended partial-order reduction was also shown to be more robust than the SPiN and NuSMV model checker - a significant advantage over NuSMV and SPiN - because their performance is more dependable on the modeling skills of the change manager.

Second, we addressed the problem to efficiently generate IT change plans for Request for Changes (RFCs), i.e., abstract goal specifications of IT changes. We examined four prominent planning paradigms (Hierarchical Task Network Planning (HTN) [76], Means End Analysis [94], Forward Chaining [10], and Planning Graphs [18]) in respect to their applicability to large-scale IT change planning and their usability by change managers. We found that Hierarchical Task Network planners (SHOP2) are, due to their rigorous decomposition concept and the hierarchical nature of IT change planning, the ones best suited for IT change planning in terms of performance and usability. However, even SHOP2 does not scale to large configurations due to the use of first-order unification algorithms. To enable IT change planning in the large, we proposed optimizations for decomposition-based IT change planning that significantly reduce the runtime complexity of a simple task decomposition planner from polynomial to linear or even constant runtime. Several micro benchmarks, which examine different characteristics of IT change activities, demonstrated the superior performance of the proposed optimization techniques. A three-tier application deployment and a network (un)configuration case study demonstrate the feasibility of the approach and confirm the results of the micro benchmarks: IT change planning has evolved from planning in the small to planning in the large.

In this work we laid the foundation to plan for and verify IT change operations on large configurations. Whether our algorithms can be applied in a practical setting with a broader set of IT changes than the ones addressed herein, is a question that concerns four aspects: The expressiveness, practicability, completeness, and capability of the approach to deal with uncertain management domains. In the following we discuss all four aspects and highlight directions for future work.

Expressiveness

To apply the change planning and verification algorithms to a larger set of change activities than the ones addressed herein, the change verification logic needs to be expressive enough to describe a broad spectrum of IT changes. While it might not always be possible to model a change verification or planning problem in the most direct way, there is - based on our experience - always a combination of infrastructure model and effects/predicates from the change verification logic that sufficiently describes a verification or planning problem. In the worst case one can always fall back to an abstract model that describes changes in an abstract way, for instance, a change might just change one property of a Configuration Item that describes whether the change has been applied to the Configuration Item or not. As we provided the precise requirements for extended partial-order reduction to be applicable (see Chapter 2) in this work, it is possible to extend the change verification logic with further predicates and effects should future research deem it appropriate.

Practicability

The systematic capture and specification of change activities, safety constraints, and IT infrastructure models remains an obstacle for the wider deployment of automated planners and verification tools. The experience gained through our experiments is that writing a domain can require significant work and is often difficult to do without prior basic knowledge in logic. We see two measures that can be undertaken to hide the complexity caused by the logical specification of change activities and safety constraints from the IT change manager:

- **Introduction of logic templates for IT change operations:** Previous work [33, 35] argues in favor of capturing existing knowledge in IT change design using plan templates. In a similar way, logic templates can be introduced that comprise the pre-defined specification of an IT change in change verification logic. Thus, a change manager only has to associate a change to a logic template and does not have to deal with its logical specification anymore. This approach works particularly well with hierarchical planners (such as our change planner discussed in Chapter 7) because higher-level change activities can be built of existing, more concrete change activities and a logical specification needs to be only provided for atomic change activities. However, providing the specification for the atomic change activities remains a challenge for the change manager. To avoid this, future research is necessary to automatically extract the preconditions and effects of atomic IT change activities from configuration snapshots taken before and after the application of a change activity. A similar approach was previously shown to work well to derive IT change templates [34].

- **Introduction of application and safety constraint templates:** Similar to the idea of change activity logic templates, which provide the pre-defined logical specification of IT change activities, typical configuration procedures could be captured in decomposition rules. Furthermore, safety constraints often used for software and IT systems could be provided as templates as well. Basic configuration constraints could be described independent of the concrete software or hardware product enabling its reuse.

Completeness

Finding and capturing the proper change activities and safety constraints to prevent an outage before it appears remains another obstacle to apply change verification. As Amazon's network outage already appeared, it was easy to formalize safety constraints and change activities in retrospective. To overcome this challenge, change verification and risk management need to be further integrated. Risk management [87, 96, 97] provides the means to detect risky change activities, which can then be chosen to be protected by means of verification.

Uncertainty

The uncertainty inherent to some infrastructures remains an obstacle for the deployment of automated planners and verification tools as well. The knowledge of a configuration can often be incomplete due to the existence of different management domains, due to the heterogeneity of applications, and due to the lack of tools to systematically record a configuration. In such environments, probabilistic model checking [64] is a promising approach to decide the likelihood of the violation of safety constraints. Furthermore, planning algorithms have been developed for uncertain planning domains as well [41]. It remains an open research question to what extent the certainty constraint can be relaxed without losing the runtime complexity necessary for large-scale IT change planning and verification.

APPENDIX A

Verification Theory

A.1 Proof of Proposition 2

This section presents the proof of Proposition 2 (see Section 2.3.4).

Proposition. ***Relationships among effects***
Let ϕ be a formula, then:

1. $NTRLs(\phi) \cup SUPPs(\phi) = PPs(\phi)$
2. $NTRLs(\phi) \cup THRTs(\phi) = NPs(\phi)$
3. $NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) = PPs(\phi) \cup NPs(\phi)$
4. $NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) \cup AMBGs(\phi) = E$
5. $THRTs(\phi) \cap NTRLs(\phi) = \emptyset$
6. $SUPPs(\phi) \cap NTRLs(\phi) = \emptyset$
7. $SUPPs(\phi) \cap THRTs(\phi) = \emptyset$

Proof. 1.

$$\begin{aligned} NTRLs(\phi) \cup SUPPs(\phi) \quad &=_{\text{Def. 3 (2.11), (2.9)}} \\ (PPs(\phi) \cap NPs(\phi)) \cup (PPs(\phi) \cap \overline{NPs(\phi)}) \quad &= \\ PPs(\phi) \quad & \end{aligned}$$

2.

$$\begin{aligned} NTRLs(\phi) \cup THRTs(\phi) &=_{\text{Def. 3 (2.11), (2.10)}} \\ (NPs(\phi) \cap PPs(\phi)) \cup (NPs(\phi) \cap \overline{PPs(\phi)}) &= \\ NPs(\phi) & \end{aligned}$$

3.

$$\begin{aligned} NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) &= \\ NTRLs(\phi) \cup SUPPs(\phi) \cup NTRLs(\phi) \cup THRTs(\phi) &=_{1.} \\ PPs(\phi) \cup NTRLs(\phi) \cup THRTs(\phi) &=_{2.} \\ PPs(\phi) \cup NPs(\phi) & \end{aligned}$$

4.

$$\begin{aligned} E &= \\ (PPs(\phi) \cup NPs(\phi)) \cup \overline{(PPs(\phi) \cup NPs(\phi))} &=_{3.} \\ NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) \cup \overline{(PPs(\phi) \cup NPs(\phi))} &=_{\text{Def. 3 (2.12)}} \\ NTRLs(\phi) \cup SUPPs(\phi) \cup THRTs(\phi) \cup AMBGs(\phi) & \end{aligned}$$

5.

$$\begin{aligned} e \in THRTs(\phi) &\rightarrow_{\text{Def. 3 (2.10)}} \\ e \in (NPs(\phi) \cap \overline{PPs(\phi)}) &\rightarrow \\ e \in NPs(\phi) \wedge e \notin PPs(\phi) &\rightarrow \\ e \notin (NPs(\phi) \cap PPs(\phi)) &\rightarrow_{\text{Def. 3 (2.11)}} \\ e \notin NTRLs(\phi) & \end{aligned}$$

Thus, $THRTs(\phi) \cap NTRLs(\phi) = \emptyset$.

6.

$$\begin{aligned} e \in SUPPs(\phi) &\rightarrow_{\text{Def. 3 (2.9)}} \\ e \in (PPs(\phi) \cap \overline{NPs(\phi)}) &\rightarrow \\ e \in PPs(\phi) \wedge e \notin NPs(\phi) &\rightarrow \\ e \notin (PPs(\phi) \cap NPs(\phi)) &\rightarrow_{\text{Def. 3 (2.11)}} \\ e \notin NTRLs(\phi) & \end{aligned}$$

Thus, $SUPPs(\phi) \cap NTRLs(\phi) = \emptyset$.

7.

$$\begin{aligned} e \in SUPPs(\phi) &\rightarrow_{\text{Def. 3 (2.9)}} \\ e \in (PPs(\phi) \cap \overline{NPs(\phi)}) &\rightarrow \\ e \notin NPs(\phi) &\rightarrow \\ e \notin (NPs(\phi) \cap \overline{PPs(\phi)}) &\rightarrow_{\text{Def. 3 (2.10)}} \\ e \notin THRTs(\phi) & \end{aligned}$$

Thus, $SUPPs(\phi) \cap THRTs(\phi) = \emptyset$.

□

A.2 Proof of Proposition 3

This section presents the proof of Proposition 3 (see Section 2.3.4).

Proposition. ***Relationships for negated formulas***
Let ϕ be a formula, then:

1. $PPs(\phi) = NPs(\neg\phi)$
2. $SUPPs(\phi) = THRTs(\neg\phi)$
3. $NTRLs(\phi) = NTRLs(\neg\phi)$
4. $AMBGs(\phi) = AMBGs(\neg\phi)$

Proof. 1.

$$\begin{aligned} e \in PPs(\phi) &\quad \leftrightarrow_{\text{Def. 1}} \\ \forall s \in S : (s \models \phi \rightarrow e(s) \models \phi) &\quad \leftrightarrow \\ \forall s \in S : (s \models \neg\neg\phi \rightarrow e(s) \models \neg\neg\phi) &\quad \leftrightarrow \\ \forall s \in S : (s \not\models \neg\phi \rightarrow e(s) \not\models \neg\phi) &\quad \leftrightarrow_{\text{Def. 2}} \\ e \in NPs(\neg\phi) & \end{aligned}$$

2.

$$\begin{aligned} SUPPs(\phi) &\quad =_{\text{Def. 3 (2.9)}} \\ PPs(\phi) \cap \overline{NPs(\phi)} &\quad =_{1.} \\ NPs(\neg\phi) \cap \overline{PPs(\neg\phi)} &\quad =_{\text{Def. 3 (2.10)}} \\ THRTs(\neg\phi) & \end{aligned}$$

3.

$$\begin{aligned} NTRLs(\phi) &\quad =_{\text{Def. 3 (2.11)}} \\ PPs(\phi) \cap NPs(\phi) &\quad =_{1.} \\ NPs(\neg\phi) \cap PPs(\neg\phi) &\quad =_{\text{Def. 3 (2.11)}} \\ NTRLs(\neg\phi) & \end{aligned}$$

4.

$$\begin{aligned} AMBGs(\phi) &\quad =_{\text{Def. 3 (2.12)}} \\ \overline{PPs(\phi) \cup NPs(\phi)} &\quad =_{1.} \\ \overline{NPs(\neg\phi) \cup PPs(\neg\phi)} &\quad =_{\text{Def. 3 (2.12)}} \\ AMBGs(\neg\phi) & \end{aligned}$$

□

A.3 Proof of Proposition 4

This section presents the proof of Proposition 4 (see Section 2.3.4).

Proposition. ***Support and threat complements***
Let $e \in E$ be an effect and ϕ a formula, then

1. *e decisively supports ϕ $\quad\leftrightarrow\quad$ e decisively threatens $\neg\phi$*
2. *e indecisively supports ϕ $\quad\leftrightarrow\quad$ e indecisively threatens $\neg\phi$*

Proof. 1. To prove:

$$(e \in SUPPs(\phi) \land e \text{ is a decisive PP effect of } \phi) \qquad \leftrightarrow$$
$$(e \in THRTs(\neg\phi) \land e \text{ is a decisive NP effect of } \neg\phi)$$

From Proposition 3, Statement (2) we obtain that $e \in SUPPs(\phi) \leftrightarrow e \in THRTs(\neg\phi)$. It remains to be shown that

$$(e \text{ is a decisive PP effect of } \phi) \leftrightarrow (e \text{ is a decisive NP effect of } \neg\phi)$$

$$\begin{aligned} e \text{ a decisive PP effect of } \phi &\quad \leftrightarrow_{\text{Def. 1}} \\ \forall s \in S : e(s) \models \phi &\quad \leftrightarrow \\ \forall s \in S : e(s) \not\models \neg\phi &\quad \leftrightarrow_{\text{Def. 2}} \\ e \text{ is a decisive NP effect of } \neg\phi & \end{aligned}$$

2. To prove:

$$(e \in SUPPs(\phi) \land e \text{ is an indecisive PP effect of } \phi) \qquad \leftrightarrow$$
$$(e \in THRTs(\neg\phi) \land e \text{ is an indecisive NP effect of } \neg\phi)$$

From Proposition 3, Statement (2) we obtain that $e \in SUPPs(\phi) \leftrightarrow e \in THRTs(\neg\phi)$. It remains to be shown that

$$(e \text{ is an indecisive PP effect of } \phi) \leftrightarrow (e \text{ is an indecisive NP effect of } \neg\phi)$$

$$\begin{aligned} e \text{ is an indecisive PP effect of } \phi &\quad \leftrightarrow_{\text{Def. 1}} \\ \exists s \in S : e(s) \not\models \phi &\quad \leftrightarrow \\ \exists s \in S : e(s) \models \neg\phi &\quad \leftrightarrow_{\text{Def. 2}} \\ e \text{ is an indecisive NP effect of } \neg\phi & \end{aligned}$$

□

A.4 Proof of Proposition 11

This section presents the complete proof of Proposition 11 (see Section 3.4.1) stating that all support, threat, and neutral relationships among effects and predicates of the change verification language are correct.

Proof. Every entry in Table 3.3a is proven separately. We denote by p_1, and p_2 integer constants in Σ^P and by Δc_1, Δc_2 fixed integer constants in Σ^C.

1. To prove:

$$inc(p_1, \Delta c_1) \text{ indecisively supports } p_1 \geq p_2 \text{ and } p_1 > p_2 \quad \leftrightarrow_{\text{Def. 3}}$$
$$\underbrace{inc(p_1, \Delta c_1) \in PPs(p_1 \geq p_2)}_{(1)} \quad \wedge \quad \underbrace{inc(p_1, \Delta c_1) \notin NPs(p_1 \geq p_2)}_{(2)}$$

We only proof this case for $p_1 \geq p_2$. The proof for $p_1 > p_2$ is done analogously.

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs : \quad cmdb \models p_1 \geq p_2 \rightarrow inc(p_1, \Delta c_1)(cmdb) \models p_1 \geq p_2$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad \geq^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \rightarrow \quad \geq^{def}(p_1^{inc(p_1,\Delta c_1)(cmdb)}, p_2^{inc(p_1,\Delta c_1)(cmdb)})$$

Let Δc_1^{def} be the absolute increase to p_1^{cmdb} induced by the application of $inc(p_1, \Delta c_1)$ to $cmdb$. Then,

$$\begin{aligned} p_2^{inc(p_1,\Delta c_1)(cmdb)} &= p_2^{cmdb} \quad \text{and} \\ p_1^{inc(p_1,\Delta c_1)(cmdb)} &= p_1^{cmdb} + \Delta c_1^{def} \end{aligned}$$

Thus, it remains to be proven that:

$$\forall cmdb \in CMDBs : \quad \geq^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \rightarrow \quad \geq^{def}(p_1^{cmdb} + \Delta c_1^{def}, p_2^{cmdb})$$

which is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\exists cmdb \in CMDBs : \quad (cmdb \not\models p_1 \geq p_2 \quad \wedge \quad inc(p_1, \Delta c_1)(cmdb) \models p_1 \geq p_2)$$

This is the case iff (Def. 15)

$$\exists cmdb \in CMDBs : \quad <^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \wedge \quad \geq^{def}(p_1^{inc(p_1,\Delta c_1)(cmdb)}, p_2^{inc(p_1,\Delta c_1)(cmdb)})$$

Let $p_1^{cmdb} = 0$ and $p_2^{cmdb} = 1$ and $\Delta c_1^{def} = 2$. Then, we obtain with $p_1^{inc(p_1,\Delta c_1)(cmdb)} = 2$:

$$<^{def}(0, 1) \quad \wedge \quad \geq^{def}(2, 1)$$

which is trivially satisfied. Thus, $inc(p_1, \Delta c_1) \notin NPs(p_1 \geq p_2)$.

$inc(p_1, \Delta c_1)$ is an indecisive PP effect because for $p_1^{cmdb} = 2$ and $p_2^{cmdb} = 5$ and $\Delta c_1^{def} = 2$, we obtain

$$\begin{aligned} p_1^{inc(p_1,\Delta c_1)(cmdb)} &= p_1^{cmdb} + \Delta c_1^{def} = 2 + 2 = 4 \\ p_2^{inc(p_1,\Delta c_1)(cmdb)} &= p_2^{cmdb} = 5 \end{aligned}$$

but $p_1^{inc(p_1,\Delta c_1)(cmdb)} \ngeq p_2^{inc(p_1,\Delta c_1)(cmdb)}$.

Consequently, $inc(p_1, \Delta c_1)$ is not a threat or neutral effect to $p_1 \geq p_2$ (Proposition 2, (6) and (7)).

2. To prove:

$$inc(p_1, \Delta c_1) \text{ indecisively threatens } p_1 \leq p_2 \text{ and } p_1 < p_2$$

This can be directly concluded from Statement (1):

$$\begin{aligned} \text{Statement (1)} &\equiv \\ inc(p_1, \Delta c_1) \text{ indecisively supports } p_1 \geq p_2 \text{ and } p_1 > p_2 &\equiv_{\text{Proposition 4}} \\ inc(p_1, \Delta c_1) \text{ indecisively threatens } p_1 \ngeq p_2 \text{ and } p_1 \ngtr p_2 &\equiv \\ inc(p_1, \Delta c_1) \text{ indecisively threatens } p_1 < p_2 \text{ and } p_1 \leq p_2 & \end{aligned}$$

3. To prove:

$$inc(p_2, \Delta c_2) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2$$

This can be directly concluded from Statement (1):

$$\begin{aligned} \text{Statement (1)} &\equiv \\ inc(p_1, \Delta c_1) \text{ indecisively supports } p_1 \geq p_2 \text{ and } p_1 > p_2 &\equiv_{\text{Substitution: } p_2 \leftrightarrow p_1 \text{ and } \Delta c_2 \leftrightarrow \Delta c_1} \\ inc(p_2, \Delta c_2) \text{ indecisively supports } p_2 \geq p_1 \text{ and } p_2 > p_1 &\equiv \\ inc(p_2, \Delta c_2) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2 & \end{aligned}$$

4. To prove:

$$inc(p_2, \Delta c_2) \text{ indecisively threatens } p_1 \geq p_2 \text{ and } p_1 > p_2$$

This can be directly concluded from Statement (3):

$$\begin{aligned} \text{Statement (3)} &\equiv \\ inc(p_2, \Delta c_2) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2 &\equiv_{\text{Proposition 4}} \\ inc(p_2, \Delta c_2) \text{ indecisively threatens } p_1 \nleq p_2 \text{ and } p_1 \nless p_2 &\equiv \\ inc(p_2, \Delta c_2) \text{ indecisively threatens } p_1 > p_2 \text{ and } p_1 \geq p_2 & \end{aligned}$$

5. To prove:

$$\begin{aligned} dec(p_2, \Delta c_2) \text{ indecisively supports } p_1 \geq p_2 (\text{ and } p_1 > p_2) &\leftrightarrow_{\text{Def. 3}} \\ \underbrace{dec(p_2, \Delta c_2) \in PPs(p_1 \geq p_2)}_{(1)} \quad \wedge \quad \underbrace{dec(p_2, \Delta c_2) \notin NPs(p_1 \geq p_2)}_{(2)} & \end{aligned}$$

We only proof this case for $p_1 \geq p_2$. The proof for $p_1 > p_2$ is done analogously.

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs : \quad cmdb \models p_1 \geq p_2 \rightarrow dec(p_2, \Delta c_2)(cmdb) \models p_1 \geq p_2$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad \geq^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \rightarrow \quad \geq^{def}(p_1^{dec(p_2,\Delta c_2)(cmdb)}, p_2^{dec(p_2,\Delta c_2)(cmdb)})$$

Let Δc_2^{def} be the absolute decrease to p_2^{cmdb} induced by the application of $dec(p_2, \Delta c_2)$ to $cmdb$. Then,

$$\begin{aligned} p_2^{dec(p_2,\Delta c_2)(cmdb)} &= p_2^{cmdb} - \Delta c_2^{def} \quad \text{and} \\ p_1^{dec(p_2,\Delta c_2)(cmdb)} &= p_1^{cmdb} \end{aligned}$$

Thus, it remains to be proven that:

$$\forall cmdb \in CMDBs : \quad \geq^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \rightarrow \quad \geq^{def}(p_1^{cmdb}, p_2^{cmdb} - \Delta c_2^{def})$$

which is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\exists cmdb \in CMDBs : \quad (cmdb \not\models p_1 \geq p_2 \quad \wedge \quad dec(p_2, \Delta c_2)(cmdb) \models p_1 \geq p_2)$$

This is the case iff (Def. 15)

$$\exists cmdb \in CMDBs : \quad <^{def}(p_1^{cmdb}, p_2^{cmdb}) \quad \wedge \quad \geq^{def}(p_1^{dec(p_2,\Delta c_2)(cmdb)}, p_2^{dec(p_2,\Delta c_2)(cmdb)})$$

Let $p_1^{cmdb} = 2$ and $p_2^{cmdb} = 3$ and $\Delta c_2^{def} = 2$. Then, we obtain with $p_2^{dec(p_2,\Delta c_2)(cmdb)} = 1$:

$$<^{def}(2, 3) \quad \wedge \quad \geq^{def}(2, 1)$$

which is trivially satisfied. Thus, $dec(p_2, \Delta c_2) \notin NPs(p_1 \geq p_2)$.

$dec(p_2, \Delta c_2)$ is an indecisive PP effect because for $p_1^{cmdb} = 2$ and $p_2^{cmdb} = 4$ and $\Delta c_2^{def} = 1$, we obtain

$$\begin{aligned} p_1^{dec(p_2,\Delta c_2)(cmdb)} &= p_1^{cmdb} = 2 \\ p_2^{dec(p_2,\Delta c_2)(cmdb)} &= p_2^{cmdb} - \Delta c_2^{def} = 4 - 1 = 3 \end{aligned}$$

but $p_1^{dec(p_2,\Delta c_2)(cmdb)} \ngeq p_2^{dec(p_2,\Delta c_2)(cmdb)}$.

Consequently, $dec(p_2, \Delta c_2)$ is not a threat or neutral effect to $p_1 \geq p_2$ (Proposition 2, (6) and (7)).

6. To prove:

$$dec(p_2, \Delta c_2) \text{ indecisively threatens } p_1 \leq p_2 \text{ and } p_1 < p_2$$

This can be directly concluded from Statement (5):

$$\begin{aligned} \text{Statement (5)} &\equiv \\ dec(p_2, \Delta c_2) \text{ indecisively supports } p_1 \geq p_2 \text{ and } p_1 > p_2 &\equiv_{\text{Proposition 4}} \\ dec(p_2, \Delta c_2) \text{ indecisively threatens } p_1 \ngeq p_2 \text{ and } p_1 \ngtr p_2 &\equiv \\ dec(p_2, \Delta c_2) \text{ indecisively threatens } p_1 < p_2 \text{ and } p_1 \leq p_2 & \end{aligned}$$

7. To prove:

$$dec(p_1, \Delta c_1) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2$$

This can be directly concluded from Statement (5):

$$\begin{aligned} \text{Statement (5)} &\equiv \\ dec(p_2, \Delta c_2) \text{ indecisively supports } p_1 \geq p_2 \text{ and } p_1 > p_2 &\equiv_{\text{Substitution: } p_2 \leftrightarrow p_1 \text{ and } \Delta c_2 \leftrightarrow \Delta c_1} \\ dec(p_1, \Delta c_1) \text{ indecisively supports } p_2 \geq p_1 \text{ and } p_2 > p_1 &\equiv \\ dec(p_1, \Delta c_1) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2 & \end{aligned}$$

8. To prove:

$$dec(p_1, \Delta c_1) \text{ indecisively threatens } p_1 \geq p_2 \text{ and } p_1 > p_2$$

This can be directly concluded from Statement (7):

$$\begin{aligned} \text{Statement (7)} &\equiv \\ dec(p_1, \Delta c_1) \text{ indecisively supports } p_1 \leq p_2 \text{ and } p_1 < p_2 &\equiv_{\text{Proposition 4}} \\ dec(p_1, \Delta c_1) \text{ indecisively threatens } p_1 \nleq p_2 \text{ and } p_1 \nless p_2 &\equiv \\ dec(p_1, \Delta c_1) \text{ indecisively threatens } p_1 > p_2 \text{ and } p_1 \geq p_2 & \end{aligned}$$

Every entry in Table 3.3b is proven separately. We denote by p a constant in Σ^P and by c_1, c_2 fixed-constants in Σ^C with values $c_1^{def} \neq c_2^{def}$.

1. To prove:

$$\begin{aligned} set(p, c_1) \text{ decisively supports } hasValue(p, c_1) &\leftrightarrow_{\text{Def. 3}} \\ \underbrace{set(p, c_1) \in PPs(hasValue(p, c_1))}_{(1)} \quad \wedge \quad \underbrace{set(p, c_1) \notin NPs(hasValue(p, c_1))}_{(2)} & \end{aligned}$$

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs: \quad cmdb \models hasValue(p, c_1) \quad \rightarrow \quad set(p, c_1)(cmdb) \models hasValue(p, c_1)$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs: \quad hasValue^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_1^{def})$$

Because $p^{set(p,c_1)(cmdb)} = c_1^{def}$, this is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\exists cmdb \in CMDBs: \quad (cmdb \not\models hasValue(p, c_1) \wedge set(p, c_1)(cmdb) \models hasValue(p, c_1))$$

This is the case iff (Def. 15)

$$\exists cmdb \in CMDBs: \quad \neg hasValue^{def}(p^{cmdb}, c_1^{def}) \quad \wedge \quad hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_1^{def})$$

Let $p^{cmdb} = c_2^{def}$ and $c_2^{def} \neq c_1^{def}$, then

$$\begin{aligned} \neg hasValue^{def}(p^{cmdb}, c_1^{def}) &\equiv true \qquad \text{and} \\ hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_1^{def}) &\equiv true \end{aligned}$$

because $p^{set(p,c_1)(cmdb)} = c_1^{def}$. Consequently, $set(p, c_1)$ is a decisive support and neither a threat nor neutral effect to $hasValue(p, c_1)$ (Proposition 2, (6) and (7)).

2. To prove:

$$set(p, c_2) \text{ decisively threatens } \neg hasValue(p, c_2)$$

This can be directly concluded from Statement (1):

$$\begin{aligned} \text{Statement (1)} &\equiv \\ set(p, c_2) \text{ decisively supports } hasValue(p, c_2) &\equiv_{\text{Proposition 4}} \\ set(p, c_2) \text{ decisively threatens } \neg hasValue(p, c_2) & \end{aligned}$$

3. To prove:

$$\begin{aligned} set(p, c_1) \text{ decisively supports } \neg hasValue(p, c_2) \quad &\leftrightarrow_{\text{Def. 3}} \\ \underbrace{set(p, c_1) \in PPs(\neg hasValue(p, c_2))}_{(1)} \quad \wedge \quad \underbrace{set(p, c_1) \notin NPs(\neg hasValue(p, c_2))}_{(2)} & \end{aligned}$$

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs : \quad cmdb \models \neg hasValue(p, c_2) \rightarrow set(p, c_1)(cmdb) \models \neg hasValue(p, c_2)$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad \neg hasValue^{def}(p^{cmdb}, c_2^{def}) \quad \rightarrow \quad \neg hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_2^{def})$$

Because $p^{set(p,c_1)(cmdb)} = c_1^{def} \neq c_2^{def}$, this is trivially satisfied.
Proof of (2): To prove (Def. 2):

$$\exists cmdb \in CMDBs : \quad (cmdb \not\models \neg hasValue(p, c_2) \wedge set(p, c_1)(cmdb) \models \neg hasValue(p, c_2))$$

This is the case iff (Def. 15)

$$\exists cmdb \in CMDBs : \quad hasValue^{def}(p^{cmdb}, c_2^{def}) \quad \wedge \quad \neg hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_2^{def})$$

Let $p^{cmdb} = c_2^{def}$ and $c_2^{def} \neq c_1^{def}$, then

$$\begin{aligned} hasValue^{def}(p^{cmdb}, c_2^{def}) &\equiv \quad true \qquad \text{and} \\ \neg hasValue^{def}(p^{set(p,c_1)(cmdb)}, c_2^{def}) &\equiv \quad true \end{aligned}$$

because $p^{set(p,c_1)(cmdb)} = c_1^{def} \neq c_2^{def}$. Consequently, $set(p, c_1)$ is a decisive support and neither a threat nor neutral effect to $\neg hasValue(p, c_2)$ (Proposition 2, (6) and (7)).

4. To prove:

$$set(p, c_1) \text{ decisively threatens } hasValue(p, c_2)$$

This can be directly concluded from Statement (3):

$$\begin{aligned} \text{Statement (3)} &\equiv \\ set(p, c_1) \text{ decisively supports } \neg hasValue(p, c_2) &\equiv_{\text{Proposition 4}} \\ set(p, c_1) \text{ decisively threatens } hasValue(p, c_2) & \end{aligned}$$

Every entry in Table 3.3c is proven separately. We denote by p a list or set constant in Σ^P and by c_1, c_2 fixed-constants in Σ^C that describe elements of list or set p.

1. To prove:

$$add(p, c_1) \text{ decisively supports } contains(p, c_1) \text{ on lists and sets} \quad \leftrightarrow_{\text{Def. 3}}$$
$$\underbrace{add(p, c_1) \in PPs(contains(p, c_1))}_{(1)} \quad \wedge \quad \underbrace{add(p, c_1) \notin NPs(contains(p, c_1))}_{(2)}$$

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs : \quad cmdb \models contains(p, c_1) \quad \rightarrow \quad add(p, c_1)(cmdb) \models contains(p, c_1)$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad contains^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad contains^{def}(p^{add(p,c_1)(cmdb)}, c_1^{def})$$

For any configuration $p^{add(p,c_1)(cmdb)}$ comprises at least one instance of c_1^{def} (if p is a list) or exactly one instance of c_1^{def} (if p is a set) independently of the value of p^{cmdb} . Thus, the implication is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\exists cmdb \in CMDBs : \quad (cmdb \not\models contains(p, c_1) \quad \wedge \quad add(p, c_1)(cmdb) \models contains(p, c_1))$$

This is the case iff (Def. 15)

$$\exists cmdb \in CMDBs : \quad \neg contains^{def}(p^{cmdb}, c_1^{def}) \quad \wedge \quad contains^{def}(p^{add(p,c_1)(cmdb)}, c_1^{def})$$

Let $p^{cmdb} = \emptyset$, then

$$\neg contains^{def}(p^{cmdb}, c_1^{def}) \quad \equiv \quad true \qquad \text{and}$$
$$contains^{def}(p^{add(p,c_1)(cmdb)}, c_1^{def}) \quad \equiv \quad true$$

because $p^{cmdb} = \emptyset$ does not comprise an instance of c_1^{def} and $p^{add(p,c_1)(cmdb)} = \{c_1^{def}\}$ comprises one instance of c_1^{def}.

$add(p, c_1)$ is a decisive PP effect because

$$\forall cmdb \in CMDBs : \quad add(p, c_1)(cmdb) \models contains(p, c_1).$$

Consequently, $add(p, c_1)$ is not a threat or neutral effect to $contains(p, c_1)$ (Proposition 2, (6) and (7)).

2. To prove:

$$add(p, c_1) \text{ decisively threatens } \neg contains(p, c_1) \text{ on lists and sets}$$

This can be directly concluded from Statement (1):

$$\begin{aligned} \text{Statement (1)} &\equiv \\ add(p, c_1) \text{ decisively supports } contains(p, c_1) \text{ on lists and sets} &\equiv_{\text{Proposition 4}} \\ add(p, c_1) \text{ decisively threatens } \neg contains(p, c_1) \text{ on lists and sets} & \end{aligned}$$

3. To prove:

$$\begin{aligned}&\mathit{remove}(p, c_1) \text{ decisively (indecisively) supports } \neg\mathit{contains}(p, c_1) \text{ on sets (lists)}\\ &\leftrightarrow_{\text{Def. 3}}\\ &\underbrace{\mathit{remove}(p, c_1) \in \mathit{PPs}(\neg\mathit{contains}(p, c_1))}_{(1)} \quad \wedge \quad \underbrace{\mathit{remove}(p, c_1) \notin \mathit{NPs}(\neg\mathit{contains}(p, c_1))}_{(2)}\end{aligned}$$

Proof of (1): To prove (Def. 1):

$$\begin{aligned}&\forall \mathit{cmdb} \in \mathit{CMDBs}:\\ &\mathit{cmdb} \models \neg\mathit{contains}(p, c_1) \quad \rightarrow \quad \mathit{remove}(p, c_1)(\mathit{cmdb}) \models \neg\mathit{contains}(p, c_1)\end{aligned}$$

This is the case iff (Def. 15)

$$\forall \mathit{cmdb} \in \mathit{CMDBs}: \quad \neg\mathit{contains}^{\mathit{def}}(p^{\mathit{cmdb}}, c_1^{\mathit{def}}) \quad \rightarrow \quad \neg\mathit{contains}^{\mathit{def}}(p^{\mathit{remove}(p,c_1)(\mathit{cmdb})}, c_1^{\mathit{def}})$$

By assumption p^{cmdb} does not contain an instance of c_1^{def}. Consequently, $p^{\mathit{remove}(p,c_1)(\mathit{cmdb})}$ does not comprise an instance of c_1^{def} as well. Thus, the implication is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\begin{aligned}&\exists \mathit{cmdb} \in \mathit{CMDBs}:\\ &(\mathit{cmdb} \not\models \neg\mathit{contains}(p, c_1) \quad \wedge \quad \mathit{remove}(p, c_1)(\mathit{cmdb}) \models \neg\mathit{contains}(p, c_1))\end{aligned}$$

This is the case iff (Def. 15)

$$\exists \mathit{cmdb} \in \mathit{CMDBs}: \quad \mathit{contains}^{\mathit{def}}(p^{\mathit{cmdb}}, c_1^{\mathit{def}}) \quad \wedge \quad \neg\mathit{contains}^{\mathit{def}}(p^{\mathit{remove}(p,c_1)(\mathit{cmdb})}, c_1^{\mathit{def}})$$

Let $p^{\mathit{cmdb}} = \{c_1^{\mathit{def}}\}$, then

$$\begin{aligned}\mathit{contains}^{\mathit{def}}(p^{\mathit{cmdb}}, c_1^{\mathit{def}}) &\equiv \mathit{true} \qquad \text{and}\\ \neg\mathit{contains}^{\mathit{def}}(p^{\mathit{remove}(p,c_1)(\mathit{cmdb})}, c_1^{\mathit{def}}) &\equiv \mathit{true}\end{aligned}$$

because $p^{\mathit{cmdb}} = \{c_1^{\mathit{def}}\}$ comprises c_1^{def} and $p^{\mathit{remove}(p,c_1)(\mathit{cmdb})} = \emptyset$ does not comprise c_1^{def}.

remove(p, c_1) is a decisive support (PP effect) on sets because for a set p:

$$\forall \mathit{cmdb} \in \mathit{CMDBs}: \quad \mathit{remove}(p, c_1)(\mathit{cmdb}) \models \neg\mathit{contains}(p, c_1).$$

However, on lists *remove*(p, c_1) is an indecisive support (PP effect) because, for example, for $p^{\mathit{cmdb}} = \{c_1^{\mathit{def}}, c_1^{\mathit{def}}\}$ we obtain:

$$\mathit{remove}(p, c_1)(\mathit{cmdb}) \models \mathit{contains}(p, c_1)$$

Consequently, *remove*(p, c_1) is not a threat or neutral effect to $\neg$*contains*(p, c_1) (Proposition 2, (6) and (7)).

4. To prove:

   $$\textit{remove}(p, c_1) \text{ decisively (indecisively) threatens } \neg\textit{contains}(p, c_1) \text{ on sets (lists)}$$

   This can be directly concluded from Statement (3):

   $$\begin{aligned} \text{Statement (1)} &\equiv \\ \textit{remove}(p, c_1) \text{ decisively (indec.) supports } \neg\textit{contains}(p, c_1) \text{ on sets (lists)} &\equiv_{\text{Proposition 4}} \\ \textit{remove}(p, c_1) \text{ decisively (indec.) threatens } \textit{contains}(p, c_1) \text{ on sets (lists)} & \end{aligned}$$

5. To prove (for $c_1^{def} \neq c_2^{def}$):

   $$\textit{remove}(p, c_2) \text{ is a neutral effect in respect to } \textit{contains}(p, c_1) \text{ on lists and sets}$$
   $$\leftrightarrow_{\text{Def. 3}}$$
   $$\underbrace{\textit{remove}(p, c_2) \in \textit{PPs}(\textit{contains}(p, c_1))}_{(1)} \quad \wedge \quad \underbrace{\textit{remove}(p, c_2) \in \textit{NPs}(\textit{contains}(p, c_1))}_{(2)}$$

   Proof of (1): To prove (Def. 1):

   $$\begin{aligned} &\forall \textit{cmdb} \in \textit{CMDBs}: \\ &\textit{cmdb} \models \textit{contains}(p, c_1) \quad \rightarrow \quad \textit{remove}(p, c_2)(\textit{cmdb}) \models \textit{contains}(p, c_1) \end{aligned}$$

   This is the case iff (Def. 15)

   $$\forall \textit{cmdb} \in \textit{CMDBs}: \quad \textit{contains}^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad \textit{contains}^{def}(p^{remove(p,c_2)(cmdb)}, c_1^{def})$$

   The number of instances of c_1^{def} is the same in p^{cmdb} and $p^{remove(p,c_2)(cmdb)}$ for any configuration *cmdb*. Thus, the implication is trivially satisfied.

   Proof of (2): To prove (Def. 2):

   $$\begin{aligned} &\forall \textit{cmdb} \in \textit{CMDBs}: \\ &\textit{cmdb} \not\models \textit{contains}(p, c_1) \quad \rightarrow \quad \textit{remove}(p, c_2)(\textit{cmdb}) \not\models \textit{contains}(p, c_1) \end{aligned}$$

   This is the case iff (Def. 15)

   $$\forall \textit{cmdb} \in \textit{CMDBs}: \quad \neg\textit{contains}^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad \neg\textit{contains}^{def}(p^{remove(p,c_2)(cmdb)}, c_1^{def})$$

   For any configuration *cmdb*, the number of instances of c_1^{def} is the same in p^{cmdb} and $p^{remove(p,c_2)(cmdb)}$. Thus, the implication is trivially satisfied.

6. To prove:

   $$\textit{remove}(p, c_2) \text{ is a neutral effect in respect to } \neg\textit{contains}(p, c_1) \text{ on lists and sets}$$

   This can be directly concluded from Statement (5):

   $$\begin{aligned} \text{Statement (5)} &\equiv \\ \textit{remove}(p, c_2) \text{ neutrals } \textit{contains}(p, c_1) \text{ on lists and sets} &\equiv_{\text{Proposition 3, Statement (3)}} \\ \textit{remove}(p, c_2) \text{ neutrals } \neg\textit{contains}(p, c_1) \text{ on lists and sets} & \end{aligned}$$

7. To prove (for $c_1^{def} \neq c_2^{def}$):

$$add(p, c_2) \text{ is a neutral effect in respect to } contains(p, c_1) \text{ on lists and sets}$$

$$\leftrightarrow_{\text{Def. 3}}$$

$$\underbrace{add(p, c_2) \in PPs(contains(p, c_1))}_{(1)} \quad \wedge \quad \underbrace{add(p, c_2) \in NPs(contains(p, c_1))}_{(2)}$$

Proof of (1): To prove (Def. 1):

$$\forall cmdb \in CMDBs : \quad cmdb \models contains(p, c_1) \quad \rightarrow \quad add(p, c_2)(cmdb) \models contains(p, c_1)$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad contains^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad contains^{def}(p^{add(p,c_2)(cmdb)}, c_1^{def})$$

The number of instances of c_1^{def} is the same in p^{cmdb} and $p^{add(p,c_2)(cmdb)}$ for any configuration *cmdb*. Thus, the implication is trivially satisfied.

Proof of (2): To prove (Def. 2):

$$\forall cmdb \in CMDBs : \quad cmdb \not\models contains(p, c_1) \quad \rightarrow \quad add(p, c_2)(cmdb) \not\models contains(p, c_1)$$

This is the case iff (Def. 15)

$$\forall cmdb \in CMDBs : \quad \neg contains^{def}(p^{cmdb}, c_1^{def}) \quad \rightarrow \quad \neg contains^{def}(p^{add(p,c_2)(cmdb)}, c_1^{def})$$

The number of instances of c_1^{def} is the same in p^{cmdb} and $p^{add(p,c_2)(cmdb)}$ for any configuration *cmdb*. Thus, the implication is trivially satisfied.

8. To prove:

$$add(p, c_2) \text{ is a neutral effect in respect to } \neg contains(p, c_1) \text{ on lists and sets}$$

This can be directly concluded from Statement (7):

$$\begin{aligned} \text{Statement (7)} \quad & \equiv \\ add(p, c_2) \text{ neutrals } contains(p, c_1) \text{ on lists and sets} \quad & \equiv_{\text{Proposition 3, Statement (3)}} \\ add(p, c_2) \text{ neutrals } \neg contains(p, c_1) \text{ on lists and sets} \quad & \end{aligned}$$

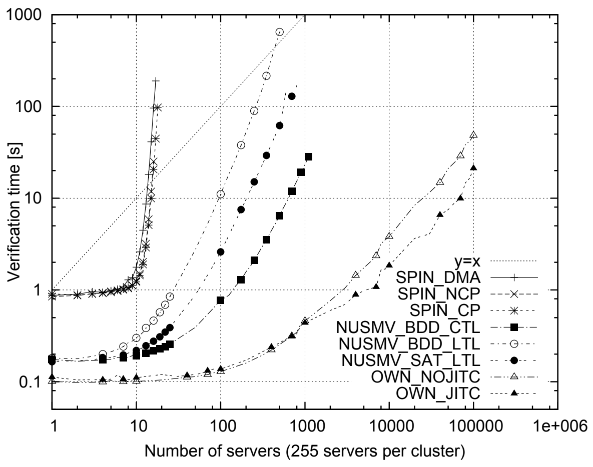
□

APPENDIX B

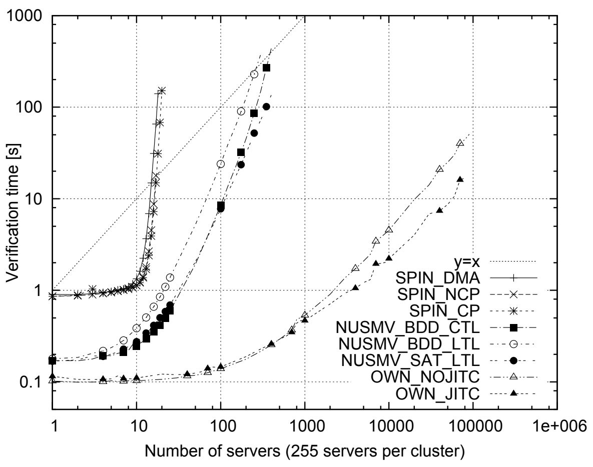
Verification Benchmarks

B.1 Verification Time Depending on Model, CMDB Size, and Change Workload

B.1.1 Model 1

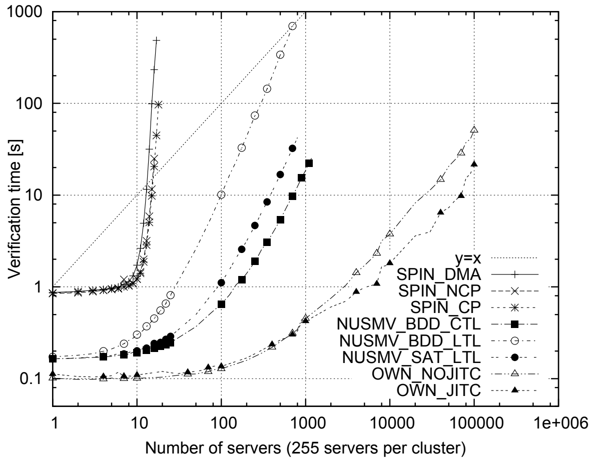


(a) *SHTp* workload on Model$_1$.

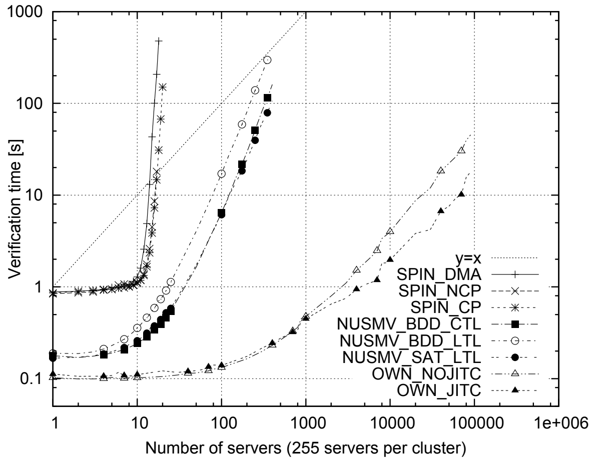


(b) *SHTn* workload on Model$_1$.

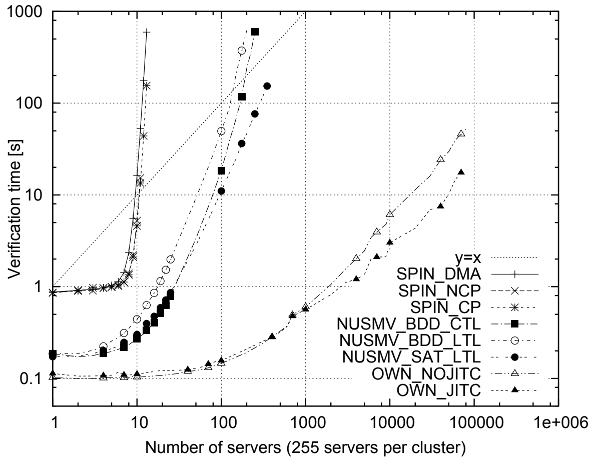
Figure B.1: Time to verify different workloads on Model$_1$ depending on the number of servers, the model checker, and optimization technique used.



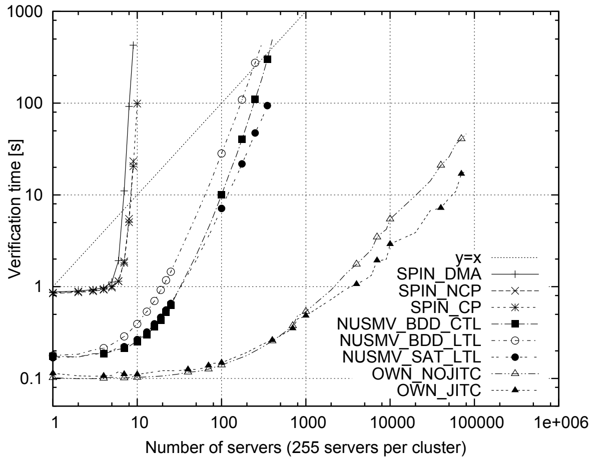
(a) *FOp* workload on Model$_1$.



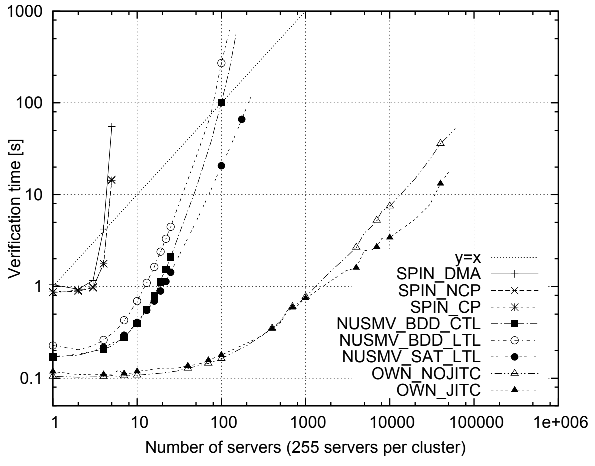
(b) *FOn* workload on Model$_1$.



(c) *SHTp* & *SHTn* workload on Model$_1$.



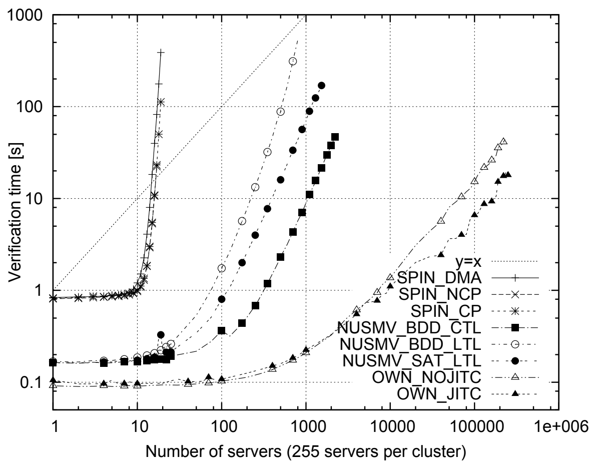
(d) *FOp* & *FOn* workload on Model$_1$.



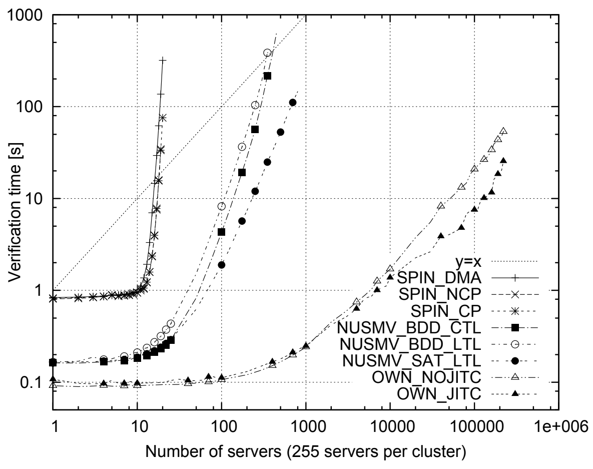
(e) *SHTp* & *SHTn* & *FOp* & *FOn* workload on Model$_1$.

Figure B.2: Time to verify different workloads on Model$_1$ depending on the number of servers, the model checker, and optimization technique used.

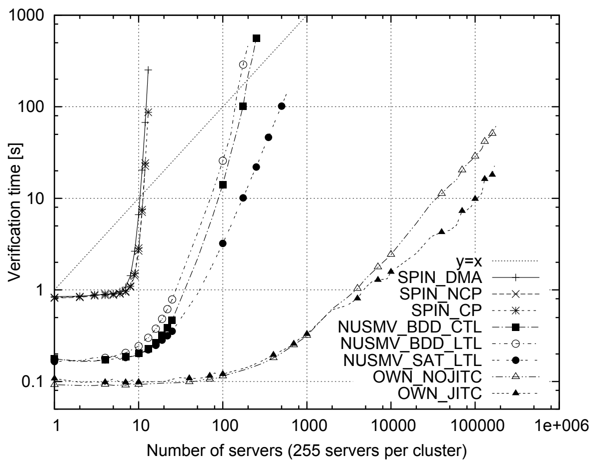
B.1.2 Model 2



(a) *SHTp* workload on Model_2.



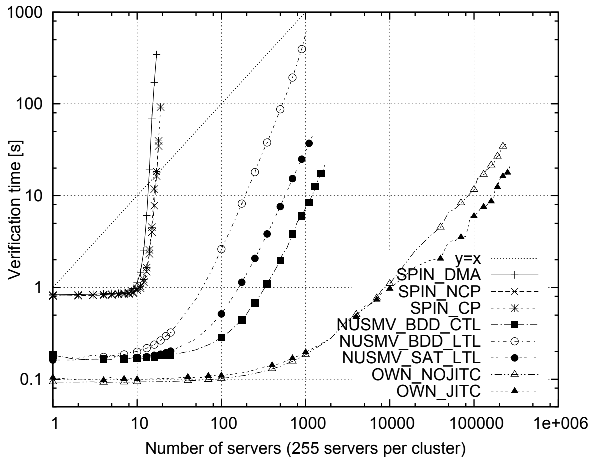
(b) *SHTn* workload on Model_2.



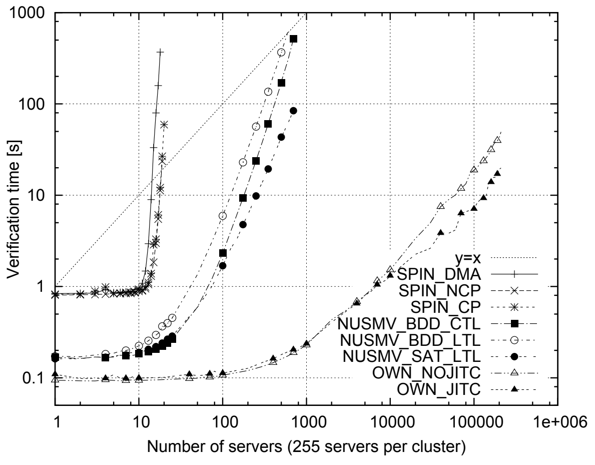
(c) *SHTp* & *SHTn* workload on Model_2.

Figure B.3: Time to verify different workloads on Model_2 depending on the number of servers, the model checker, and optimization technique used.

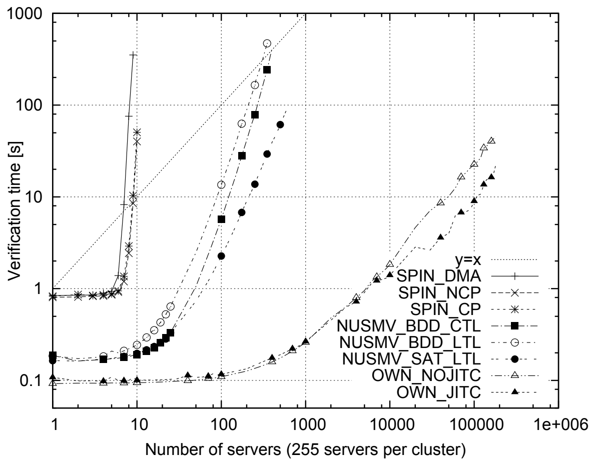
B.1.3 Model 3



(a) *FOp* workload on Model$_3$.



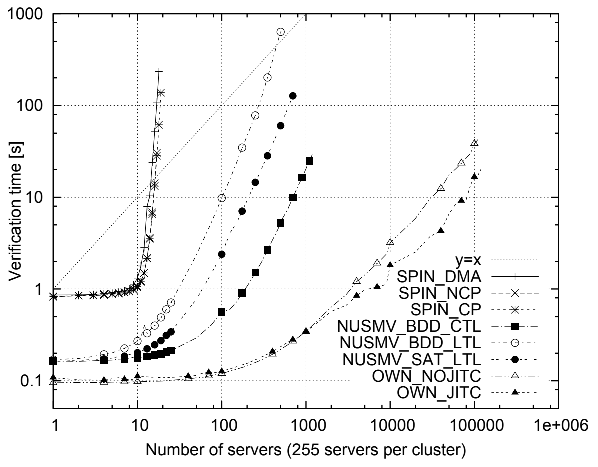
(b) *FOn* workload on Model$_3$.



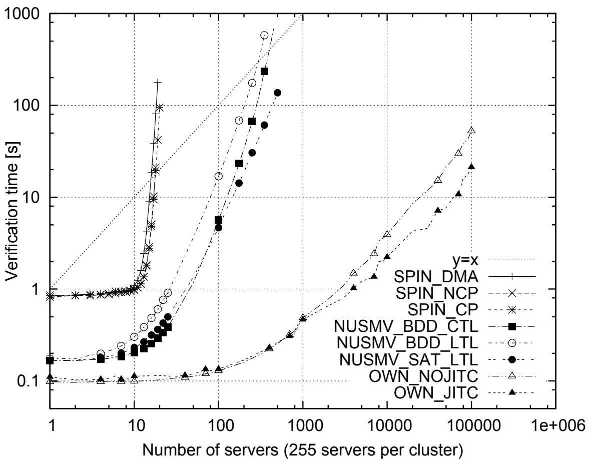
(c) *FOp* & *FOn* workload on Model$_3$.

Figure B.4: Time to verify different workloads on Model$_3$ depending on the number of servers, the model checker, and optimization technique used.

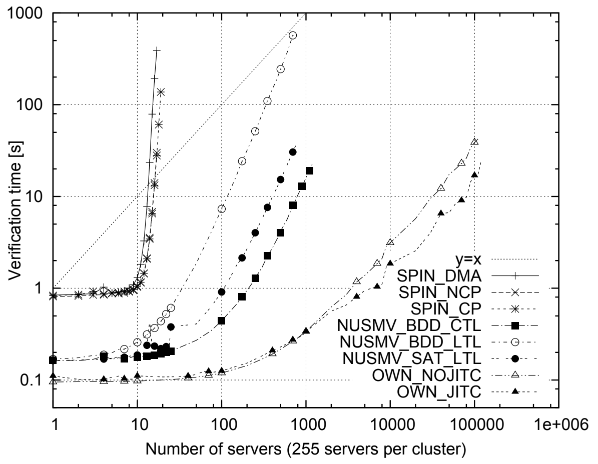
B.1.4 Model 4



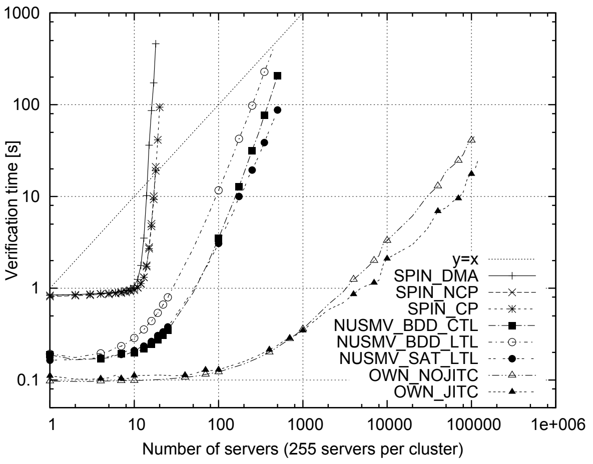
(a) *SHTp* workload on Model$_4$.



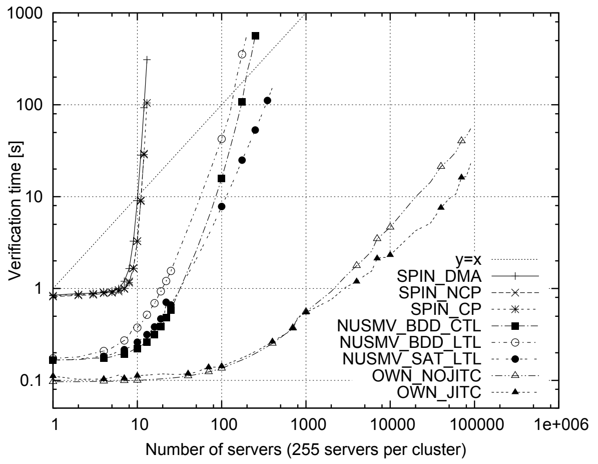
(b) *SHTn* workload on Model$_4$.



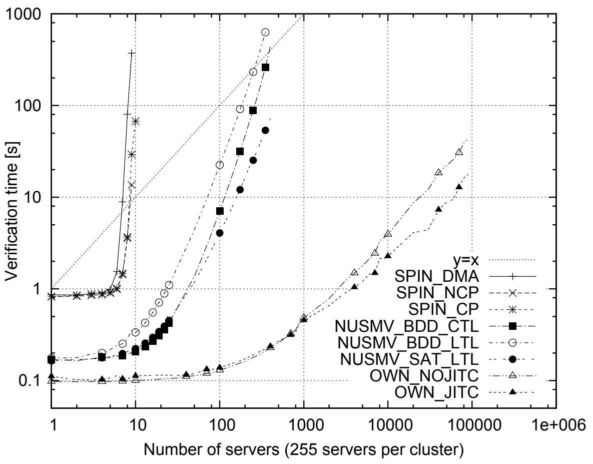
(c) *FOp* workload on Model$_4$.



(d) *FOn* workload on Model$_4$.



(e) *SHTp* & *SHTn* workload on Model$_4$.



(f) *FOp* & *FOn* workload on Model$_4$.

Figure B.5: Time to verify different workloads on Model$_4$ depending on the number of servers, the model checker, and optimization technique used.

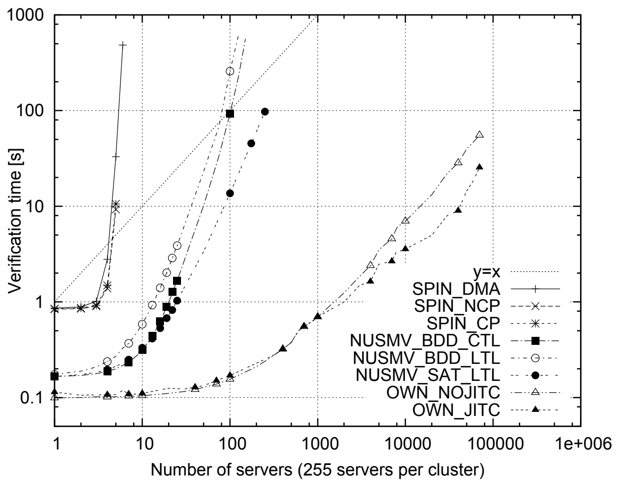
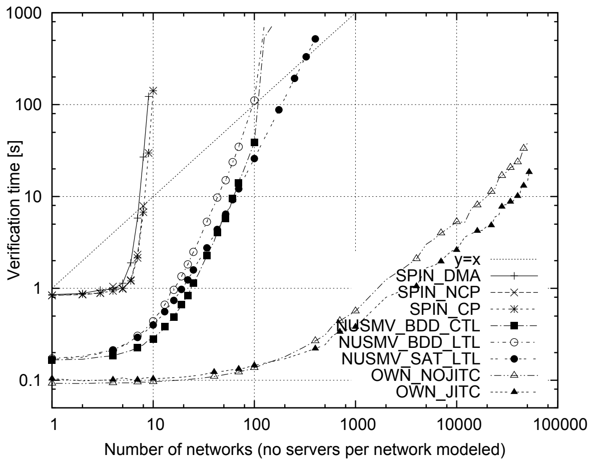
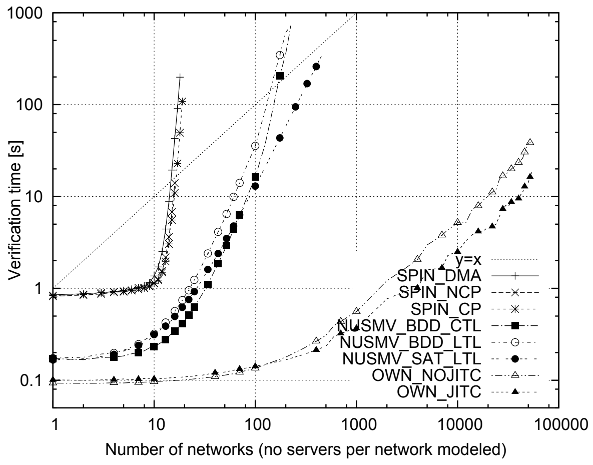


Figure B.6: Time to verify *SHTp* & *SHTn* & *FOp* & *FOn* workload on Model$_4$.

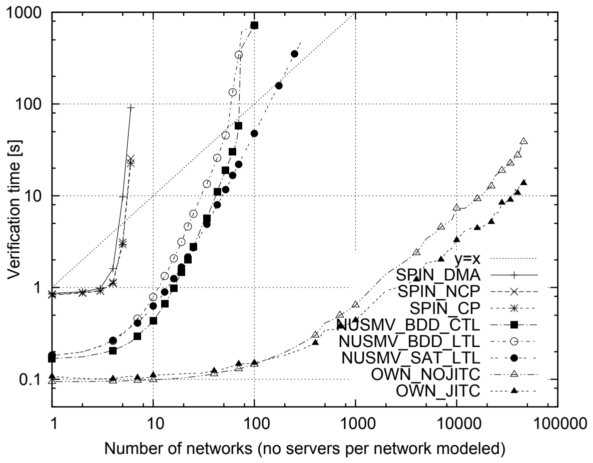
B.1.5 Model 5



(a) *HCRIncrM* workload on Model$_5$.



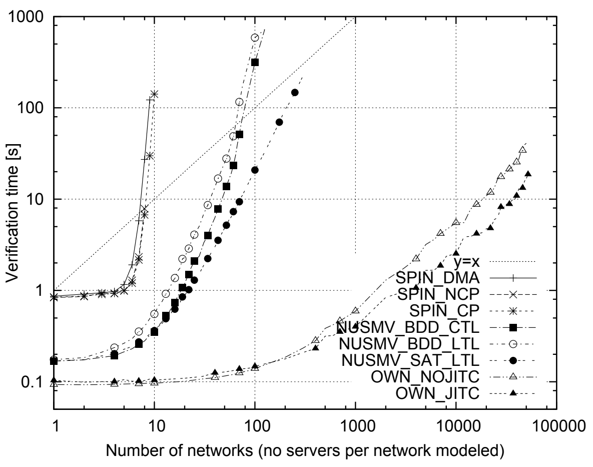
(b) *LCRDecrM* workload on Model$_5$.



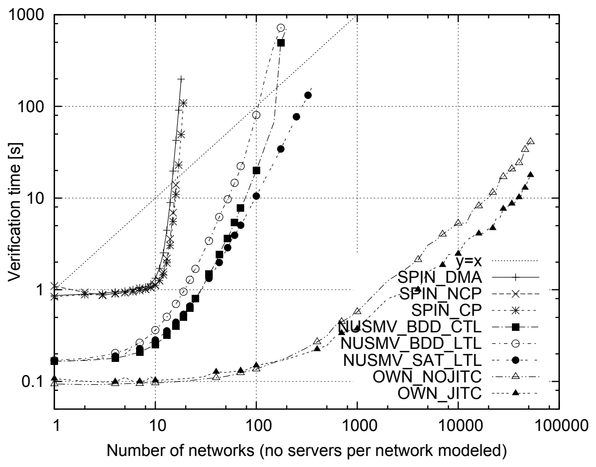
(c) *HCRIncrM* & *LCRDecrM* workload on Model$_5$.

Figure B.7: Time to verify different workloads on Model$_5$ depending on the number of servers, the model checker, and optimization technique used.

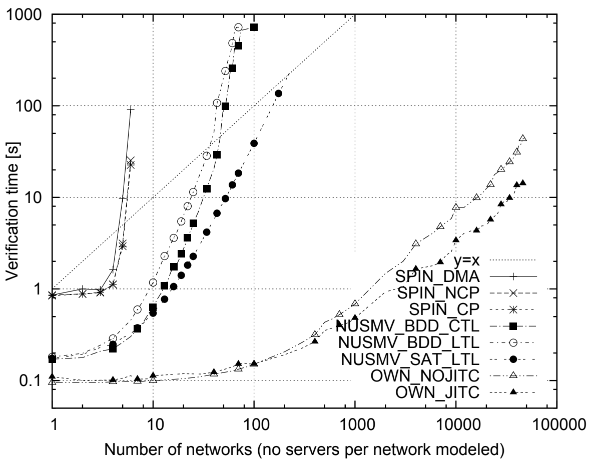
B.1.6 Model 6



(a) *HCRIncrOSPF* workload on Model_6.



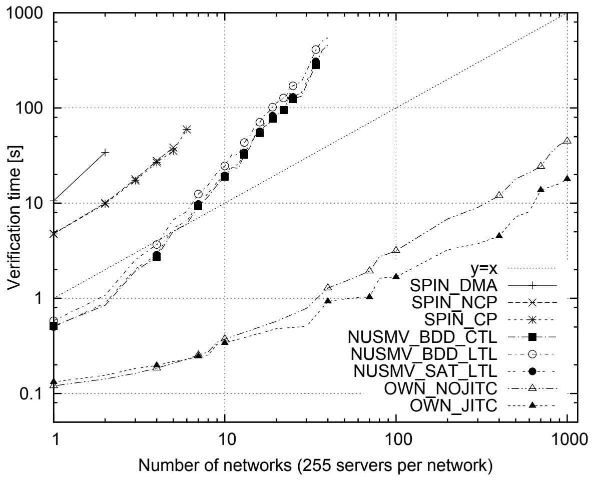
(b) *LCRDecrOSPF* workload on Model_6.



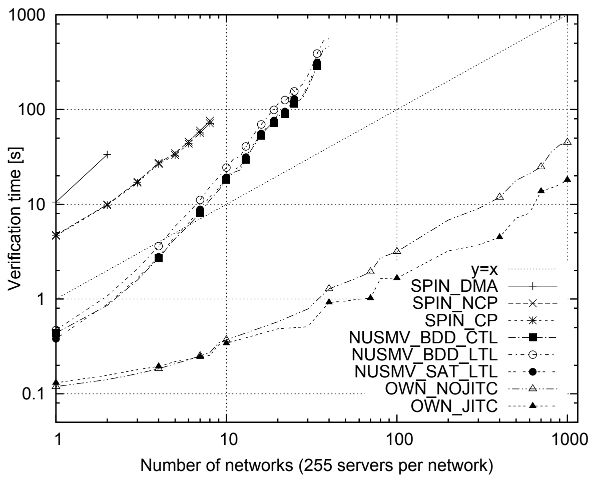
(c) *HCRIncrOSPF* & *LCRDecrOSPF* workload on Model_6.

Figure B.8: Time to verify different workloads on Model_6 depending on the number of servers, the model checker, and optimization technique used.

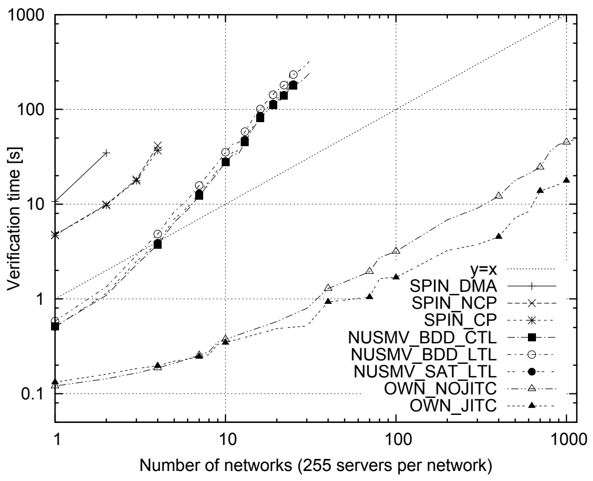
B.1.7 Model 7



(**a**) *HCRIncrM* workload on Model$_7$.



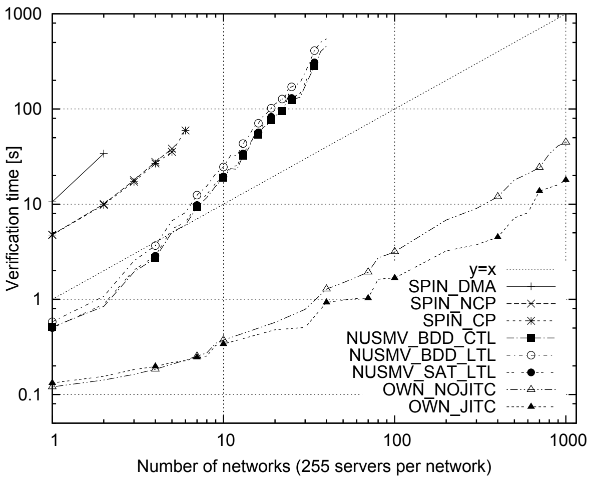
(**b**) *LCRDecrM* workload on Model$_7$.



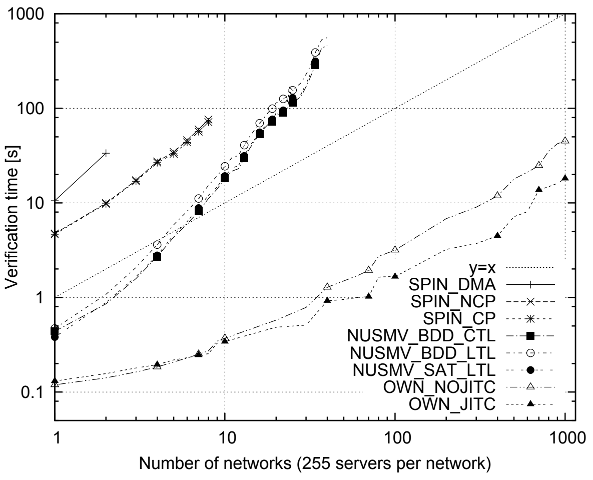
(**c**) *HCRIncrM* & *LCRDecrM* workload on Model$_7$.

Figure B.9: Time to verify different workloads on Model$_7$ depending on the number of servers, the model checker, and optimization technique used.

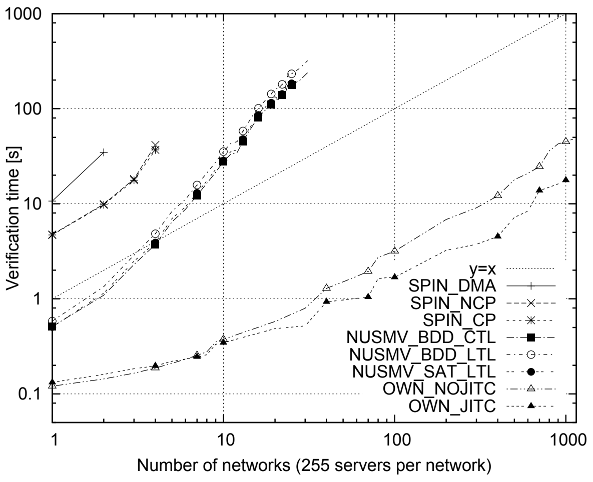
B.1.8 Model 8



(a) *HCRIncrOSPF* workload on Model$_8$.



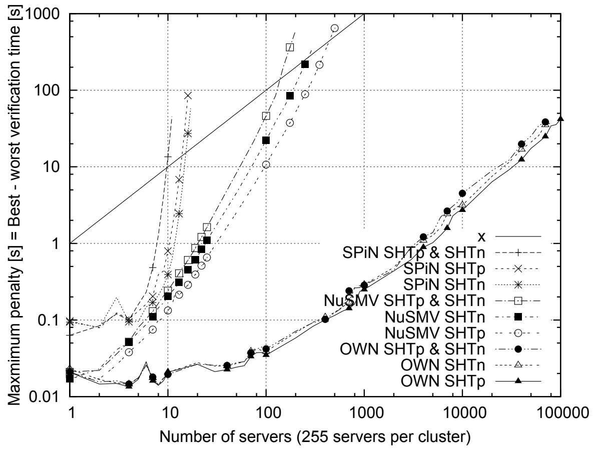
(b) *LCRDecrOSPF* workload on Model$_8$.



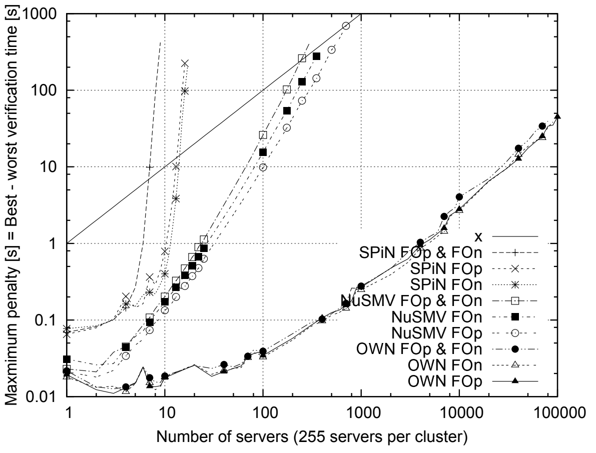
(c) *HCRIncrOSPF* & *LCRDecrOSPF* workload on Model$_8$.

Figure B.10: Time to verify different workloads on Model$_8$ depending on the number of servers, the model checker, and optimization technique used.

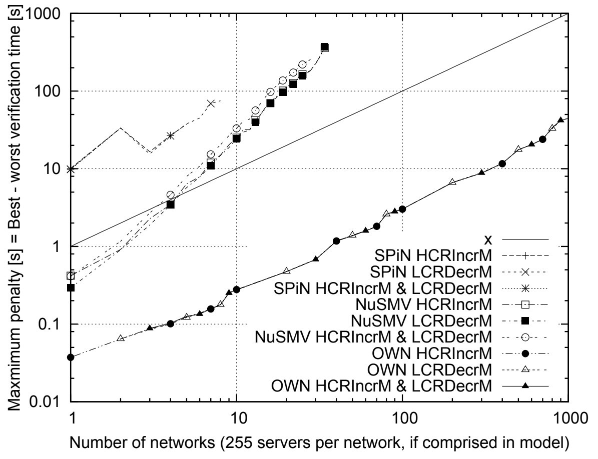
B.2 Robustness of Verification Performance



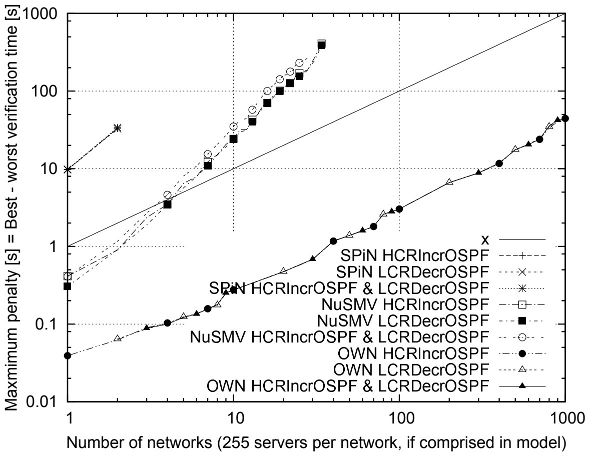
(a) Maximum penalty for the *SHTp*, *SHTn*, and *SHTp* & *SHTn* workloads.



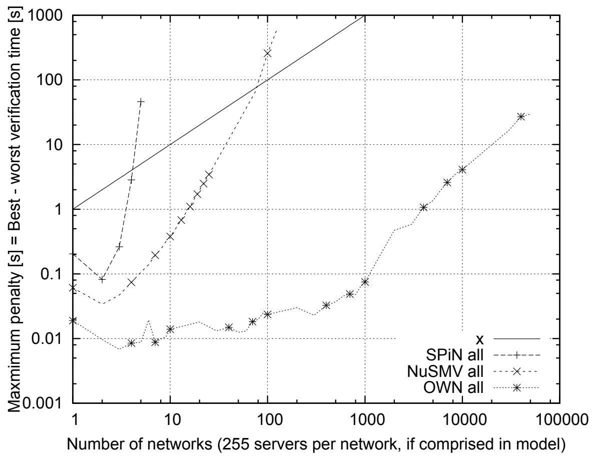
(b) Maximum penalty for the *FOp*, *FOn*, and *FOp* & *FOn* workloads.



(c) Maximum penalty for the *HCRIncrM*, *LCRDecrM*, and *HCRIncrM* & *LCRDecrM* workloads.



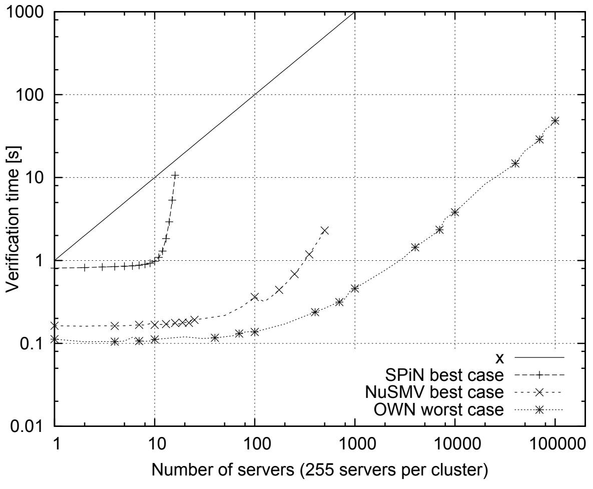
(d) Maximum penalty for the *HCRIncrOSPF*, *LCRDecrOSPF*, and *HCRIncrOSPF* & *LCRDecrOSPF* workloads.



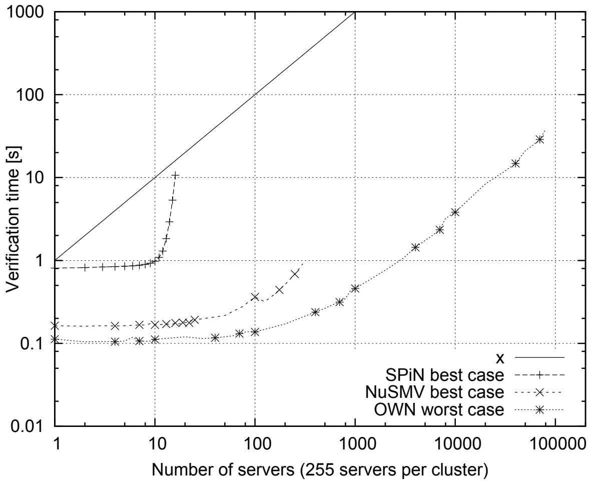
(e) Maximum penalty for the *SHTp* & *SHTn* & *FOp* & *FOn* workload.

Figure B.11: Maximum penalties for all model checkers and workloads.

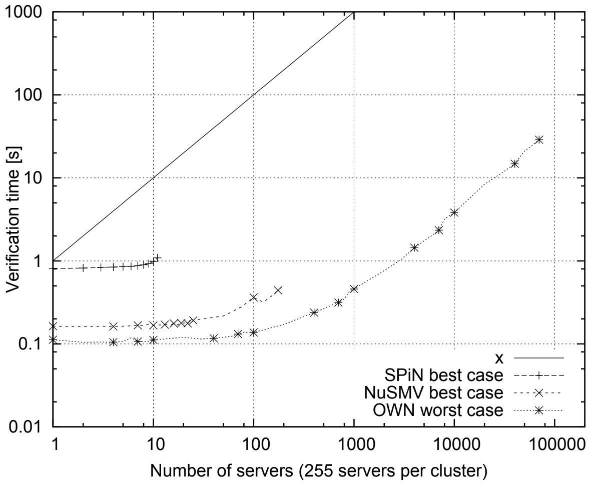
B.3 Best- vs. Worst-case Verification Performance



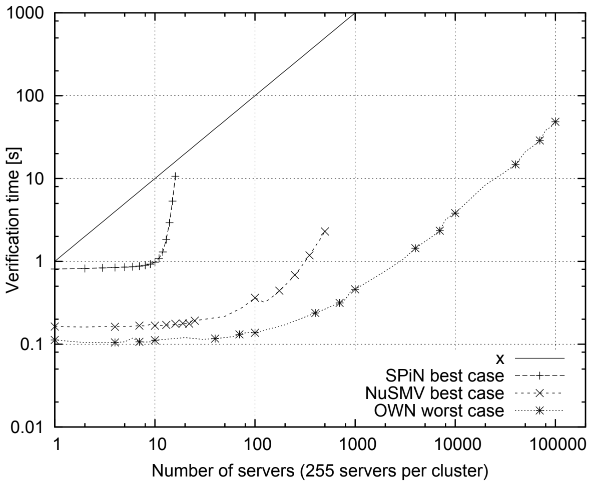
(a) *SHTp* workload.



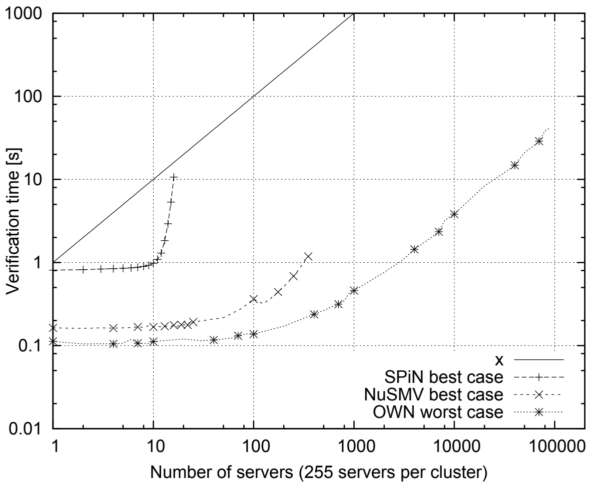
(b) *SHTn* workload.



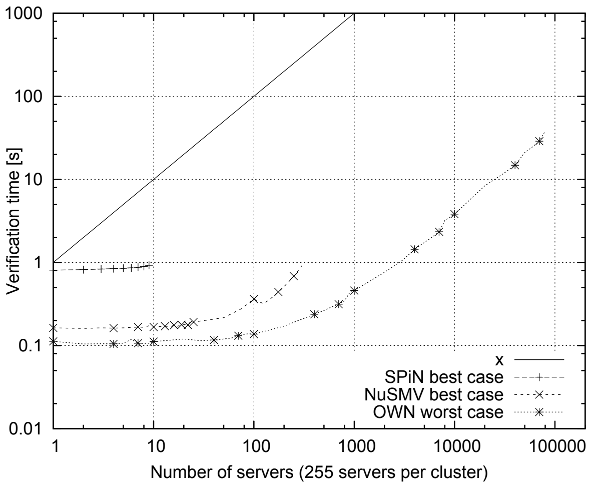
(c) *SHTp* & *SHTn* workload.



(d) *FOp* workload.

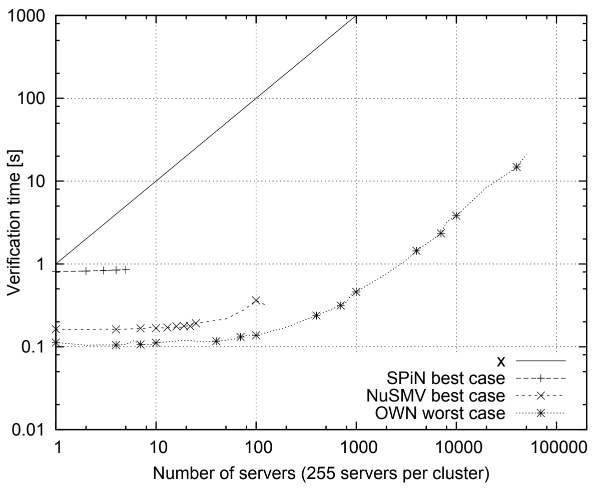


(e) *FOn* workload.

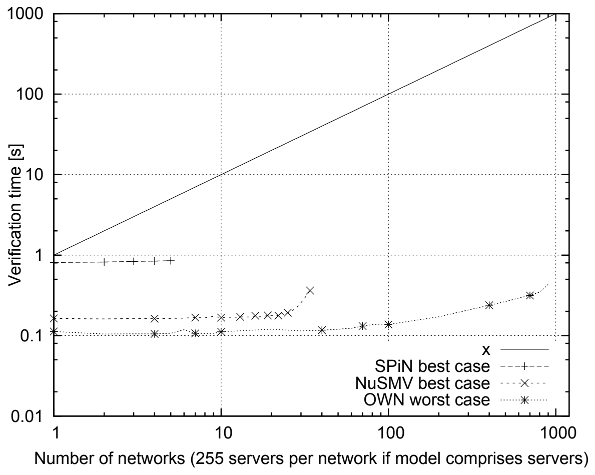


(f) *FOp* & *FOn* workload.

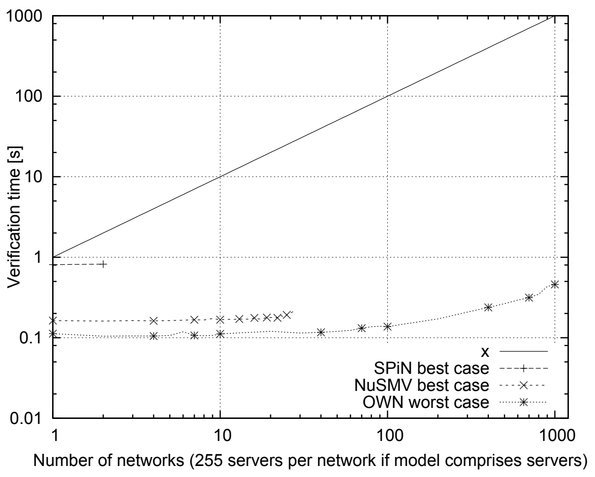
Figure B.12: Worst-case times of our model checker vs. best-case times of SPiN and NuSMV model checkers (selected among all models and optimization techniques).



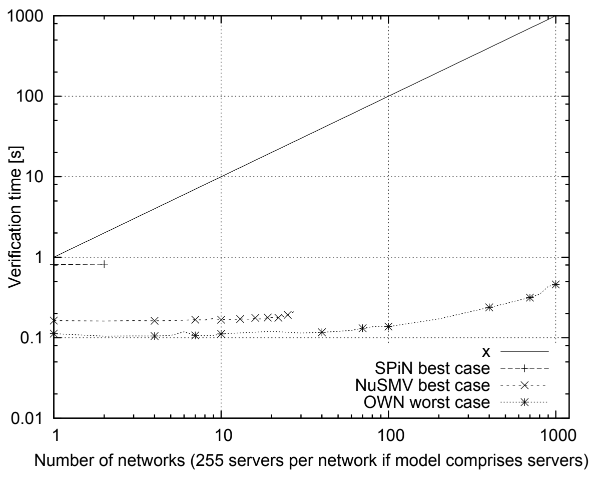
(a) *SHTp* & *SHTn* & *FOp* & *FOn* workload.



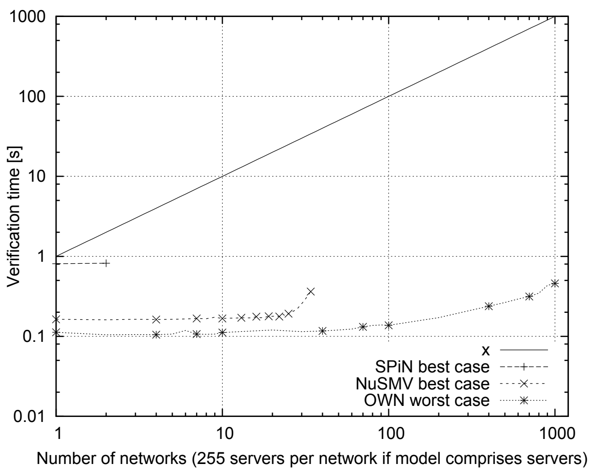
(b) *HCRIncrM* workload.



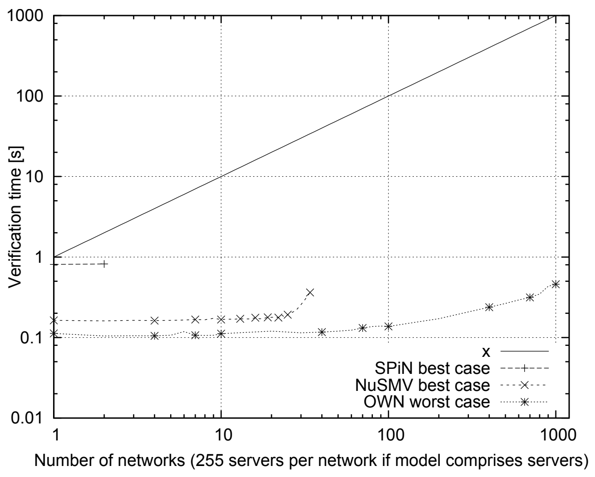
(c) *LCRDecrM* workload.



(d) *HCRIncrM* & *LCRDecrM* workload.



(e) *HCRIncrOSPF* workload.



(f) *LCRDecrOSPF* workload.

Figure B.13: Worst-case times of our model checker vs. best-case times of SPiN and NuSMV model checkers (selected among all models and optimization techniques).

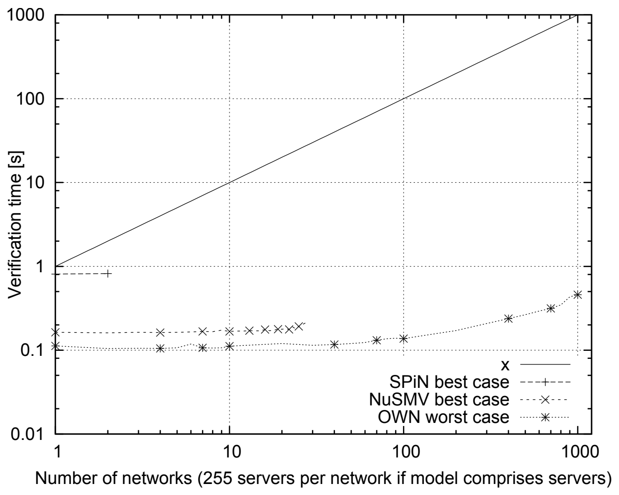


Figure B.14: Worst-case vs. best-case for model checkers and *HCRIncrOSPF* & *LCRDecrOSPF* workload.

APPENDIX C

Verification Models

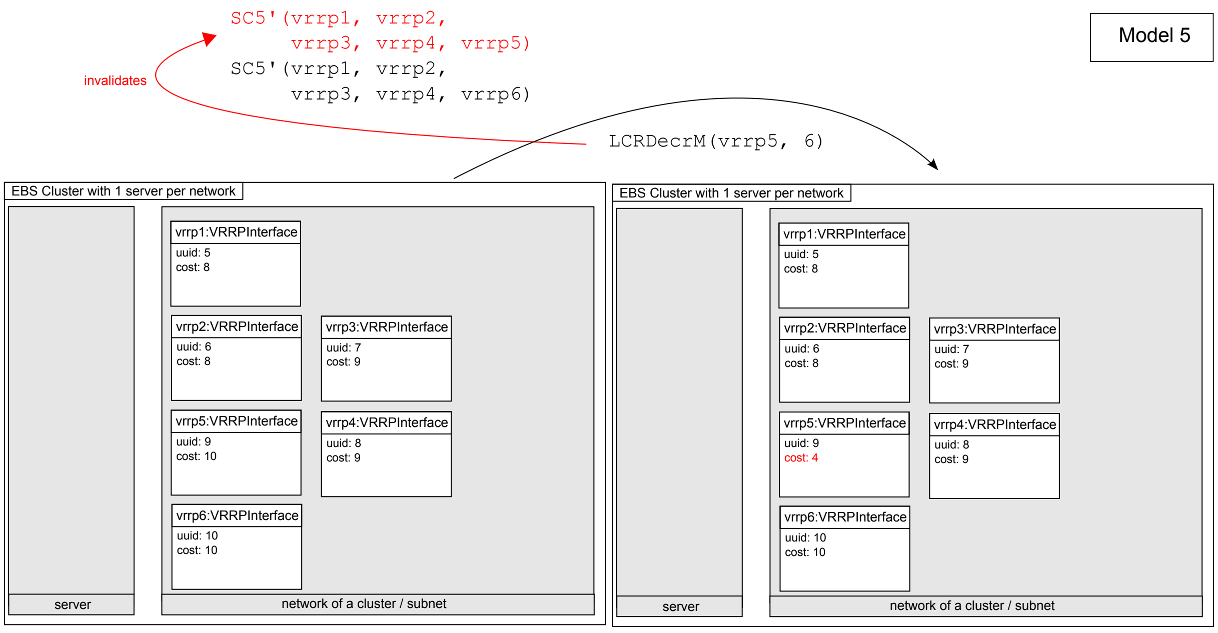


Figure C.1: *LCRDecrM* change activity and safety constraints in a one-server EBS cluster in Model$_5$.

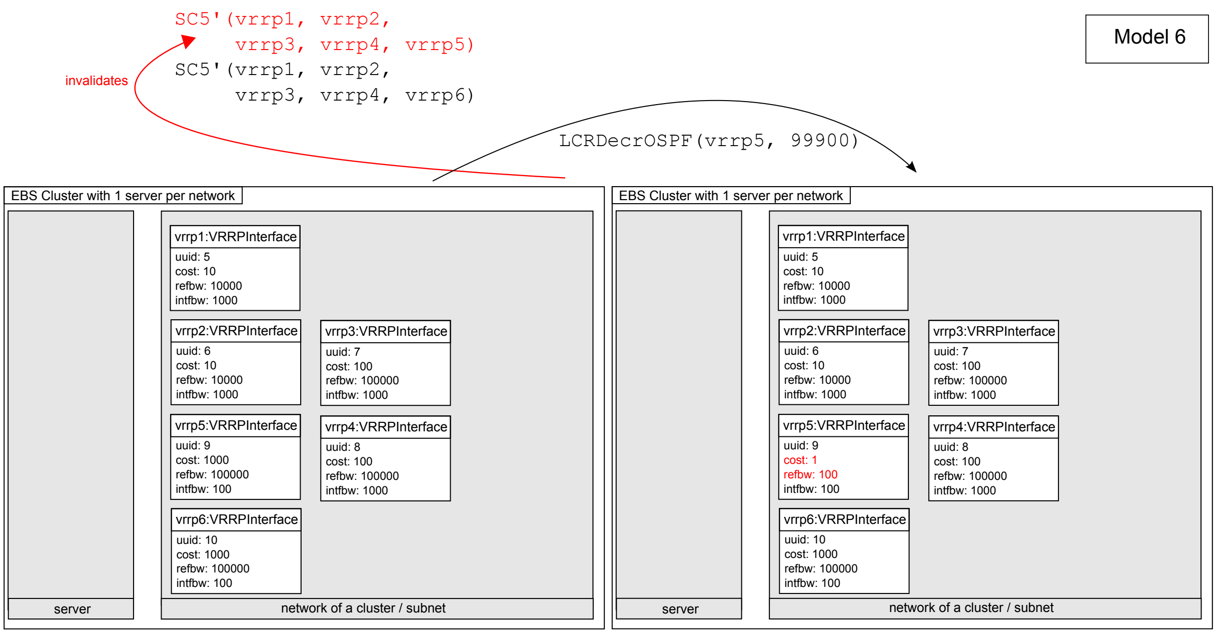


Figure C.2: *LCRDecrOSPF* change activity and safety constraints in a one-server EBS cluster in Model$_6$.

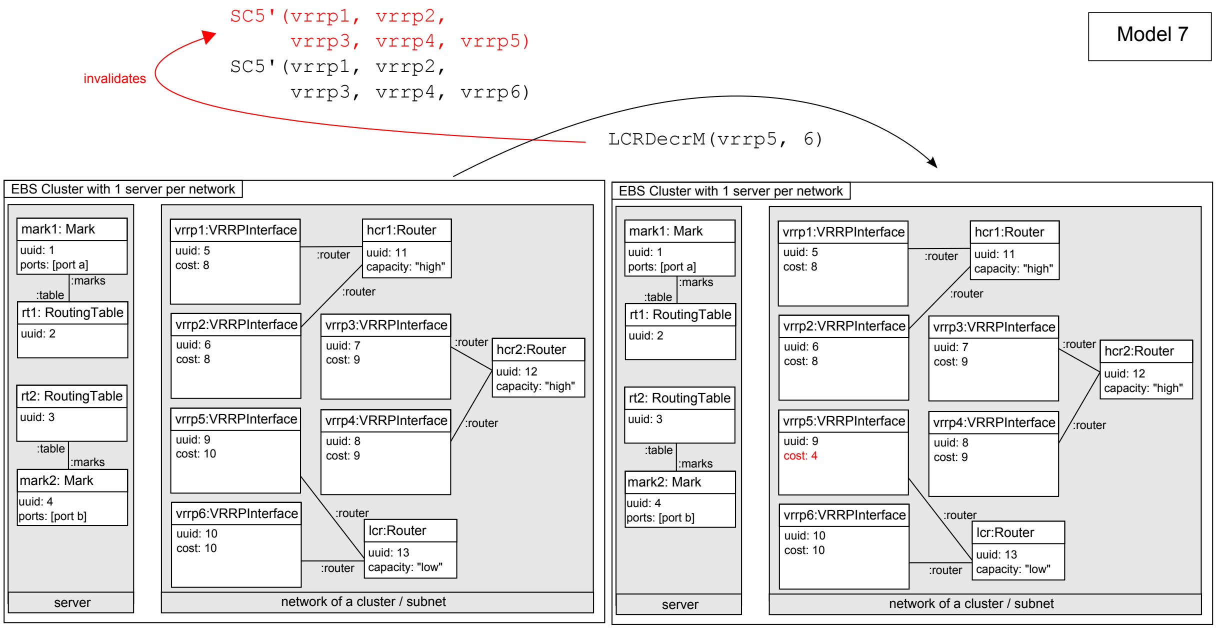


Figure C.3: *LCRDecrM* change activity and safety constraints in a one-server EBS cluster in Model$_7$.

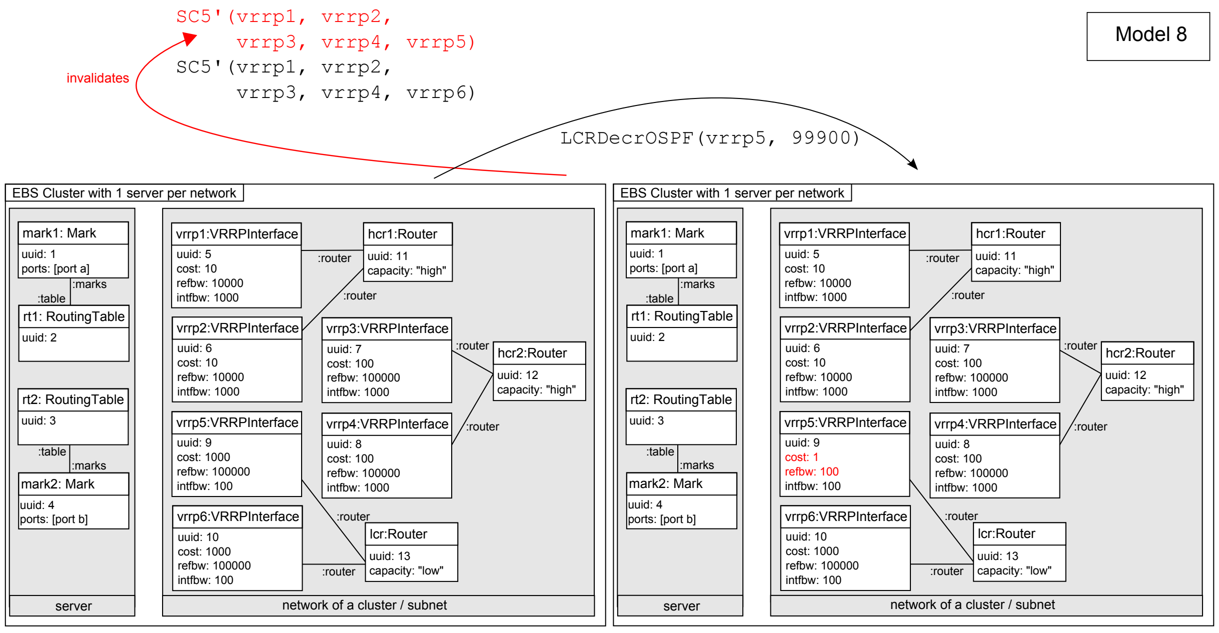
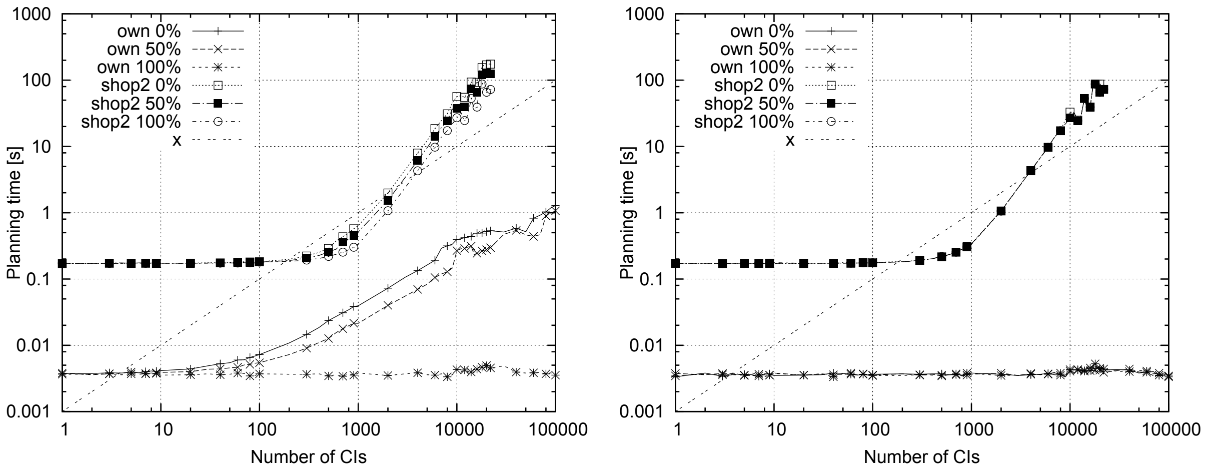


Figure C.4: *LCRDecrOSPF* change activity and safety constraints in a one-server EBS cluster in Model$_8$.

APPENDIX D

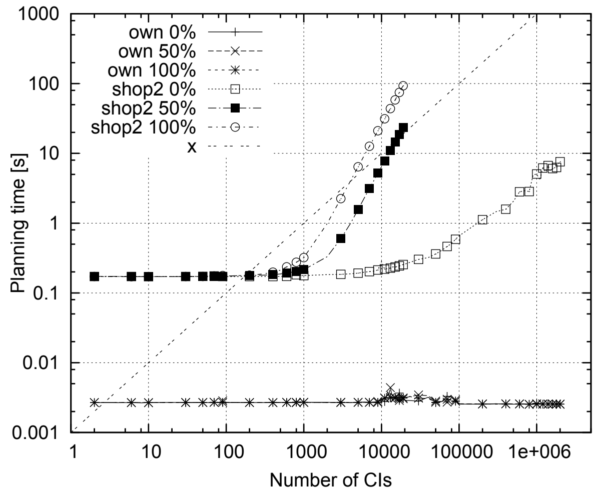
Planning Benchmarks and Microbenchmarks

This chapter provides the remaining graphs of the benchmarks that have have been summarized in Table 7.1 (Microbenchmark 1, Section 7.3.3), Table 7.2 (Microbenchmark 2, Section 7.3.4), Table 7.3 (Microbenchmark 3, Section 7.3.5), and Table 7.6 (network unconfiguration case study, Section 7.4.2).

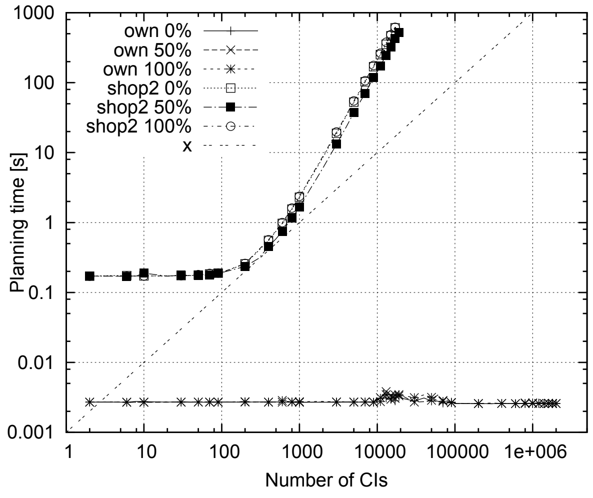


(a) MB2 (4,5,6): UQ-Q, without effect. **(b)** MB2 (10,11,12): Q-UQ, without effect.

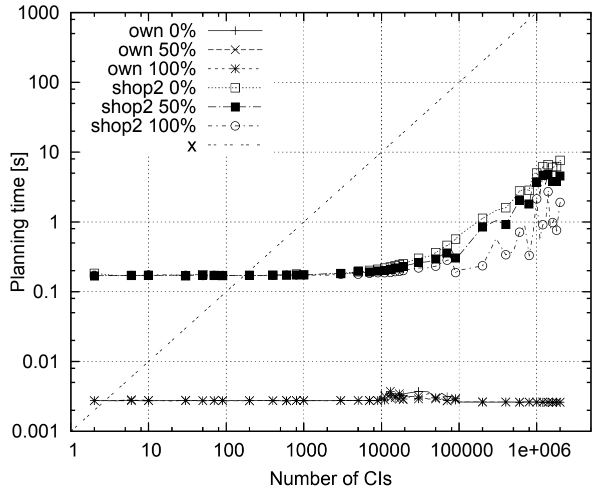
Figure D.1: Runtime performance of Microbenchmark 2 without effects.



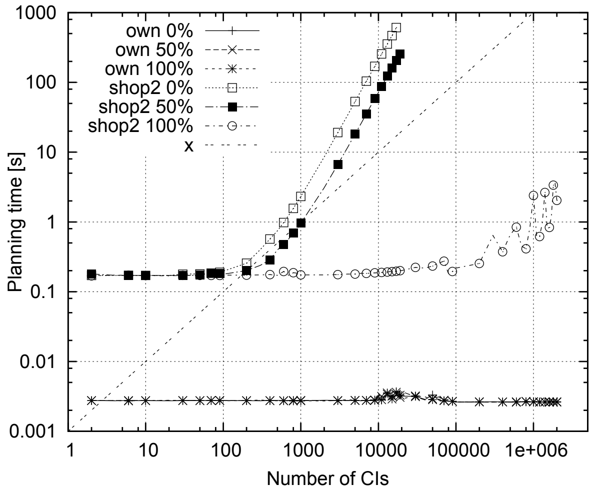
(a) MB1 (1,2,3): Q-UQ, Method, 1 CI.



(b) MB1 (4,5,6): Q-UQ, Method, 2 CIs.



(c) MB1 (7,8,9): Q-UQ, Activity, 1 CI.



(d) MB1 (10,11,12): Q-UQ, Activity, 2 CIs.

Figure D.2: Runtime performance of Microbenchmark 1 for Q-UQ CMDB layouts.

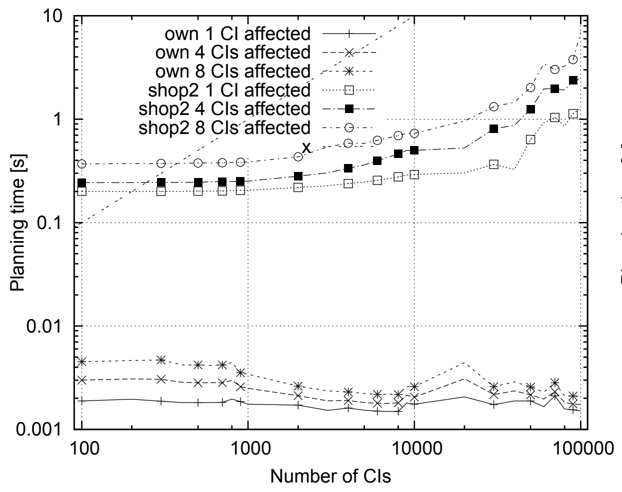


Figure D.3: MB3 (4,5,6): CI not at front.

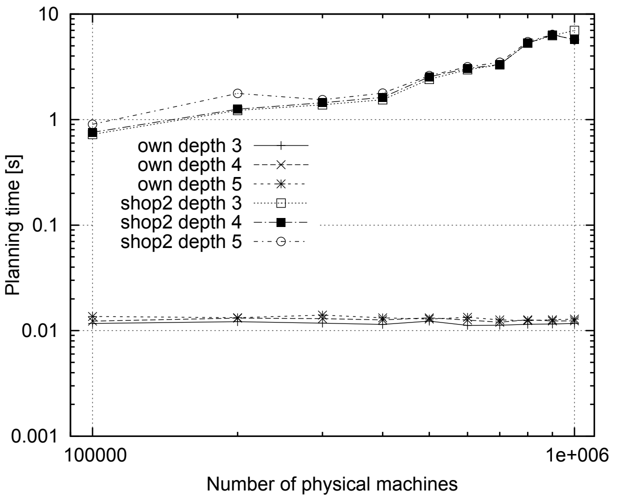


Figure D.4: Network unconfiguration case study (4,5,6): PMS-NW CMDB layout.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/en/message/65648/, Apr. 2011.

[2] J. M. Agosta and D. E. Wilkins. Using SIPE-2 to Plan Emergency Response to Marine Oil Spills. *IEEE Expert*, 11(6):6–8, Dec 1996.

[3] D. Agrawal, J. Giles, K.-W. Lee, K. Voruganti, and K. Filali-Adib. Policy-Based Validation of SAN Configuration. In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004), Yorktown Heights, NY, USA, June 7-9, 2004*, pages 77–86. IEEE, 2004.

[4] D. Agrawal, L. Kang-Won, and J. Lobo. Policy-based Management of Networked Computing Systems. *Communications Magazine, IEEE*, 43(10):69 – 75, oct. 2005.

[5] E. S. Al-Shaer and H. H. Hamed. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In *Proceedings of the 8th International Symposium on Integrated Network Management (IM 2003), Colorado Springs, CO, USA, March 24-28, 2003*, volume 246 of *IFIP Conference Proceedings*, pages 17–30. Kluwer, 2003.

[6] E. S. Al-Shaer and H. H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *Proceedings of the 23rd International Conference on Computer Communications (INFOCOM 2004), Hong Kong, China, March 7-11, 2004*, volume 4, pages 2605–2616, 2004.

[7] E. S. Al-Shaer and H. H. Hamed. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management*, 1(1):2–10, 2004.

[8] M. Arregoces and M. Portolani. *Data Center Fundamentals: Understand Data Canter Network Design and Infrastructure Architecture, Including Load Balancing, SSL, and Security*. Macmillan Technical Publishing, 2003.

[9] F. Baader and W. Snyder. Unification Theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 445–532. Elsevier B.V., 2001.

[10] F. Bacchus and F. Kabanza. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artifical Intelligence*, 116(1-2):123–191, 2000.

[11] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[12] C. Baier and J.-P. Katoen. *Principles of Model Checking*, chapter Partial Order Reduction. The MIT Press, 2008.

[13] A. K. Bandara, E. Lupu, and A. Russo. Using Event Calculus to Formalise Policy Specification and Analysis. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy, June 4-6 June, 2003*, pages 26–39. IEEE, 2003.

[14] C. Bartolini, C. Stefanelli, and M. Tortonesi. SYMIAN: Analysis and Performance Improvement of the IT Incident Management Process. *IEEE Transactions on Network and Service Management*, 7(3):132–144, 2010.

[15] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[16] L. A. Bianchin, J. A. Wickboldt, L. Z. Granville, L. P. Gaspary, C. Bartolini, and M. Rahmouni. Similarity Metric for Risk Assessment in IT Change Plans. In *Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), Niagara Falls, Canada, October 25-29, 2010*, pages 25–32. IEEE, 2010.

[17] L. Blair and K. J. Turner. Handling Policy Conflicts in Call Control. In *Proceedings of the 8th Conference on Feature Interactions in Telecommunications and Software Systems (ICFI 2005), Leicester, UK, June 28-30, 2005*, pages 39–57. IOS Press, 2005.

[18] A. Blum and M. L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[19] J. M. Bradshaw, A. Uszok, R. Jeffers, N. Suri, P. J. Hayes, M. H. Burstein, A. Acquisti, B. Benyo, M. R. Breedy, M. M. Carvalho, D. J. Diller, M. Johnson, S. Kulkarni, J. Lott, M. Sierhuis, and R. van Hoof. Representation and Reasoning for DAML-based Policy and Domain Services in KAoS and Nomads. In *Proceedings of the 2nd International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2003), Melbourne, Victoria, Australia, July 14-18, 2003*, pages 835–842. ACM, 2003.

[20] M. A. Brown. Guide to IP Layer Network Administration with Linux. http://linux-ip.net/, 2003.

[21] W. Bumpus, J. W. Sweitzer, P. Thompson, A. R. Westerinen, and R. C. Williams. *Common Information Model: Implementing the Object Model for Enterprise Management*. John Wiley & Sons, 2000.

[22] R. Calinescu and S. Kikuchi. Formal Methods @ Runtime. In R. Calinescu and E. Jackson, editors, *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*, volume 6662 of *Lecture Notes in Computer Science*, pages 122–135. Springer Berlin / Heidelberg, 2011.

[23] M. Charalambides, P. Flegkas, G. Pavlou, A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, and J. Rubio-Loyola. Policy Conflict Analysis for Quality of Service Management. In *Proceedings of the 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), Stockholm, Sweden, June 6-8, 2005*, pages 99–108. IEEE, 2005.

[24] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, and M. Sloman. Policy Conflict Analysis for DiffServ Quality of Service Management. *IEEE Transactions on Network and Service Management*, 6(1):15–30, 2009.

[25] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, M. Sloman, and N. Dulay. Dynamic Policy Analysis and Conflict Resolution for DiffServ Quality of Service Management. In *Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006), Vancouver, Canada, April 3-7, 2006*, pages 294–304. IEEE, 2006.

[26] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002), Copenhagen, Denmark, July 27-31, 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[27] A. Cimatti, E. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Integrating BDD-based and SAT-based Symbolic Model Checking. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS 2002), Santa Margherita Ligure, Italy, April 8-10, 2002*, volume 2309 of *Lecture Notes in Computer Science*, pages 49–56. Springer, 2002.

[28] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the Encoding of LTL Model Checking into SAT. In *Proceedings of the 3rd International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI 2002), Venice, Italy, January 21-22, 2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2002.

[29] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2001.

[30] E. M. Clarke Jr., O. Grumberg, and D. A. Peled. *Model Checking*, chapter Partial Order Reduction. The MIT Press, 2001.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2009.

[32] W. L. da Costa Cordeiro, G. S. Machado, F. G. Andreis, A. D. dos Santos, C. B. Both, L. P. Gaspary, L. Z. Granville, C. Bartolini, and D. Trastour. A Runtime Constraint-Aware Solution for Automated Refinement of IT Change Plans. In *Proceedings of the 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2008), Samos Island, Greece, September 22-26, 2008*, volume 5273 of *Lecture Notes in Computer Science*, pages 69–82. Springer, 2008.

[33] W. L. da Costa Cordeiro, G. S. Machado, F. G. Andreis, A. D. dos Santos, C. B. Both, L. P. Gaspary, L. Z. Granville, C. Bartolini, and D. Trastour. ChangeLedge: Change Design and Planning in Networked Systems Based on Reuse of Knowledge and Automation. *Computer Networks*, 53(16):2782–2799, 2009.

[34] W. L. da Costa Cordeiro, G. S. Machado, F. G. Andreis, J. A. Wickboldt, R. C. Lunardi, A. D. dos Santos, C. B. Both, L. P. Gaspary, L. Z. Granville, D. Trastour, and C. Bartolini. Changeminer: A solution for discovering it change templates from past execution traces. In *Integrated Network Management*, pages 97–104. IEEE, 2009.

[35] W. L. da Costa Cordeiro, G. S. Machado, F. F. Daitx, C. B. Both, L. P. Gaspary, L. Z. Granville, A. Sahai, C. Bartolini, D. Trastour, and K. B. Saikoski. A Template-based Solution to Support Knowledge Reuse in IT Change Design. In *Proceedings of the 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Bahia, Brazil, April 7-11, 2008*, pages 355–362. IEEE, 2008.

[36] Y. Diao and A. Heching. Staffing Optimization in Complex Service Delivery Systems. In *Proceedings of the 7th International Conference on Network and Service Management (CNSM 2011), Paris, France, October 24-28, 2011*, pages 1–9. IEEE, 2011.

[37] N. Dunlop, J. Indulska, and K. Raymond. Dynamic Conflict Detection in Policy-Based Management Systems. In *Proceedings of the 6th International Enterprise Distributed Object Computing Conference (EDOC 2002), Lausanne, Switzerland, September 17-20, 2002*, pages 15–26. IEEE, 2002.

[38] T. Eilam, M. H. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the Configuration Complexity of Distributed Applications in Internet Data Centers. *IEEE Communications Magazine*, 44(3):166–177, 2006.

[39] T. A. Estlin, S. A. Chien, and X. Wang. An Argument for a Hybrid HTN/Operator-Based Approach to Planning. In *Proceedings of the 4th European Conference on Planning (ECP 1997), Toulouse, France, September 24-26, 1997*, volume 1348 of *Lecture Notes in Computer Science*, pages 182–194. Springer, 1997.

[40] S. Ferraresi, S. Pesic, L. Trazza, and A. Baiocchi. Automatic Conflict Analysis and Resolution of Traffic Filtering Policy for Firewall and Security Gateway. In *Proceedings of the IEEE International Conference on Communications (ICC 2007), Glasgow, Scotland, June 24-28 June 2007*, pages 1304–1310. IEEE, 2007.

[41] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann, 2004.

[42] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory & Practice*, chapter Hierarchical Task Network Planning. Morgan Kaufmann, 2004.

[43] V. D. Gligor, S. I. Gavrila, and D. F. Ferraiolo. On the Formal Definition of Separation-of-Duty Policies and their Composition. In *Proceedings of the 19th IEEE Symposium on Security and Privacy (S&P 1998), Oakland, CA, USA, May 3-6, 1998*, pages 172–183. IEEE, 1998.

[44] P. Goldsack, J. Guijarro, S. Loughran, A. N. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog Configuration Management Framework. *Operating Systems Review*, 43(1):16–25, 2009.

[45] S. Hagen, W. L. da Costa Cordeiro, L. P. Gaspary, L. Z. Granville, M. Seibold, and A. Kemper. Plannig in the Large: Efficient Generation of IT Change Plans on Large Infrastructures. In *Proceedings of the 8th International Conference on Network and Service Management (CNSM 2012), Las Vegas, NV, USA, October 22-26, 2012*, pages 108–116. IEEE, 2012.

[46] S. Hagen, N. Edwards, L. Wilcock, J. Kirschnick, and J. Rolia. One Is Not Enough: A Hybrid Approach for IT Change Planning. In *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2009), Venice, Italy, October 27-28, 2009*, volume 5841 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2009. *This work has been part of the author's preceding master's degree.*

[47] S. Hagen and A. Kemper. Facing The Unpredictable: Automated Adaption of IT Change Plans for Unpredictable Management Domains. In *Proceedings of the 6th International Conference on Network and Service Management (CNSM 2010), Niagara Falls, Canada, October 25-29, 2010*, pages 33–40. IEEE, 2010.

[48] S. Hagen and A. Kemper. Model-based Planning for State-related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. In *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD 2010), Miami, FL, USA, July 5-10, 2010*, pages 11–18. IEEE, 2010.

[49] S. Hagen and A. Kemper. A Performance and Usability Comparison of Automated Planners for IT Change Planning. In *Proceedings of the 7th International Conference on Network and Service Management (CNSM 2011), Paris, France, October 24-28, 2011*, pages 1–9. IEEE, 2011.

[50] S. Hagen and A. Kemper. Towards Solid IT Change Management: Automated Detection of Conflicting IT Change Plans. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011), Dublin, Ireland, May 23-27, 2011*, pages 265–272. IEEE, 2011.

[51] S. Hagen, M. Seibold, and A. Kemper. Efficient Verification of IT Change Operations or: How We Could Have Prevented Amazon's Cloud Outage. In *Proceedings of the IEEE Network Operations and Management Symposium (NOMS 2012), Maui, HI, USA, April 16-20, 2012*, pages 368–376. IEEE, 2012.

[52] P. Harsh, F. Dudouet, R. G. Cascella, Y. Jégou, and C. Morin. Using Open Standards for Interoperability - Issues, Solutions, and Challenges Facing Cloud Computing. *CoRR*, abs/1207.5949, 2012.

[53] H. Herry, P. Anderson, and G. Wickler. Automated Planning for Configuration Changes. In *Proceedings of the 25th Large Installation System Administration Conference (LISA 2011), Boston, MA, USA, December 4-9, 2011*, pages 57–68, 2011.

[54] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley, 2003.

[55] V. C. Hu, D. R. Kuhn, and T. Xie. Property Verification for Generic Access Control Models. In *Proceedings of the 6th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC 2008), Shanghai, China, December 17-20, 2008, Volume II: Workshops*, pages 243–250. IEEE, 2008.

[56] V. C. Hu, D. R. Kuhn, T. Xie, and J. Hwang. Model Checking for Verification of Mandatory Access Control Models and Properties. *International Journal of Software Engineering and Knowledge Engineering*, 21(1):103–127, 2011.

[57] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, 26(2):214–260, 2001.

[58] J.-P. Jouannaud and C. Kirchner. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of A. Robinson*, chapter Solving Equations in Abstract Algebras: A Rule-based Survey of Unification. MIT Press, Cambridge, MA, USA, 1991.

[59] A. Keller, J. L. Hellerstein, J. L. Wolf, K.-L. Wu, and V. Krishnan. The CHAMPS System: Change Management with Planning and Scheduling. In *Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), Seoul, Korea, April 19-23, 2004*, pages 395–408. IEEE, 2004.

[60] A. Keller and S. Subramanian. Best Practices for Deploying a CMDB in Large-scale Environments. In *Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM 2009), Hofstra University, Long Island, NY, USA, June 1-5, 2009*, pages 732–745. IEEE, 2009.

[61] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama. Policy Verification and Validation Framework Based on Model Checking Approach. In *Proceedings of the 4th International Conference on Autonomic Computing (ICAC 2007), Jacksonville, Florida, USA, June 11-15, 2007*, page 1. IEEE, 2007.

[62] M. Kolon. Intelligent Logical Router Service. White paper, Juniper Networks, Inc., 1194 North Mathilda Avenue, Sunnyvale, CA 94089, USA, oct 2004.

[63] R. A. Kowalski and M. J. Sergot. A Logic-based Calculus of Events. *New Generation Computing*, 4(1):67–95, 1986.

[64] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*, pages 200–204. Springer Berlin Heidelberg, 2002.

[65] S. Lacy and I. Macfarlane. *Service Transition.* The Stationery Office, 2007.

[66] N. Loutas, E. Kamateri, and K. A. Tarabanis. A Semantic Interoperability Framework for Cloud Platform as a Service. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2011), Athens, Greece, November 29 - December 1, 2011*, pages 280–287. IEEE, 2011.

[67] R. C. Lunardi, F. G. Andreis, W. L. da Costa Cordeiro, J. A. Wickboldt, B. L. Dalmazo, R. L. dos Santos, L. A. Bianchin, L. P. Gaspary, L. Z. Granville, and C. Bartolini. On Strategies for Planning the Assignment of Human Resources to IT Change Activities. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium (NOMS 2010), Osaka, Japan, April 19-23, 2010*, pages 248–255. IEEE, 2010.

[68] E. Lupu and M. Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[69] G. S. Machado, W. L. da Costa Cordeiro, A. D. dos Santos, J. A. Wickboldt, R. C. Lunardi, F. G. Andreis, C. B. Both, L. P. Gaspary, L. Z. Granville, D. Trastour, and C. Bartolini. Refined Failure Remediation for IT Change Management Systems. In *Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM 2009), Hofstra University, Long Island, NY, USA, June 1-5, 2009*, pages 638–645. IEEE, 2009.

[70] G. S. Machado, F. F. Daitx, W. L. da Costa Cordeiro, C. B. Both, L. P. Gaspary, L. Z. Granville, C. Bartolini, A. Sahai, D. Trastour, and K. B. Saikoski. Enabling Rollback Support in IT Change Management Systems. In *Proceedings of the 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Bahia, Brazil, April 7-11, 2008*, pages 347–354. IEEE, 2008.

[71] K. E. Maghraoui, A. Meghranjani, T. Eilam, M. H. Kalantar, and A. V. Konstantinou. Model Driven Provisioning: Bridging the Gap Between Declarative Object Models and Procedural Provisioning Tools. In *Proceedings of the 7th ACM/IFIP/USENIX International Middleware Conference (Middleware 2006), Melbourne, Australia, November 27-December 1, 2006*, volume 4290 of *Lecture Notes in Computer Science*, pages 404–423. Springer, 2006.

[72] M. Manzano. Introduction to Many-sorted Logic. In K. Meinke and J. V. Tucker, editors, *Many-sorted Logic and its Applications*, pages 3–86. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[73] M. L. Massie, B. N. Chun, and D. E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[74] B. Morisset and M. Ghallab. Learning How to Combine Sensory-motor Functions into a Robust Behavior. *Artificial Intelligence*, 172(4-5):392–412, 2008.

[75] H. Muñoz-Avila, D. W. Aha, D. S. Nau, R. Weber, L. Breslow, and F. Yaman. SiN: Integrating Case-based Reasoning with Task Decomposition. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, Washington, USA, August 4-10, 2001*, pages 999–1004. Morgan Kaufmann, 2001.

[76] D. S. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

[77] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1982.

[78] Office of Government Commerce. *ITIL Lifecycle Publication Suite*. The Stationery Office, 2007.

[79] F. Oliveira, A. Tjang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Barricade: Defending Systems Against Operator Mistakes. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys 2010), Paris, France, April 13-16, 2010*, pages 83–96. ACM, 2010.

[80] D. Peled. Verification for Robust Specification. In *Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logic (TPHOLs 1997), Murray Hill, NJ, USA, August 19-22, 1997*, volume 1275 of *Lecture Notes in Computer Science*, pages 231–241. Springer, 1997.

[81] D. Peled, T. Wilke, and P. Wolper. An Algorithmic Approach for Checking Closure Properties of omega-Regular Languages. In *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR 1996), Pisa, Italy, August 26-29, 1996.*, volume 1119 of *Lecture Notes in Computer Science*, pages 596–610. Springer, 1996.

[82] J. S. Penberthy and D. S. Weld. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR 1992), Cambridge, MA, USA, October 25-29, 1992*, pages 103–114. Morgan Kaufmann, 1992.

[83] C. Radu, S. Kikuchi, and M. Kwiatkowska. Formal Methods for the Development and Verification of Autonomic IT Systems. In P. Cong-Vinh, editor, *Formal and Practical Aspects of Autonomic Computing and Networking: Specification, Development, and Verification*, pages 1–37. Igi Global, 2012.

[84] R. Rebouças, J. P. Sauvé, A. Moura, C. Bartolini, and D. Trastour. A Decision Support Tool to Optimize Scheduling of IT Changes. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management (IM 2007), Munich, Germany, May 21-25, 2007*, pages 343–352. IEEE, 2007.

[85] E. D. Sacerdoti. The Nonlinear Nature of Plans. In *Advance Papers of the 4th International Joint Conference on Artificial Intelligence (IJCIA 1975), Tbilisi, Georgia, USSR, September 3-8, 1975*, pages 206–214, 1975.

[86] T. Samak, E. Al-Shaer, and H. Li. QoS Policy Modeling and Conflict Analysis. In *Proceedings of the 9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), Palisades, New York, USA, June 2-4, 2008*, pages 19–26. IEEE, 2008.

[87] J. P. Sauvé, R. Santos, R. Rebouças, A. Moura, and C. Bartolini. Change Priority Determination in IT Service Management Based on Risk Exposure. *IEEE Transactions on Network and Service Management*, 5(3):178–187, 2008.

[88] T. Setzer, K. Bhattacharya, and H. Ludwig. Change Scheduling Based on Business Impact Analysis of Change-related Risk. *IEEE Transactions on Network and Service Management*, 7(1):58–71, 2010.

[89] A. Tate. Generating Project Networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCIA 1977), Cambridge, MA, USA, August 1977*, pages 888–893. William Kaufmann, 1977.

[90] A. Tjang, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Model-based Validation for Internet Services. In *Proceedings of the 28th IEEE Symposium on Reliable Distributed Systems (SRDS 2009), Niagara Falls, New York, USA, September 27-30, 2009*, pages 61–70. IEEE, 2009.

[91] D. Trastour, R. Fink, and F. Liu. ChangeRefinery: Assisted Refinement of High-level IT Change Requests. In *Proceedings of the 10th IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2009), London, UK, July 20-22, 2009*, pages 68–75. IEEE, 2009.

[92] A. Uszok, J. M. Bradshaw, R. Jeffers, N. Suri, P. J. Hayes, M. R. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott. KAoS Policy and Domain Services: Toward a Description-Logic Approach to Policy Representation, Deconfliction, and Enforcement. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy, June 4-6 June, 2003*, pages 93–96. IEEE, 2003.

[93] A. Uszok, J. M. Bradshaw, J. Lott, M. R. Breedy, L. Bunch, P. J. Feltovich, M. Johnson, and H. Jung. New Developments in Ontology-Based Policy Management: Increasing the Practicality and Comprehensiveness of KAoS. In *Proceedings of the 9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), Palisades, New York, USA, June 2-4, 2008*, pages 145–152. IEEE, 2008.

[94] M. M. Veloso, J. G. Carbonell, M. A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating Planning and Learning: the PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[95] D. S. Weld, C. R. Anderson, and D. E. Smith. Extending Graphplan to handle uncertainty and sensing actions. In *Proceeding of the 15th national/10th conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI 1998/IAAI 1998), Madison, Wisconsin, USA, July 26-30, 1998*, pages 897–904. AAAI Press, 1998.

[96] J. A. Wickboldt, L. A. Bianchin, R. C. Lunardi, F. G. Andreis, W. L. da Costa Cordeiro, C. B. Both, L. Z. Granville, L. P. Gaspary, D. Trastour, and C. Bartolini. Improving IT Change Management Processes with Automated Risk Assessment. In *Proceedings of the 20th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2009), Venice, Italy, October 27-28, 2009*, volume 5841 of *Lecture Notes in Computer Science*, pages 71–84. Springer, 2009.

[97] J. A. Wickboldt, G. S. Machado, W. L. da Costa Cordeiro, R. C. Lunardi, A. D. dos Santos, F. G. Andreis, C. B. Both, L. Z. Granville, L. P. Gaspary, C. Bartolini, and D. Trastour. A Solution to Support Risk Analysis on IT Change Management. In *Proceedings of the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM 2009), Hofstra University, Long Island, NY, USA, June 1-5, 2009*, pages 445–452. IEEE, 2009.

[98] D. E. Wilkins and M. desJardins. A Call for Knowledge-Based Planning. *AI Magazine*, 22(1):99–115, 2001.

[99] D. E. Wilkins and K. L. Myers. A Multiagent Planning Architecture. pages 154–162. AAAI, 1998.

[100] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2006), Berkeley, California, USA, May 21-24, 2006*, pages 199–213. IEEE, 2006.

[101] L. Zia, Y. Diao, D. Rosu, C. Ward, and K. Bhattacharya. Optimizing Change Request Scheduling in IT Service Management. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2008), Honolulu, HI, USA, July 8-11, 2008*, pages 41–48. IEEE, 2008.

[102] L. Zia, Y. Diao, C. Ward, and K. Bhattacharya. Using Mixed Integer Programming to Schedule IT Change Requests. In *Proceedings of the 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Bahia, Brazil, April 7-11, 2008*, pages 895–898. IEEE, 2008.