# Supporting Planning and Refactoring of Refinement Structure of Event-B Models

# Event-B モデルの詳細化構造の計画とリファクタリングの支援手法

by

Tsutomu Kobayashi
小林 努

A Doctor Thesis
博士論文

Submitted to
the Graduate School of the University of Tokyo
on December 9, 2016
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Information Science and Technology
in Computer Science

Thesis Supervisor: Shinichi Honiden　本位田 真一
Professor of Computer Science

## ABSTRACT

Systematic construction of highly reliable software systems is crucial. Constructing formal specification is one of the most rigorous ways to that end since it enables developers to thoroughly verify target systems from early phases of development. Event-B as one such method, has been attracting much interest from academia and industry because it supports a flexible refinement mechanism that mitigates the complexity of constructing and verifying models of complex target systems by considering multiple abstraction layers of models. Moreover, the refinement mechanism of Event-B enables developers to flexibly select what elements of the target system are introduced in each step of refinement (refinement structure).

Although the refinement mechanism supports comprehensible yet rigorous verification, there are difficulties in exploiting it in software development. First, although most existing studies on Event-B currently focus on reducing the complexity of constructing models, models are used even after being constructed. Constructed models should be maintained, and there is a strong demand to reuse a part of existing models to construct other models. In the area of program code maintenance, highly automated methods and tools to support refactoring have been leveraged. Similar methods should be provided for models of Event-B. Second, in order to construct models, developers need to plan the structure of refinement before constructing models. Although this activity is advanced due to lack of rigorous representation of the specification, it is worth investigating how we can support it. The space of reasonable plans is too large to grasp, and models constructed by following an intuition tend to cause inconsistencies and ineffective use of the refinement mechanism. Therefore, helping analysis of design spaces of reasonable refinement plans is important. However, these problems are currently solved partially or only in specific domains, and there is no generic and systematic approach.

To address the problems above, we focus on the refinement structure and propose methods to explicitly handle the refinement structure from engineering perspective.

As a preliminary work, we propose a generic and systematic view of the refinement of Event-B models. The problems above are generalized on the basis of the generic view and we provide a generic approach to handle the problem.

On the basis of this view, we provide methods to improve the maintainability and reusability of existing Event-B models as an instantiation of the generic approach. The method supports reconstructing the refinement structure of existing models by supporting the construction of models about different sets of variables than those of original models, but that keep consistencies defined in original models. The core of the method supports the decomposition of a refinement step by finding certain model properties from existing models and helping developers to find additional properties from proof for existing models to make new models consistent with the original ones. By combining the decomposing of refinements with the composing of refinements, we provide a method to help developers restructure a refinement chain according to given sets of variables to be considered in each step.

We also tackle the problem of planning the refinement structure for a target system before constructing its models. We view this problem as another instance of the generic problem and propose methods to effectively search reasonable refinement plans and show comprehensible views of the solution space. We define rationales of the refinement structure to avoid invalid refinement and follow common refinement strategies in practice. On the basis of the rationales, we propose a search method that effectively removes invalid and ineffective refinement plans and comprehensibly shows solutions.

In our case studies to evaluate the proposed methods, we succeeded in decomposing large refinement steps in existing models, restructuring existing models to extract reusable parts for construction of other models, and planning reasonable and effective refinement plans from informal but structured descriptions of target systems. Considering the results and discussion on ways to elicit information necessary for our methods, we conclude that our methods can help developers to utilize Event-B and its refinement mechanism in software development.

## 論文要旨

高信頼なソフトウェアシステムの体系的な構築はきわめて重要である．形式的な表現の仕様を構築する手法を用いると開発の早期の段階から対象システムを徹底的に検証することが可能になる．そのため，このような手法は特に厳密に高信頼なシステムの構築を可能にする手法の 1 つであると言える．Event-B という形式仕様記述手法を用いると，その柔軟な詳細化機構により，複雑なシステムについて複数の抽象度を厳密に考慮することでその構築と検証の複雑さを軽減することができる．そのため， Event-B は近年学術界と産業界の両者からの注目を浴びている．その上， Event-B の柔軟な詳細化機構により，開発者は詳細化の各段階で対象システムのどの要素に着目するか（詳細化構造）を柔軟に選択することができる．

この詳細化機構は厳密で包括的な検証を支援するが， Event-B をソフトウェア開発に導入する際には難しさが存在する．第 1 に，現在多くの Event-B に関する研究はモデルの構築の複雑さを軽減することを主眼においているが，モデルは構築された後にも利用される．構築されたモデルは保守されるべきであり，また他のモデルの構築のために既存のモデルを再利用する需要も高い．プログラムコードの保守の分野においては，自動化の程度の大きいリファクタリングの支援ツールが利用されている．我々は Event-B のモデルに対しても同様の手法が必要であると考える．第 2 に，開発者はモデルの構築の前に詳細化構造に関する計画を立てる必要がある．モデルの構築の前には厳密な仕様の表現がないためこの問題はより難しく，自動で支援できることは限られるが，どれだけの支援ができるかを考察する価値があると思われる．問題のない詳細化計画の可能性は大量に存在する一方で，直観に従って立てた計画に沿ってモデルを構築すると整合性のないモデルが構築されたり詳細化機構を活かし切れないモデルが構築されかねないという問題がある．そのため問題のない詳細化のデザインの可能性の分析の支援は重要である．しかし，既存手法ではこの問題は部分的に解かれているのみであるかドメイン依存の手法で解かれており，一般的で体系的なアプローチは提案されていない．

上記の問題に対処するために我々は詳細化の構造に着目し，詳細化構造を明示的・工学的に取り扱う手法を提案する．

まず準備として， Event-B モデルの詳細化の一般的・体系的な捉え方を提案する．我々は上記の問題をこの視点から一般化し，一般的なアプローチを提供する．

我々はこの視点に基づき，上記の一般化したアプローチの具体化として，既存の Event-B モデルの保守性と再利用性を向上させる手法を提供する．この手法は既存の Event-B モデルで定義された整合性を保ち，かつ元のモデルで記述されている変数の集合とは異なる変数の集合に関する記述のなされたモデルの構築を補助することによって詳細化構造の再構築を補助する．この手法の核となるのは既存のモデルから，また既存のモデルに関する証明から適切な記述を開発者が探し出すことを補助し，それらをもとに新しいモデルを構築することで，詳細化ステップの分割を補助する手法である．我々は，この詳細化ステップの分割手法と，詳細化ステップの結合手法とを提案し，それらを組み合わせることによって，与えられた詳細化構造に従って開発者によるモデルの詳細化の再構築を補助する手法を提案する．

我々はモデルの構築前に詳細化構造を計画する問題にも取り組む．我々はこの問題を上記の一般化された問題の別の具体化として捉えて，効果的に問題のない詳細化計画を探索

し，結果の理解しやすいビューを出力する手法を提案する．本提案ではまず問題のある詳細化計画を除き，よく採用される詳細化の戦略に従った出力を得るための理論的な根拠を定義した．上記の根拠に基づき，問題のある詳細化計画や効果的でない詳細化計画を取り除き，結果を分かりやすく一望できるような出力を与える手法を構築した．

提案手法の評価のためのケーススタディにおいて，我々は既存のモデルの大規模な詳細化ステップを分割すること，新しいモデルを構築するために既存のモデルの部品を再利用するために詳細化構造を再構築すること，そして対象システムに関する構造化されているが非形式的な情報をもとに問題がなく効果的な詳細化計画を出力することに成功した．これらの結果の考察に加えて提案手法のために必要な情報の獲得に関する議論を行った結果，我々の提案手法は開発者がソフトウェア開発において Event-B とその詳細化機構を利用することを支援できると結論づけた．

# Acknowledgements

First and foremost, I would like to express the deepest gratitude to my supervisor, Professor Shinichi Honiden, who has cultivated my fundamental attitude toward research activity and given essential advice to me through frequent discussions. In addition, he provided me with an excellent research environment and many quality opportunities to advance my research ability. Without his unfailing and continuous support and encouragement, I would not have been able to achieve my challenging goal.

I would also like to sincerely thank the thesis committee members — Associate Professor Ichiro Hasuo as a chair and Professor Shigeru Chiba, Professor Akihiko Takano, Professor Naoki Kobayashi, and Professor Masami Hagiya — for giving detailed, constructive, and important feedback.

In addition, my heartfelt gratitude extends to Associate Professor Fuyuki Ishikawa, who has guided me and provided lots of insightful feedback through long and detailed discussions. He also connected me with many enthusiastic researchers and students at various opportunities. With his help, I was able to tackle various challenging problems.

I would also like to show my deep appreciation to Professor Alexander Romanovsky for his kind help during my visits at the Centre for Software Reliability of Newcastle University. My appreciation is extended to the researchers and citizens at Newcastle upon Tyne who welcomed and guided me, including Dr. Alexei Iliasov, Dr. Ilya Lopatkin, Dr. Carl Gamble, Dr. Budi Arief, Dr. Minh Tran, Dr. Zoe Andrews, Dr. Richard Payne, Duncan Bennett, and Sarah Bennett. I am indebted to them for interesting and comfortable experiences there, which greatly improved my research and broadened my perspective.

Moreover, I am grateful to prominent researchers who I had discussions with at our laboratory and conferences and who provided me with valuable feedback on my work, including Professor Jean-Raymond Abrial, Professor John Fitzgerald, Professor Cliff B Jones, Dr. Hironobu Kuruma, Dr. Thai Son Hoang, Professor Bashar Nuseibeh, Dr. Yijun Yu, Associate Professor Kenji Tei, Dr. Ryuichi Takahashi, Dr. Kazunori Sakamoto, Dr. Takuo Doi, Dr. Yuta Maezawa, Professor Nobukazu Yoshioka, Professor Yoshinori Tanabe, Dr. Shunichiro Suenaga, Professor Zhenjiang Hu, Professor Fumihiro Kumeno, and Associate Professor Hiroyuki Nakagawa. Research occasions with them have always been inspiring, and I am fortunate to have met and interacted with them.

In addition, I would like to thank all members and visitors of Honiden Laboratory at National Institute of Informatics. Through many memorable shared experiences, I have been helped, amused, comforted, stimulated, and grown. I would like to express special gratitude to former visitors Aivar Kripsaar and Dr. Inna Pereverzeva, who helped me with intensive discussions and software implementation on my work. Their help was indispensable to achieving important results. I am also particularly grateful to members and visitors of the formal methods group and the software engineering group who have discussed our research, including Dr. Taku Inoue, Dr. Johan Nyström-Persson, Fan Jiang, Fernando Tarín, Lasse Berglund, Ksenia Nasedkina, Maxime Chereau, Bui Phi Diep, Nguyen Hoang Anh, Massimo Bortolamei, Hadrien Luttiau, Katsuhiko Ikeshita, Takaya Saeki, Shinnosuke Saruwatari, and Daichi Morita. I would like to extend my gratitude to other colleagues as well, including Soramichi Akiyama, Shouichi Kamiya, Shingo Horiuchi, Naoki Tsurumi, Kohsuke Yatoh, Susumu Tokumoto, Kazuki Nishiura, Keiichiro Hoshi, Hiroki Sawano, Kiichi Ueta, Tomoya Katagi, Ryo Shimizu, Atsushi Watanabe, Yukino Baba, Daisuke Fukuchi, Susumu Toriumi, Hirotaka Moriguchi, Katsushige Hino, Yusaku Kimura, Atsushi Suyama, Nobuaki Hiratsuka, Satoshi Katafuchi, Valentina Balijak, Adrian Klein, Florian Wagner, Yoshiyuki Nakamura, Koichi Fujikawa, Yuki Inoue, Shengbo Xu, Fernando Tarin Morales, Takayuki Suzuki, Shun Lee, Junto Nakaoka, Natsumi Asahara, Yuta Tokitake, Kazuya Aizawa, Miki Yagita, Yasuhiro Sezaki, Katsuhiko Ikeshita, Masaki Katae, Moeka Tanabe, Yasuo Tsurugai, Paul Harvey, Aurélien Vialon, Kazuyuki Honda, Zhen Cui, Ziyuan Wang, Takahiro Sugiura, Keita Tsukamoto, Yetian Mao, Koki Kato, Shunpei Ito, Virginia Brassesco, and Daniel Millevik.

I am grateful to staff at the University of Tokyo and National Institute of Informatics who kindly supported my academic life, including Dr. Masayuki Nii, Masaru Nagaku, Nao Kaneko, Kyoko Oda, Saki Narimatsu, Kazue Kusama, Masaomi Soga, Yuko Yumoto, Kimiko Otani, and Akiko Shimada.

Furthermore, I wish to thank the friends I met in Nada High School, the University of Tokyo, the Shoshikan dormitory, and meet-ups on my hobbies. They have always stimulated my curiosity and broadened my perspective. During my academic life, I shared interesting, relaxing, and refreshing moments with them outside the laboratory. It has been a great pleasure spending time with them.

Lastly and mostly, I would like to express deep gratitude to my father Souichirou, mother Satomi, and sister Natsuko, who have always supported me with love throughout my life.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Citations to Printed Publications

Parts of this thesis have appeared in the following publications.

## Proceedings

1. Tsutomu Kobayashi and Shinichi Honiden, “Towards Refinement Strategy Planning for Event-B,” In Proceedings of Workshop on the Experience of and Advances in Developing Dependable Systems in Event-B (DS-Event-B 2012), pp. 72–81, October 2012.

2. Tsutomu Kobayashi, Fuyuki Ishikawa, and Shinichi Honiden, “Understanding and Planning Event-B Refinement through Primitive Rationales,” In Proceedings of The 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2014), pp. 277–283, June 2014.

3. Tsutomu Kobayashi, Fuyuki Ishikawa, and Shinichi Honiden, “Refactoring Refinement Structures of Event-B Machines,” In Proceedings of The 21st International Symposium on Formal Methods (FM 2016), pp. 444–459, November 2016.

4. Tsutomu Kobayashi, Fuyuki Ishikawa, and Shinichi Honiden, “Stepwise Refinement of Software Development Problem Analysis,” In Proceedings of The 35th International Conference on Conceptual Modeling (ER 2016), pp. 488–495, November 2016.

# Chapter 1

# Introduction

## 1.1 Background

Software is a vital infrastructure of society, and systematic construction of highly reliable software systems is crucial. Formal methods, namely constructing the specifications of target systems with formal notations and verifying them, have been studied and practiced to increase the reliability of software systems. In particular, some formal specification methods support the *stepwise refinement* approach, which gradually converts initial specifications to make specifications closer to implementation, have been considered to be useful. Program code (or other kinds of concrete software artifacts) obtained with stepwise refinement is considered to be *correct by construction*, meaning it is guaranteed to be a concrete version of the initial specification by a systematic refinement mechanism.

Nowadays, a few decades after the proposal of classical formal methods, both the scale and the application area of software have become significantly large. As a result, today's software is expected to interact with various components of an environment and form highly complex *systems* as a whole. Therefore, a complex process of inspecting the interaction between software and its environment is important. However, the starting point of that process, namely constructing an initial specification of such systems, is difficult due to its complexity. Thus, there have been demands for a rigorous method for constructing initial specifications of target systems that satisfy requirements.

Event-B [2] has been proposed to address this problem and has been gaining much attention from academia and industry [68, 6]. It aims at modeling systems and reasoning on models to construct a comprehensive initial specification that is correct by construction. To this end, Event-B supports a flexible stepwise refinement mechanism (*horizontal refinement*) for rigorous and evolutional modeling. In Event-B, developers can construct a simple and abstract model first, then construct a concrete model that includes more aspects of the target system, and then check both the consistency of models themselves and the consistency between the abstract model and the concrete one. Developers can check such consistencies with theorem proving in a set-theoretic mathematical language. Proof obligations are systematically generated by the tool platform of Event-B so that developers can concentrate on proofs. By repeating this process, a comprehensive specification can be constructed in a stepwise and flexible manner, and thus, the method mitigates the complexity of modeling and the verification of complex system models.

Because of its advantages, Event-B has been used in various industrial projects.

In Japan, six major software vendors and a research institute formed Dependable Software Forum (DSF) [35], which is a joint research forum that aims to apply advanced methods to reduce failures of software systems. A working group of DSF chose Event-B as the primary method to be applied to its projects, including complex enterprise systems, and published resulting idioms and a guideline [7]. In addition, researchers in Japanese industry have been studying applications and extensions of Event-B, such as crossed-project reference of Event-B models [50].

A large 4-year project in the EU, named Deploy [30, 68] and involving 15 organizations from academia and industry, intended to apply Event-B to large-scale industrial software systems. Applications of the project include many challenging and interesting projects in various areas. Researchers in SAP AG applied Event-B to model business software that involves complex configuration options and business processes. A team with Space Systems Finland Ltd. constructed models of satellite systems from Ada code as a document. A Project with Robert Bosch GmbH applied Event-B to model automotive systems and dealt with problems with requirements analysis and timed systems. A team of XMOS modeled the ISA of XCore, a commercial microprocessor made by the company. The model of XCore [75] consists of 97 components, and it is one of the largest publicly available Event-B models. The Deploy project produced highly successful results as not only products and documents but also methodologies and their tool implementations.

## 1.2 Problem and Motivation

The unique refinement mechanism of Event-B allows developers to flexibly choose what elements and aspects are introduced in each step of refinement. We focus on this point, *refinement structure*, and address problems in exploiting Event-B in systems development. To this end, we formalize the general problem of selecting refinement structure and provide concrete solutions based on the general problem. The remainder of this section describes the concrete problems and motivations of our work.

### 1.2.1 Refactoring

Maintenance of system artifacts — including eliminating faults, dealing with changes of environment and requirements, and extending functionality — is necessary in software development. Thus, improving maintainability of system artifacts is crucial both during and after development. Although maintenance of Event-B models has not yet been actively studied, it is obviously important to improve their maintainability.

In fact, the experiences from Deploy project [68] highlight the importance of improving the maintainability of Event-B models. Reports on development of satellite systems and automotive systems describe challenges of dealing with changing requirements and evolving models. In addition, many deliverables reported the development of Event-B models driven by teams, where understandability of Event-B models is considered crucial.

Improving reusability is another important activity in software development. To improve reusability of program code, various approaches — including modularization, object-oriented development, aspect-oriented development, and component-based development — have been proposed. Event-B and its extensions also facilitate reuse of artifacts (i.e., models). In modeling in Event-B, an abstract model of a target system tends to be generic enough to be used as an abstract model of another target system. Thus, Event-B's refinement mechanism encourages reusing models.

Reusing Event-B models is considered important since formal modeling and verification cost. Researchers involved in industrial projects have stressed this importance because of the cost of modeling and verification in Event-B [68]. For example, reports on the development of satellite systems and automotive systems describe the importance of reuse and evolution of Event-B models. Extensions of Event-B with patterns [46], abstract data structure [73], and the decomposition/-composition of models [5, 18, 38] have been proposed from such experiences.

There have been many studies on restructuring and modularizing program code. In particular, *refactoring* [36], namely reformation of the structure of artifacts that preserves their behavior, is considered an effective approach for improving maintainability and reusability, and thus, it has been widely studied and practiced for program code. Its benefits include improvement of extensibility, modularity, reusability, understandability, and maintainability of source code. Refactoring formal specifications has also been studied, including inlining, move, transformation, and parameterization of expressions in formal specifications.

The refinement structure of Event-B models highly affects their maintainability and reusability, because it reflects aspects that are considered in each abstraction level. For example, despite the refinement mechanism, introducing too many aspects in one refinement step is common in practice due to dependencies between requirements and elements of target systems. Models constructed in such a way tend to be difficult to comprehend and thus lack maintainability. Another common situation is that a developer who plans to reuse existing Event-B models finds that reusable aspects were introduced in later steps of refinement (concrete models), and therefore, it is difficult to separate them from non-reusable aspects in the same steps. Thus, although the refinement mechanism facilitates the reuse of Event-B models, it is not straightforward.

Despite refinement structure being important for maintainability and reusability, the refactoring refinement structure has not been studied. The existing refactoring method for Event-B models is only for renaming identifiers in models.

As a foundation to support refinement restructuring, we propose a method for decomposing refinements. In particular, for given consistent (i.e., proven) machines $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ such that $M_{\mathrm{C}}$ refines $M_{\mathrm{A}}$, our method automatically constructs a part of an *intermediate* machine $M_{\mathrm{B}}$ and helps developers to manually complement it so that $M_{\mathrm{B}}$ is consistent with given machines, namely $M_{\mathrm{C}}$ refines $M_{\mathrm{B}}$ and $M_{\mathrm{B}}$ refines $M_{\mathrm{A}}$. This enables users to decompose a refinement step into several substeps. The decomposition method can be combined with the merging of refinements, which is simpler than decomposition, to restructure refinements.

We show the usefulness of refinement restructuring through two case studies. The first shows how decomposing large-scale refinements can improve the maintainability of existing machines. The second shows how to extract parts of existing machines and reuse them for constructing new machines of another system that, at first glance, looks different from the original system.

### 1.2.2 Planning

Because of the flexibility of the refinement mechanism of Event-B, it is necessary for developers to plan the refinement structure of Event-B models. The planning should be done before the construction of models because changing the refinement structure causes changes of the model description, and thus, a developer needs to redo the construction and verification of a model.

However, planning a refinement structure is difficult. Unless developers have experience constructing models of a system that is similar to the target system, planning is not straightforward, and thus, they need to repeat trial and error. Although developers have many choices of model elements in each step, too many choices can lead to difficulty in deciding plans and a failure of proving. When an unstructured requirements document of the target system is given, developers are usually unsure of what to do to even start planning. Moreover, due to consistency checking, a naive plan, following the intuitive "from abstract to concrete" prescription, may fail and cause rollback (e.g., when possible state changes are unspecified in a previous step). If developers fail to do sufficient proving, they may need to subsequently undertake a demanding task, namely reconstruction and proving the models. Although the number of developers using Event-B is increasing, experienced developers tend to rely on their artisanship and repeat trial and error to plan refinement structures.

Reports about industrial experiences show that planning is one of the most difficult and important parts in modeling in Event-B. The importance and difficulty of planning refinement have been reported from experiences of developing satellite systems, developing systems on military aircraft [68], and providing lecture courses on Event-B [7]. A report from DSF [7] describes that the skill of planning refinement is difficult to acquire, and engineers who attended a lecture by DSF tried to learn the artisanship of refinement planning by reading a textbook about Event-B [2]. The report also mentions the large cost of rework from an experience of a team who constructed a model by following an inappropriate plan.

To the best of our knowledge, most of the existing studies have never tried to solve the problem. There are teaching materials for Event-B, such as the book by the founder of the method [2], as well as a number of application studies. However, they only report successful experiences of modeling and do not discuss how they developed the refinement plans or how good the plans are compared with other possible plans. Other studies provide methods for constructing Event-B models, but some only provide one way [62], while others provide only domain-specific knowledge [86] and thus lack generality. Moreover, they do not discuss how and why the methods and guidelines derive good refinement plans. Therefore, there is a demand to provide a generic method to support planning and analyzing refinement structure.

We propose a method to derive good refinement plans by analyzing the system's elements, functionalities, and dependencies. Moreover, we implemented a tool as a proof of concept and applied the tool to several systems with different properties and analyzed the relationships between properties of the systems and the results (Section 5.3). We describe how our method supports planning and comparison of possible refinement plans of several target systems through a case study.

## 1.3 Contributions

In this section, we summarize our contributions in this study.

To address challenges in development using Event-B, we explicitly manipulate refinement structure — that is, what aspects are introduced in each step of refinement — from the engineering perspective. Our proposals below help developers to plan and reconstruct refinement structure by considering the dependency and consistency to control usability, comprehensibility, and complexity of Event-B models.

- Definitions of refinement structure as means to explicitly handle differences between an abstract model and its concrete model.
- Methods for helping developers to refactor the refinement structure of existing Event-B models.
- Methods for helping developers to explore possible refinement structures.

In addition, we discuss case studies we conducted as follows:

- Case studies demonstrating that refactoring method can improve maintainability and reusability of existing Event-B machines.
- A case study to analyze possible refinement structure of various target systems and their characteristics.

Finally, we also discuss a possible preliminary phase for our methods.

- Discussion on a method to elicit information on refinement by analyzing problem structures of target systems.

## 1.4 Organization

The rest of this thesis is organized as follows.

**Chapter 2** We provide background on modeling and verification in Event-B with an example to explain our methods.

**Chapter 3** We describe basic notions that are important for our approach, such as refinement structure and implicit predicates, in Sections 3.1 and 3.2. We then explain problems we tackle and our approaches in Section 3.3.

**Chapter 4** We propose a method for supporting refactoring of the refinement structure of existing Event-B machines. In Section 4.4, we describe a method to decompose a refinement step into two refinement steps by choosing a valid set of variables and slicing from the existing machines, and to help developers manually complement the model under construction to have consistencies with the original models. To realize restructuring of the refinement structure, we then extend the decomposition method by combining it with a merging of two refinement steps into a single step in Section 4.5. We also explain the feasibility of automating our methods in Section 4.6.

**Chapter 5** We tackle an advanced problem, namely planning the refinement structure before construction of actual modeling. In Section 5.2, we define rationales for planning so that our method eliminates obviously meaningless refinement plans and reflects common refinement strategies in practice. We then, in Section 5.3, explain our method for constructing plans of refinement and generating a simple view of them.

**Chapter 6** We conduct case studies to evaluate the usefulness of our methods. Case studies of our refactoring method and our planning method are explained in Sections 6.1 and 6.2, respectively.

**Chapter 7** We describe a possible approach to analyze problem structures of target systems for finding information that strongly affects refinement strategies. We explain the problem and motivation in Section 7.1. In Section 7.2, we give a brief description of the problem diagram, which we use as informal representation of target problems. Then we describe various kinds of information of refinement in Section 7.3. We show our method to find such information in Section 7.4. A preliminary experiment on the method is explained in Section 7.5.

**Chapter 8** We discuss on our methods and the results of case studies. In Sections 8.1 and 8.2, discussions from the results of case studies (Chapter 6) are described. Discussions on preliminary phases (Chapter 7) are described in Section 8.3. The discussions are summarized in Section 8.4.

**Chapter 9** We present related work to our methods in the areas of support of modeling and refinement in Event-B, arrangement and refactoring of formal specification and proof, arrangement of requirements and informal models, bridging informal artifacts and formal artifacts, and application of Craig interpolation.

**Chapter 10** Finally, we conclude this study and give directions for future work.

# Chapter 2

# Background on Event-B

## 2.1 Event-B

### 2.1.1 Classical Approaches for Software Reliability

Studies on verification methods and validation methods for enhanced software reliability have a long history and a large variety of approaches.

Software testing is one of the dynamic verification methods and has been one of the most active areas of software engineering research. In software development projects in industry, testing is indispensable. However, although testing effectively reveals the existence of faults, it is not enough for the safety critical systems because it is not exhaustive [31].

Formal methods have been proposed to improve the reliability of software. Representing target systems in a formal manner has the following merits: First, it helps developers to find faults in an early phase of development, thus significantly reducing the cost of repairing them. A study [80] reports that the cost to repair a defect in the beta-testing phase is around 15 times the cost of dealing with the same defect in the requirements gathering analysis/architectural design phase. Second, formal specifications can be implemented in various ways because they are usually independent of implementation details. Finally, rigorous expressions facilitate scientific analysis and verification on the basis of mathematical logic.

Model checking is a major formal method approach that inspects the specification of state transition of particular behavior of software. A model checker thoroughly and automatically searches the state space of target specification and checks whether the specification satisfies requirements. It gives a counterexample in cases where specification does not satisfy given requirements. It is especially effective for inspecting the complex path and order of program executions. Many model checkers — such as SPIN [48], SMV [21], and Java PathFinder [43] — have been proposed and widely used in practice.

Another family of major formal method approaches relies on mathematical deduction for reasoning. A typical method of such approaches involves theorem proving on the formal description of data structure and operations of the whole target system. Although such methods require that users are experienced in mathematical reasoning, they are effective for reasoning on data structure and each operation.

Some of such methods support an approach called *stepwise refinement* [84]. Stepwise refinement aims at correctly refining the correct specifications to obtain the correct program code and realizes the code development concept that is *correct by construction*. Such methods are based on mathematical foundation for stepwise refinement that has long been studied [47, 10].

Notable methods that facilitate stepwise refinement include several methods proposed by Jean-Raymond Abrial. Z notation [74], which represents specifications of program with set-theoretic notation and first-order logic, was proposed in 1977. It has been standardized as ISO/IEC 13568:2002 and applied to various projects including large-scale industrial ones [44].

In the early 1990s, Abrial proposed B method [1], which can be considered a successor of Z notation. With influence from another famous formal specification method, VDM [54], specifications of B method are based on notation of *abstract machines* with operations and have clearer correspondence with programs. The refinement mechanism of B method enables rigorous construction of implementation specification from initial specification by considering the weakest precondition of operations. Tool implementation, such as Atelier B [23], supports generation of proof obligations for correctness of refinement and discharging some proof obligations using automatic provers. The B method has been applied to many large-scale industrial projects, including ones for developing the auto-pilot system of a Paris metro line (Météor project) [13] and shuttles in an airport [11].

Nowadays, the scale and complexity of software systems have significantly grown. The primary causes of the complexity of software systems include interactions of software and its environment [52] because the application area of software has widened. Such interactions are key properties in recent remarkable system paradigms, such as cyber physical systems, system of systems, and systems of engagement.

Although the correct by construction approaches described above can be used to derive programs in such complex systems in theory, they rely on the validity of the initial specification. Because it is difficult to check whether the initial specification of such complex systems is valid, this complexity prevents developers from applying classical correct by construction approaches in practice. This is a problem of complexity in *systems engineering*. In other words, developers need to face the complexity of systems including developed software and its environment, in the design phase before constructing programs.

Therefore, it is important to analyze the whole system and problems that should be solved by constructing programs so that developers can construct specification that satisfies requirements on applications. To this end, developers need to discuss the whole system by reasoning on models about functionalities of a software system and assumptions on its environment.

This approach—validation of problems—is different from validation of programs, which checks whether programs satisfy software properties such as no access to null pointer. In Abrial's book [2], he explained this as follows:

> In doing this as engineers, we are not supposed to instruct a computer; rather, we are supposed to instruct ourselves.

B method and other verification methods — such as testing, abstract interpretation, and model checking — are usually used to validate not initial specifications but programs.

### 2.1.2 Features of Event-B

To address this problem, Abrial proposed a successor of B method named Event-B [2]. The refinement mechanism of B method aims to design programs with a fixed interface, and it does not allow changing extensional elements of the target system, such as components and input/output of them. This restriction prevents the construction of complex initial specification of the whole system. Conversely, refinement in Event-B is designed to allow the addition of components and functionalities to facilitate flexible and rigorous construction of complex initial specification, including that of the target system's interfaces.

In addition, the refinement of B method consists of simple steps, such as translating set operations into array operations, and thus can be automated (BART tool [67]). In contrast, by using refinement in Event-B, developers can choose what aspects are to be handled in which order in each step, as well as what elements (e.g., variables) are introduced to specify each aspect.

Event-B has been applied to various large-scale projects, such as the Deploy project [30] and Advance project [6] of the EU. Moreover, there has been a proposal to apply Event-B to advanced systems, such as enterprise systems [16] and hybrid systems [77, 12].

Formalism of Event-B is influenced by those of Action Systems [9, 20], TLA+ [56], and UNITY [65]. The behavior of target systems is represented with guarded commands as *events*, and thus, Event-B models are considered descriptions of observed phenomena of target systems rather than a high-level programming language.

The refinement mechanism of Event-B supports *horizontal refinement* that allows evolutionary modeling based on refinement (i.e., gradual addition of elements of the target system through refinement). Horizontal refinement helps developers to do rigorous modeling while mitigating the complexity of modeling and verification by distributing it amidst multiple steps. Contrary to refinement in B method (*vertical refinement*), which has an inherent order of refinement, refinement of Event-B has many possible orders. This flexibility enables complexity mitigation, but it poses unique difficulties, as described in Section 1.2.

### 2.1.3 Modeling in Event-B

In modeling in Event-B, developers construct specifications with a set of *machines*. After constructing an abstract machine, they introduce more aspects of the target system by constructing a new machine with more details and verifying the consistency between the new machine and the abstract one.

In modeling in Event-B, developers first construct a simple abstract model and then iteratively construct concrete models by gradually introducing various elements of the system. For example, in the first example in a book [2] by the founder of Event-B, models are constructed, as illustrated in Figure 2.1. The target system is a controller of traffic lights that regulate the movements of cars that enter and leave a bridge from the mainland to an island. The regulations include the capacity of the whole bridge and island, and the fact that the bridge is one way. In the initial model (illustrated on the left), the bridge and the island are abstracted to being outside the mainland by combining them. The model specifies the properties as *invariants* (e.g., number of cars outside the mainland is less than or equal to its capacity) and behaviors as *events* (e.g., number of cars outside the mainland increases) of cars leaving and entering the mainland. The second model treats the bridge and island separately and incorporates new properties, such as the one-way property. The third model considers properties of traffic lights that are consistent with the properties discussed in the previous models. After that, more elements, such as car sensors and detailed functionalities of the controller, are introduced through further refinement.

Once a model is constructed, it is verified in terms of its own self-consistency and its consistency with previous models. The consistency properties include the application constraints described above and rules imposed by Event-B. In addi-

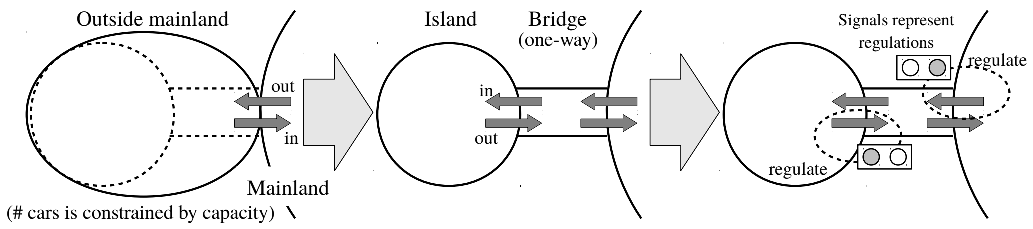


Figure 2.1: Example: Cars on an island and bridge

tion to consistency as a specification, developers want to prove that an application satisfies requirements. Refinement enables them to do this in multiple steps instead of proving in one shot. Although application constraints are basically about safety (i.e., something bad never occurs), because they are expressed as invariants, recent studies [45] propose methods to prove liveness properties (i.e., something good ultimately occurs) by expressing them as multiple lemmas in Event-B notation. Valid refinement guarantees that behaviors of the concrete model simulate those of its abstract model, and properties that hold for an abstract model also hold for its concrete model. Therefore, the refinement mechanism decomposes a task for proving that a property holds for a concrete model into two smaller tasks, namely proving that the property holds for an abstract machine and proving the consistency of refinement. Thus, refinement helps developers mitigate the complexity of verification.

The Rodin platform [3], which is the development environment of Event-B, automatically generates proof obligations for consistencies and helps developers discharge them by providing automatic provers and an interface for manual proof. Thus, developers can use automatic provers to discharge simple proof obligations and try to manually discharge proof obligations that cannot be discharged automatically. Proof obligations that cannot be discharged indicate that there are inconsistencies in models and that a developer needs to modify models. Rodin can be extended by providing plugins, and various extensions have been proposed and implemented [33].

## 2.2 Structure of Event-B Specifications

A model in Event-B is composed of contexts and machines. Static properties of the target system are specified in *contexts*, whereas the system's dynamic properties are specified in *machines* as predicates of invariants and events.

Figure 2.2 shows the structure of a context.

In the *sets* clause, user-defined data types are declared as *carrier sets*. All *constants* and *variables* in Event-B must be typed using carrier sets or primitive types such as Boolean values and natural numbers. In the *constants* clause, constants of the target system are declared. In the *axioms* clause, properties of carrier sets and constants are described. Carrier sets are often declared as a set of multiple constants, as shown in Figure 2.2. Types of constants are also described in this clause. A context can be defined as an extension of another context. If context $C$ extends another context $D$, carrier sets, constants, and axioms defined in $D$ can be referred in $C$.

Machines can refer to specifications in contexts. The main part of a specification of events consists of guards and actions. Guards of an event describe the necessary condition for executing the state transitions of the event. Actions describe the state transitions of an event with *before-after predicates* (BAPs), which

```
<context_identifier>
extends extended_context
sets
  TRAFFIC_LIGHTS_COLOR
constants
  red
  green
axioms
  axm_1:
       TRAFFIC_LIGHTS_COLOR = {red, green}
  axm_2: red ≠ green
```

Figure 2.2: Structure of Event-B context

```
    variables: a, b

    Event evt_a
      when
        grd_a1: 0 ≤ a
      then
        act_a1: a :| a′ = a + 1
        act_a2: b :| b′ = b + 2
      end
```

```
    typ_a: {a, b} ⊂ ℕ

    Event initialisation
      begin
        init_a1: a :| a′ = 0
        init_a2: b :| b′ = 0
      end
```

Figure 2.3: Abstract machine `ma`

are relationships between the pre and post states of variables.

For example, a machine `ma` (Figure 2.3) has specifications of variables $a$ and $b$, invariant `typ_a`, and events `initialisation` and `evt_a`. An action is composed of the variables that are changed by the action and a BAP. In BAPs, the after states of variables are expressed using variables with primes, such as $a'$. In Figure 2.3, event `evt_a` increases the values of $a$ and $b$ by 1 and 2, respectively, and it can be executed if $0 \leq a$.

In an *invariants* clause, invariants not only about variables of the current machine but also variables of the refined machine can be specified, while guards can only use variables of the current machine. If an invariant has both variables of abstract machine and those of concrete machine, the invariant is called a *gluing invariant*. Thus, gluing invariants semantically "glue" together the state spaces of two specifications. For instance, in the island example, the first refinement (a specification depicted in the middle of Figure 2.1) has a gluing invariant such that "the number of cars outside the mainland is equal to the sum of number of cars on the bridge and number of cars on the island." This gluing invariant describes semantic relationships between a variable of the initial specification (number of cars outside the mainland) and variables of the first refinement (the number of cars on the bridge and the number of cars on the island). Gluing invariants are very important for considering and planning refinements in Event-B. Without gluing invariants, it is impossible to reason about the consistency of refinements (Section 2.5).

| | |
|---|---|
| `typ_a` | $\{a, b\} \subset \mathbb{N}$ |
| BAP of `act_a1` | $a' = a + 1$ |
| BAP of `act_a2` | $b' = b + 2$ |
| ... | ... |
| $\vdash$ | $\vdash$ |
| Modified `typ_a` | $\{a', b'\} \subset \mathbb{N}$ |

Figure 2.4: Invariant preservation of `typ_a` by `evt_a` in `ma`

## 2.3 Consistency of a Machine

The Rodin platform [3] generates sequents as *proof obligations* (POs) based on specifications on a machine. Consistency of a machine is guaranteed by proving POs. When POs cannot be discharged, developers need to modify the specification.

There are several POs to guarantee a machine's consistency. A primary kind of such POs is *invariant preservation* (INV), which is generated for each pair of an event and an invariant that contains variables affected by the event. INV about an event $e$ and an invariant $i$ (called $e/i$/INV) means that $i$ still holds after an occurrence of $e$. The formal notation of $e/i$/INV is as follows:

$$A \wedge I \wedge G \wedge \mathit{BA1} \Rightarrow i[\mathbf{v}'/\mathbf{v}],$$

where $A$, $I$, $G$, $\mathit{BA1}$, and $i[\mathbf{v}'/\mathbf{v}]$ respectively denote axioms of the referred context, invariants of the machine, guards of $e$, before-after predicates in actions of $e$, and a substitution of all free occurrences of variables $\mathbf{v}$ in $i$ by after-states of $\mathbf{v}$ ($\mathbf{v}'$).

For example, the PO `evt_a/typ_a`/INV of `ma` is as shown in Figure 2.4.

## 2.4 Refinement

In modeling in Event-B, new aspects and details are gradually introduced to a machine through a refinement mechanism. A machine $M_{\mathrm{C}}$ can be defined as a refinement of another machine $M_{\mathrm{A}}$. Here, $M_{\mathrm{C}}$ and $M_{\mathrm{A}}$ are called a concrete machine and an abstract machine, respectively.

We use the symbols $V_{\mathrm{A}}$ and $V_{\mathrm{C}}$ to denote $M_{\mathrm{A}}$'s variables and $M_{\mathrm{C}}$'s variables, respectively. The invariants in a concrete machine $M_{\mathrm{C}}$ can refer to $V_{\mathrm{A}}$ in addition to $V_{\mathrm{C}}$, whereas events of a concrete machine must not be written with variables in $V_{\mathrm{A}} \setminus V_{\mathrm{C}}$. Those that refer to both variables in $V_{\mathrm{A}}$ and those in $V_{\mathrm{C}}$ are called *gluing invariants*, because they connect the state spaces of two machines.

$V_{\mathrm{C}}$ does not need to be a superset of $V_{\mathrm{A}}$. If $V_{\mathrm{A}} \not\subseteq V_{\mathrm{C}}$, some of the variables in $V_{\mathrm{A}}$ are *replaced* with some of the variables in $V_{\mathrm{C}}$. In such a replacement, developers also need to provide gluing invariants that refer to the replaced variables (in $V_{\mathrm{A}}$) and replacing variables (in $V_{\mathrm{C}}$), in order to prove consistency between an abstract machine and a concrete machine.

Moreover, events in $M_{\mathrm{C}}$ may refine events in $M_{\mathrm{A}}$. Concrete events, which refine events in the abstract machine (abstract events), need to have guards that are stronger than the guards of abstract events. Also, the actions of concrete events should simulate the actions of their abstract events.

For instance, suppose that machine `mc` (Figure 2.5) [1] is defined as a refinement of `ma` (Figure 2.3). In `mc`, a variable $a$ is inherited from `ma`; variables $c$, $d$, $e$, and $f$

[1]Assume that a function mod2($n$) that returns $n$ modulo 2 is defined in a context.

```
variables: $a, c, d, e, f$
```

```
typ_c: $\{a, c, d, e, f\} \subset \mathbb{N}$
gluinv_c1: $b = c + d$
inv_c1: $\mathrm{mod2}(a + e) = 0 \Rightarrow a < 1$
inv_c2: $\mathrm{mod2}(e + f) = 1$
```

```
Event evt_c
  refines evt_a
  when
    grd_c1: $0 \leq a \land 0 \leq c$
    grd_c2: $\mathrm{mod2}(a + f) = 0$
  then
    act_c1: $a$ :| $a' = a + 1$
    act_c3: $c$ :| $c' = c + 1$
    act_c4: $d$ :| $d' = d + 1$
    act_c5: $e$ :| $e' = f + 2$
    act_c6: $f$ :|
        $f' = f + 3$
  end
```

```
Event initialisation
  begin
    init_c1: $a$ :| $a' = 0$
    init_c3: $c$ :| $c' = 0$
    init_c4: $d$ :| $d' = 0$
    init_c5: $e$ :| $e' = 1$
    init_c6: $f$ :| $f' = 2$
  end
```

Figure 2.5: Concrete machine `mc`

are newly introduced; and a variable $b$, which is specified in `ma`, has disappeared. The gluing invariant `gluinv_c1` describes the relationship among $b$, $c$, and $d$. Event `evt_c` is defined as a concrete event of `evt_a` of `ma`.

The refinement mechanism enables two styles of refinement, namely, gradual addition of concrete elements (*horizontal refinement*) and transformation of expressions to make them closer to the implementation (*vertical refinement*).

## 2.5 Consistency of Refinements

POs are generated not only for a machine's self-consistency but also for consistency of a machine with its abstract machine.

One of the primary kinds of POs for consistency of a refinement is called *guard strengthening* (GRD), which is generated for each refinement of an event. GRD about a guard $g$ of an abstract event $e_\mathrm{A}$ that is refined by a concrete event $e_\mathrm{C}$ means that $e_\mathrm{C}$ is only enabled if $g$ holds. The formal notation of $e_\mathrm{A}/g/$GRD is as follows:

$$A \land I \land J \land H \land W \Rightarrow g,$$

where $J$, $H$, and $W$ respectively, denote invariants of the concrete machine, guards of the concrete event, and witnesses of the concrete event. Note that $I$ here denotes the conjunction of invariants of not only a machine directly refined by the concrete machine but also machines that are recursively refined.

Another important PO is called *action simulation* (SIM), which is generated for each refinement of an event. SIM about an action $a$ of an abstract event $e_\mathrm{A}$ that is refined by a concrete event $e_\mathrm{C}$ means that state transitions of $e_\mathrm{C}$ are not contradictory with $e_\mathrm{A}$'s state transitions. The formal notation of $e_\mathrm{A}/a/$SIM is as follows:

$$A \land I \land J \land H \land W \land \mathit{BA2} \Rightarrow a,$$

where *BA2* denotes before-after predicates of the concrete event.

Moreover, PO of invariant preservation is also generated for a refinement with an extended form as follows:

$$A \land I \land J \land H \land W \land \mathit{BA2} \Rightarrow i[\mathbf{v}'/\mathbf{v}].$$

For example, the PO `mc/evt_c/inv_c1`/INV is shown in Figure 2.6.

| | |
|---|---|
| `grd_c2` | $\text{mod2}(a+f)=0$ |
| BAP of `act_c1` | $a' = a+1$ |
| BAP of `act_c5` | $e' = f+2$ |
| $\dots$ | $\dots$ |
| $\vdash$ | $\vdash$ |
| Modified `inv_c1` | $\text{mod2}(a'+e')=0 \Rightarrow a' < 1$ |

Figure 2.6: Invariant preservation of `inv_c1` by `evt_c` in `mc` (provable)

# Chapter 3

# Approach

## 3.1 Refinement Structure

The main objective of modeling in Event-B is verification of certain model properties concerned with functionalities of the state space and state transitions. Such properties are expressed as predicates, such as invariants, guards, and before-after predicates. Consistencies of models are verified by discharging POs, which are generated as instances of PO patterns (such as INV, GRD, and SIM) with predicates in the models. To specify predicates, developers need to declare variables of predicates in models. We treat variables and predicates as primary objects of our approach.

The refinement mechanism of Event-B is designed for gradual addition of aspects of target systems to models while keeping properties of abstract models. Thus, a refinement in Event-B involves gradual addition of variables and predicates to models. One of the most important features of Event-B is the flexibility of the refinement mechanism. Developers have many choices of what variables are added to models in each step.

In this thesis, we are interested in what variables are declared in each step of refinement. We call a sequence of sets of variables declared in each refinement step *refinement structure*. For example, a possible refinement structure for modeling a target system about aspects that correspond to variables $v_1$, $v_2$, $v_3$, and $v_4$ is $[\{v_1\}, \{v_1, v_2, v_3\}, \{v_1, v_2, v_3, v_4\}]$. This structure denotes that there are three steps of refinement, and sets of variables $\{v_1\}$, $\{v_2, v_3\}$, and $\{v_4\}$ are introduced in the first, second, and third step, respectively.

Refinement structures dominate Event-B models because predicates in each refinement step are specified as a projection of properties of the target system onto state spaces of variables of the step. Moreover, as we discuss later, refinement structures affect effectiveness of Event-B's refinement mechanism, such as effectiveness for improving understandability, maintainability, and reusability of models and effectiveness for mitigating complexity of modeling and verification. Therefore, it is important to study refinement structure.

## 3.2 Explicit Predicates and Implicit Predicates

Obviously, some predicates that are not explicitly specified in a model can be inferred from the model. For instance, a part of a concrete machine of traffic example (described in Section 2.1.3) is shown in Figure 3.1. It indicates that a predicate $ml\_tl = red$, which is not directly written in the model, holds in an event `IL_out_red_concrete`, since `grd_c1` ($il\_tl = green$) and `inv_c7` ($ml\_tl = red \lor il\_tl = red$) imply it.

```
variables : a, b, c, ml_tl, il_tl
```

```
inv_c1 : ml_tl ∈ {red, green}
inv_c2 : il_tl ∈ {red, green}
inv_c3 : ml_tl = green ⇒ c = 0
inv_c4 :
      ml_tl = green ⇒ a + b + c < d
inv_c5 : il_tl = green ⇒ a = 0
inv_c6 : il_tl = green ⇒ b > 0
inv_c7 : ml_tl = red ∨ il_tl = red
```

```
Event IL_out_red_concrete
  refines ...
  when
    grd_c1 : il_tl = green
    grd_c2 : b = 1
  then
    bap_c1 : b :| b′ = b − 1
    bap_c2 : il_tl :|
          il_tl′ = red
    bap_c3 : c :| c′ = c + 1
  end
```

Figure 3.1: Part of concrete machine (about $\{a, b, c, ml\_tl, il\_tl\}$) of traffic example

```
variables : a, b, c, ml_tl
```

```
inv_c1 : ml_tl ∈ {red, green}
inv_c3 : ml_tl = green ⇒ c = 0
inv_c4 :
      ml_tl = green ⇒ a + b + c < d
```

```
Event IL_out_red_abstract
  refines ...
  when
    grd_x : ml_tl = red
    grd_c2 : b = 1
  then
    bap_c1 : b :| b′ = b − 1
    bap_c3 : c :| c′ = c + 1
  end
```

Figure 3.2: Part of abstract machine (about $\{a, b, c, ml\_tl\}$) of traffic example

Although such predicates are not always directly written for the sake of simplicity, developers can use them as hypotheses for proof of consistency of models. We call such predicates *implicit* predicates and predicates that are directly written in models *explicit* predicates.

In this thesis, we study refinement structure, namely the order of introducing elements and aspects to construct a model of a target system through refinement. Thus, we are interested in the difference between models of a target system with different sets of variables.

Whether a predicate is explicit, can be implicit, or is unspecifiable differs according to variables of models. For example, Figure 3.2 shows an abstract version of the machine shown in Figure 3.1. All invariants, guards, and before-after predicates in the abstract machine except a predicate of guard `grd_x` ($ml\_tl = red$) are also specified in the concrete machine. `grd_c1` ($il\_tl = green$) and `inv_c7` ($ml\_tl = red \lor il\_tl = red$) imply `grd_x` is unspecifiable in the abstract machine,since the variable $il\_tl$ in the predicates is not declared in the abstract machine. Thus, `grd_x` is implicit in the concrete machine, but it is explicitly written in the abstract machine.

Figure 3.3 shows explicit predicates and implicit predicates of the abstract machine and the concrete machine. Predicates such as `grd_c2`, `bap_c1`, and `bap_c3` are explicit in both the abstract machine and the concrete machine. Some predicates — such as `inv_c7`, `inv_c2`, and `inv_c5` — are explicitly specified in the concrete machine but unspecifiable in the abstract machine. `grd_x` is implicit in the concrete machine but explicitly specified in the abstract machine.

In other words, predicates that are explicitly written in an abstract model are inherited through refinement, but they may become implicit in concrete models.

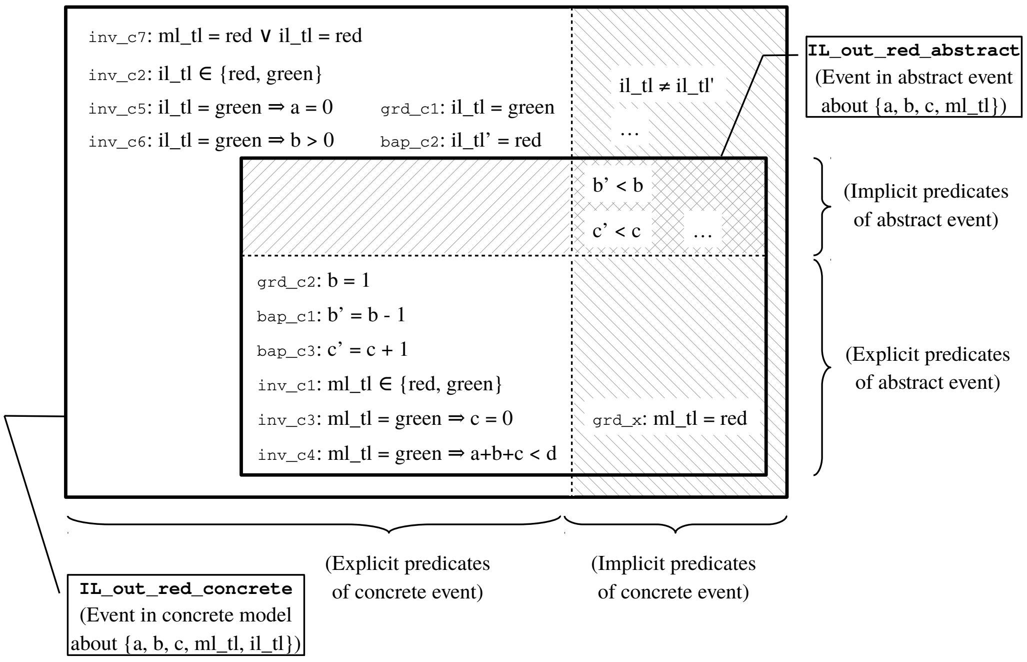

inv_c7: ml_tl = red ∨ il_tl = red
inv_c2: il_tl ∈ {red, green}
inv_c5: il_tl = green ⇒ a = 0
inv_c6: il_tl = green ⇒ b > 0
grd_c1: il_tl = green
bap_c2: il_tl' = red
il_tl ≠ il_tl'
…
IL_out_red_abstract
(Event in abstract event about {a, b, c, ml_tl})
b' < b
c' < c
…
(Implicit predicates of abstract event)
grd_c2: b = 1
bap_c1: b' = b - 1
bap_c3: c' = c + 1
inv_c1: ml_tl ∈ {red, green}
inv_c3: ml_tl = green ⇒ c = 0
inv_c4: ml_tl = green ⇒ a+b+c < d
grd_x: ml_tl = red
(Explicit predicates of abstract event)
(Explicit predicates of concrete event)
(Implicit predicates of concrete event)
IL_out_red_concrete
(Event in concrete model about {a, b, c, ml_tl, il_tl})

Figure 3.3: Explicit predicates and implicit predicates of abstract machine and concrete machine. Implicit predicates of models are placed in areas with hatching.

Not only predicates but also variables may disappear from specification through a refinement. The refinement mechanism also enables modelers to add gluing invariants with concrete variables and convert expressions in an abstract model into a concrete form that uses concrete variables. For example, the first refinement in an example described in Section 2.1.3 converts a variable and predicates about cars outside the mainland into those about cars on the bridge and cars on the island. A gluing invariant in this example is "#cars outside the mainland = #cars on the island + #cars on the bridge." Thus, abstract variables disappear and are replaced with concrete variables through such a refinement. This style of refinement is called *vertical refinement*, and refinement without disappearance of variables is called *horizontal refinement*.

However, even with vertical refinement, predicates specified in abstract models are preserved because gluing invariants relate state spaces of abstract models and concrete models.

## 3.3 Problem and Approach

### 3.3.1 General Problem and Approach

During construction or modification of Event-B models, modelers have an idea (maybe incomplete) of the most concrete model. The idea consists of explicit elements, explicit properties, implicit elements, and implicit properties of the model.

Before constructing Event-B models, developers usually have the whole idea of the most concrete model as informal documents. During construction of such informal documents, multiple abstraction layers are usually not considered. That is, the idea of the most concrete model is constructed on a single abstraction layer. Although the idea usually describes all parts of the target system, consistency of the idea is not yet verified. On the other hand, after constructing Event-B models, the idea is the most concrete model itself.

In the beginning of modeling, modelers construct abstract models that describe the most concrete model with only abstract vocabulary. After constructing models, modelers may want to modify abstract models for maintenance.

Because of the functionality of refinement, how to determine what elements and properties are included in each refinement step is important. However, modelers currently rely on complex and unsystematic artisanship to determine this. We found this process includes a part that can be treated in a rational way and propose a systematic method to treat it.

We aim to handle the following general problem.

**General problem** For given information of properties and elements of the most concrete model, including implied ones, find an appropriate allocation of them to several refinement steps.

In order to tackle the problem, we pay attention to the *dependencies* of elements and properties.

The relationships of dependency are as follows:

**Elements occurrence** (Relationship between a property and elements.) In order to specify a property, elements that appear in the property need to be specified in the model.

**Invariants for typing** (Relationship between an element and a property.) In order to introduce an element $v$ into a model $M$, a typing invariant for $v$ should also be introduced in $M$.

**Replacement with gluing invariants** (Relationship between elements and properties.) Let $I_V$ be the set of given invariants that are about a set of variables $V$. In order to replace variables $V$ in an abstract machine $M_{\mathrm{A}}$ through refinement by a concrete machine $M_{\mathrm{C}}$, all invariants in $I_V$ that are not specified in $M_{\mathrm{A}}$ should be specified in $M_{\mathrm{C}}$.

**Static elements for typing** (Relationship between elements.) For typing of elements, some elements require other elements to be included in the model.

**Abstract before concrete** (Relationship between elements.) If an element $e$ is considered to be an abstract version of another element $f$, then $e$ should not be introduced later than $f$.

**Conceptually similar properties** (Relationship between a property and properties.) If elements that are required to introduce property $p$ in a model $M$ allow specifying other properties $p'$, then $p'$ should also be introduced to $M$.

In actual development, collecting appropriate information on dependencies, abstract elements, and abstract properties before constructing models is not straightforward. We discuss the feasibility of this process in Chapter 7.

Defining this generic problem and approach enables us to deal with multiple problems that developers face during actual development.

### 3.3.2 Refactoring

The refinement structure of Event-B machines reflects aspects that are considered in each refinement step. Therefore, refinement structure is important for understandability and maintainability of the machines. In particular, decomposing one refinement step into some steps is useful, because modelers tend to construct a large refinement step in actual development. In addition, although refinement mechanism of Event-B is effective for reuse of existing machines, in actual development, existing machines are usually mixtures of reusable parts and non-reusable parts.

Therefore, there is a strong demand for refactoring refinements of existing models. Thus, by analyzing consistent models (i.e., POs of them have already been discharged), we aim to construct new models that focus on particular sets of variables and are consistent with given models.

This goal is achieved by extracting predicates in given models that are written with particular variables. A straightforward approach is syntactic slicing, namely finding predicates that are written with the variables. However, this approach does not find implicit predicates, which are necessary for consistency of the new model in most cases. Thus, for the sake of consistency, we need a method to find appropriate implicit predicates from given models.

To this end, we analyze the proofs of POs of given models and find implicit predicates of given models that are necessary for consistency of new models.

Thus, our refactoring method can be illustrated as Figure 3.4. In this example, our refactoring method analyzes predicates of original model (`IL_out_red_concrete`) and finds predicates that become explicit predicates of new model (`IL_out_red_abstract`). It finds not only explicit predicates of original model that can be written with variables $\{a, b, c, ml\_tl\}$ such as `grd_c2` and `bap_c1` but also implicit predicates of original model that should be explicitly written in new model such as `grd_x`.

We describe this approach in Chapter 4.

There has been a tool support for refactoring of Event-B machines [32]. Although the functionalities of the tool, such as global renaming of identifiers, are useful, this tool does not support our goal, namely consistent refactoring of refinement structure. Supports for helping developers to understand Event-B models — such as model checking tool [57], animator tool [64], and a method for animating refinement [42] — have been proposed. They provide an appropriate view for understanding without changing models, while our refactoring approach aims at changing the refinement structure of a model to improve its inherent understandability.

Tools and methods effective for improving reusability of Event-B have also been proposed. Generic instantiation [73] enables Event-B modelers to use abstract data types to enhance reusability of *static* parts of Event-B models. Methods for machine decomposition [5, 18] have been used to cut a machine into sub-machines of *the same abstraction level* to reduce complexity. Design pattern management mechanism [46] provides a functionality that applies predicate-level *patterns* to automate certain forms of refinement. In contrast, our method aims to enhance reusability of *dynamic* parts of Event-B models by *extracting actual predicates* of specific aspects that range *multiple abstraction layers*. Thus, we consider the tools and methods above as complementary to ours.

### 3.3.3 Planning

Developers need to have a plan of refinement structure before they construct Event-B models. However, by following an arbitrary plan, they tend to construct invalid models or models that do not effectively use the refinement mechanism. In such cases, developers suffer from rework of constructing and verifying models. In addition, the space of reasonable refinement plans is usually too large for

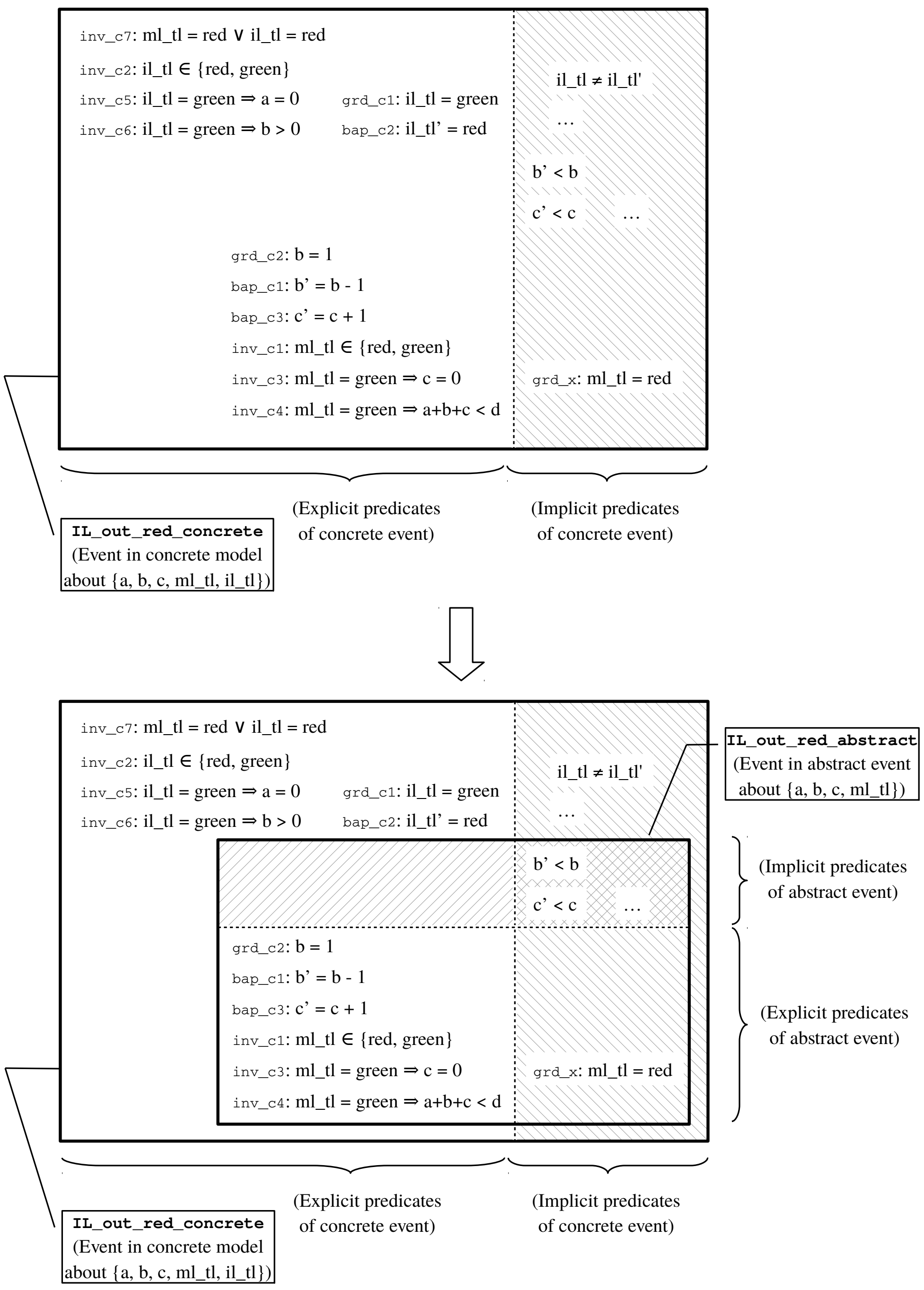

inv_c7: ml_tl = red ∨ il_tl = red
inv_c2: il_tl ∈ {red, green}
inv_c5: il_tl = green ⇒ a = 0
grd_c1: il_tl = green
inv_c6: il_tl = green ⇒ b > 0
bap_c2: il_tl' = red
il_tl ≠ il_tl'
…
b' < b
c' < c
…
grd_c2: b = 1
bap_c1: b' = b - 1
bap_c3: c' = c + 1
inv_c1: ml_tl ∈ {red, green}
inv_c3: ml_tl = green ⇒ c = 0
inv_c4: ml_tl = green ⇒ a+b+c < d
grd_x: ml_tl = red
IL_out_red_concrete
(Event in concrete model about {a, b, c, ml_tl, il_tl})
(Explicit predicates of concrete event)
(Implicit predicates of concrete event)
IL_out_red_abstract
(Event in abstract event about {a, b, c, ml_tl})
(Implicit predicates of abstract event)
(Explicit predicates of abstract event)

Figure 3.4: Refactoring approach

developers to manually grasp and explore.

Therefore, it is important to help developers understand the possibilities of refinement plans so that they can examine the complexity of each refinement step and the consistency of models constructed by following the plans.

However, there has not been any criterion of desirable refinement plans. Moreover, dependencies of variables and predicates should be handled to find reasonable plans.

To address these problems, we provide rationales of refinement structure to find structures that facilitate effective use of the refinement mechanism. In addition, we also provide a method that searches plans by considering the rationales and analyzing dependencies of variables and predicates.

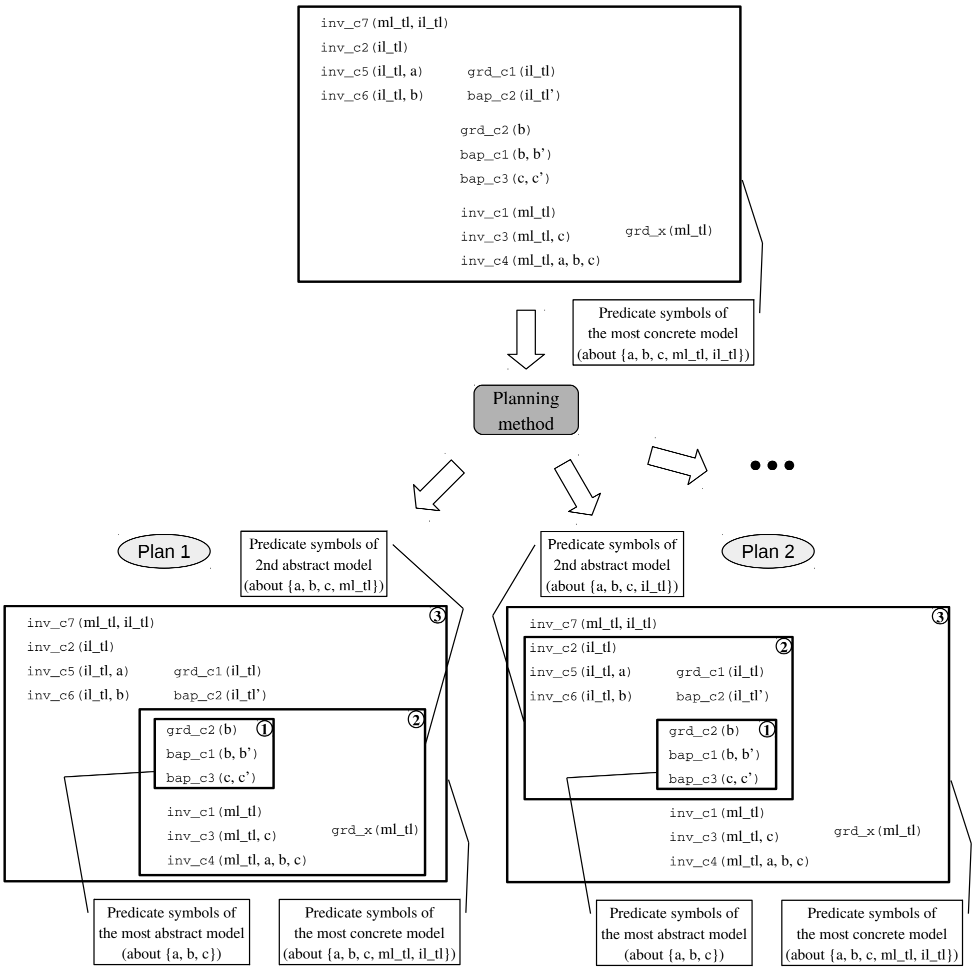
Before modeling, the idea of the most concrete machine is usually described as informal documents instead of formal expression of predicates. Although predicates are not given, we assume that developers can identify what variables are necessary to specify the predicates in this phase. Therefore, we assume that a list of predicate symbols and their variables are provided. It should be noted that we need to not only consider predicate symbols of the most concrete model but also those of abstract models because we need to plan the introduction of predicates through multiple steps of refinement. Thus, the planning method also requires some symbols of predicates that are implicit in the concrete model.

As we explained in Section 3.2, predicates in abstract models (often specified with abstract variables that are replaced through vertical refinements) may become implicit in specification of concrete models, but they are inherited by concrete models. Whether they are implicit is a problem about expressions of specification. Because the planning approach does not handle expressions of specifications, we do not consider the hiding of variables and predicates through refinement in this approach. In other words, we focus on the accumulation of variables and predicate symbols through refinement.

Our planning method can be illustrated as Figure 3.5. In this example, a list of predicate symbols and their variables, such as `inv_c7`$(ml\_tl, il\_tl)$, are given. The list includes predicate symbols of abstract predicates, such as `grd_x`$(ml\_tl)$. Our planning method generates multiple plans of refinement structure. Plan 1 in Figure 3.5 shows a plan that corresponds to refinement structure $[\{a, b, c\}, \{a, b, c, ml\_tl\}, \{a, b, c, ml\_tl, il\_tl\}]$. Plan 2 corresponds to another refinement strategy: $[\{a, b, c\}, \{a, b, c, il\_tl\}, \{a, b, c, il\_tl, ml\_tl\}]$. The user can select and examine a plan from generated candidates.

We describe this approach in Chapter 5.

Various methods that help refinement have been proposed. Studies of [70, 55, 17] generate Event-B models from informal models in popular forms, such as UML diagrams, SysML models, or BPMN models. However, such approaches are not suitable for refinement planning because multiple abstraction layers are not considered in such informal models. The ProR approach [53] is useful to reason about a relationship between requirements and models because it provides traceability between informal requirements document and Event-B models, but it does not support handling refinement structure. In contrast, our planning approach aims at explicitly handling refinement structure. The design pattern management mechanism [46] also guides refinement because the pattern applied to a machine is instantiated as another machine that refines the existing one. However, it does not support our goal, namely planning the whole refinement structure before the start of modeling. Therefore, it is important to provide a method that aims at our goal.



inv_c7 (ml_tl, il_tl)
inv_c2 (il_tl)
inv_c5 (il_tl, a)
inv_c6 (il_tl, b)
grd_c1 (il_tl)
bap_c2 (il_tl')
grd_c2 (b)
bap_c1 (b, b')
bap_c3 (c, c')
inv_c1 (ml_tl)
inv_c3 (ml_tl, c)
inv_c4 (ml_tl, a, b, c)
grd_x (ml_tl)
Predicate symbols of the most concrete model (about {a, b, c, ml_tl, il_tl})
Planning method
Plan 1
Plan 2
Predicate symbols of 2nd abstract model (about {a, b, c, ml_tl})
Predicate symbols of 2nd abstract model (about {a, b, c, il_tl})
Predicate symbols of the most abstract model (about {a, b, c})
Predicate symbols of the most concrete model (about {a, b, c, ml_tl, il_tl})
1
2
3

Figure 3.5: Planning approach

# Chapter 4

# Refinement Refactoring

## 4.1 Approach

Our view of the refactoring problem and an approach to solve it are as follows.

We treat variables of machines as elements. Properties we treat are written in machines as invariants, guards, and before-after predicates of actions.

Actual machines that form a refinement chain $[M_n, M_{n+1}, \ldots, M_m]$ are given. In other words, elements and properties that are explicitly specified in $M_n, M_{n+1}, \ldots, M_m$ are given.

Our goal is to restructure the refinement structure of given machines. Thus, when a sequence of sets of elements $E'_s, E'_{s+1}, \ldots, E'_t$ such that

- $E'_s$ is equal to the set of elements of $M_n$
- $E'_t$ is equal to the set of elements of $M_m$
- Other conditions described in Section 4.5

is given, our method constructs machines $M'_s, M'_{s+1}, \ldots, M'_t$ such that

- variables of $M'_i$ are equal to $E'_i$
- $M'_{i+1}$ refines $M'_i$

by finding elements and actual expressions of properties for every $M'_i$.

For the sake of simplicity, in this thesis we omit parameters of each event, with which we can construct a similar argument.

The relationships of dependencies we pay attention to are as follows:

**Elements occurrence** (Relationship between a property and elements.) In order to specify a property, elements that appear in the property need to be specified in the model.

**Invariants for typing** (Relationship between an element and a property.) In order to introduce an element $e$ into a model $M$, a typing invariant for $e$ should also be introduced in $M$.

**Replacement with gluing invariants** (Relationship between elements and properties.) Let $I_V$ be the set of given invariants that are about a set of variables $V$. In order to replace variables in $V$ in an abstract machine $M_{\mathrm{A}}$ through refinement by a concrete machine $M_{\mathrm{C}}$, all invariants in $I_V$ that are not specified in $M_{\mathrm{A}}$ should be specified in $M_{\mathrm{C}}$. We elaborate on this relationship in Section 4.4.1.

## 4.2 Overview

We assume that we have given consistent (proved) machines $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ such that $M_{\mathrm{C}}$ refines $M_{\mathrm{A}}$. The goal of our decomposition method is to construct an intermediate machine $M_{\mathrm{B}}$ such that $M_{\mathrm{C}}$ refines $M_{\mathrm{B}}$ and $M_{\mathrm{B}}$ refines $M_{\mathrm{A}}$ by using as much of the original specifications as possible.

For this purpose, users give a slicing criterion as a set of variables $V_{\mathrm{B0}}$, which actually may be given by selecting variables in $V_{\mathrm{C}}$.

The first step is to use this criterion for syntactic slicing from $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ to construct the initial base $M_{\mathrm{B0}}$. The actual criterion for slicing $V_{\mathrm{B}}$ is extended from $V_{\mathrm{B0}}$ because of consistency constraints. In general, the result of this first step $M_{\mathrm{B0}}$ may have POs that are not provable.

Thus, the second step helps developers to manually add additional predicates to $M_{\mathrm{B0}}$ to make a consistent intermediate machine $M_{\mathrm{B}}$. By handling replacement of variables through refinement and proof obligations, our decomposition method deals with both horizontal refinement and vertical refinement.

Combined with merging of refinements, the decomposition method is extended as a restructuring method (Section 4.5).

## 4.3 Symbols, Definitions, and Assumptions

In this chapter, we use the following symbols:

### 4.3.1 Symbols about Given Input Information

$M_{\mathrm{A}}, M_{\mathrm{C}}$ Abstract machine and concrete machine. We assume that both are proved and that $M_{\mathrm{C}}$ refines $M_{\mathrm{A}}$.

$V_{\mathrm{A}}, V_{\mathrm{C}}$ Variables declared in $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$, respectively.

$I_{\mathrm{A}}$ The set of all invariants of $M_{\mathrm{A}}$ such that $\mathrm{var}(I_{\mathrm{A}}) \subseteq V_{\mathrm{A}}$, where $\mathrm{var}(P)$ represents the variables that occur in set of predicates $P$.

$I_{\mathrm{C}}$ Invariants specified in $M_{\mathrm{C}}$. $\mathrm{var}(I_{\mathrm{C}}) \subseteq V_{\mathrm{A}} \cup V_{\mathrm{C}}$.

$G_{\mathrm{A}}, G_{\mathrm{C}}$ Guards specified in events of $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$, respectively. Because guards can only use variables in current machine, $\mathrm{var}(G_{\mathrm{A}}) \subseteq V_{\mathrm{A}}$ and $\mathrm{var}(G_{\mathrm{C}}) \subseteq V_{\mathrm{C}}$.

$V_{\mathrm{B0}}$ Slicing criterion. It is a subset of $V_{\mathrm{C}}$.

### 4.3.2 Symbols about Output of Decomposition Method

$M_{\mathrm{B}}$ Intermediate machine.

$V_{\mathrm{B}}$ Variables declared in $M_{\mathrm{B}}$. It is found by adding elements to $V_{\mathrm{B0}}$.

### 4.3.3 Definitions of Terms in This Chapter

**Gluing invariant** An invariant $i$ written in $M_{\mathrm{C}}$ such that $\mathrm{var}(\{i\}) \cap (V_{\mathrm{A}} \setminus V_{\mathrm{C}}) \neq \emptyset$ and $\mathrm{var}(\{i\}) \cap (V_{\mathrm{C}} \setminus V_{\mathrm{A}}) \neq \emptyset$.

**Replacement of variables by gluing invariants** Through a vertical refinement, variables $V$ of the abstract machine may disappear. In such a case, a developer needs to provide the concrete machine with gluing invariants $I$, and $V \subseteq \mathrm{var}(I)$ should hold; otherwise, it is impossible to verify the

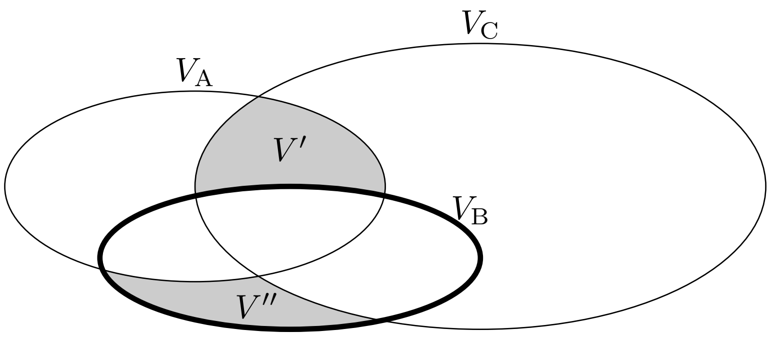


Figure 4.1: Arbitrary variables of $M_B$ (invalid case)

consistency between two machines because events of abstract machine are written with $V$, but those of concrete machine are not. We refer to such a situation by stating that $V$ *has been replaced with* $I$.

### 4.3.4 Assumption

**Assumption 4.3.1** *Variables* $V_C \backslash V_A$ *occur in all invariants of concrete machine. In other words, there are no invariants of concrete machine that are written only with variables of* $V_A$*.*

We consider this assumption as reasonable since invariants that are written only with variables of $V_A$ are properties that can be specified in the abstract machine.

## 4.4 Decomposition

### 4.4.1 Step 1 of Decomposing Refinement: Slicing

#### Finding Additional Variables

If $M_C$ refines $M_A$, then $M_C$ inherits variables $V_A \cap V_C$ from $M_A$. The remaining variables of $M_A$ — that is, $V_A \setminus V_C$ — are replaced with some of the variables in $V_C \setminus V_A$, as described in Section 2.4.

As shown in Figure 4.1, if $V_A \cap V_C \nsubseteq V_B$, then the variables in $(V_A \cap V_C) \setminus V_B$ ($V'$ in Figure 4.1), which is a subset of $V_A \setminus V_B$, are declared in $M_C$.

The variables in $V_A \setminus V_B$ are, however, replaced with other variables in $M_B$. This means that some variables disappear through refinement of "$M_B$ refines $M_A$" and then revive through refinement of "$M_C$ refines $M_B$". Such refinements are invalid. Therefore, $V_B$ must satisfy $V_A \cap V_C \subseteq V_B$.

In addition, to take advantage of predicates in existing machines to construct $M_B$, $V_B$ should satisfy $V_B \subseteq V_A \cup V_C$; otherwise, a user needs to design new variables $(V_B \setminus (V_A \cup V_C))(V''$ in Figure 4.1) and predicates of them. Thus, $V_B$ should be as depicted in Figure 4.2.

Hereinafter, we will use the symbols $\tilde{V_A}$, $V_{AB}$, $V_{ABC}$, $V_{BC}$, and $\tilde{V_C}$ to represent $V_A \setminus V_B$, $V_B \setminus V_C$, $V_A \cap V_C$, $V_B \setminus V_A$, and $V_C \setminus V_B$, respectively.

Our refactoring method assumes that $V_{B0} = V_{ABC} \cup V_{BC}$, which is a subset of $V_C$, is given as an input. This is because it is easy for a user to select the criterion from $V_C$, without considering which variables in $V_A$ must be replaced. To construct $M_B$, the remaining variables $V_B \setminus V_{B0} (= V_{AB})$ need to be identified. The remainder of this section describes a heuristic for automatically finding $V_{AB}$.

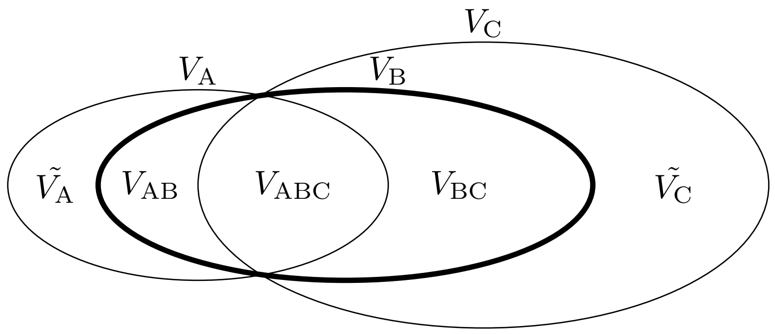


Figure 4.2: Valid variables of $M_{\text{B}}$

To replace abstract variables with concrete ones in a concrete machine, a user needs to provide gluing invariants about the relationships between the two sets of variables. In the case of constructing $M_{\text{B}}$ as a machine that refines $M_{\text{A}}$, the set of newly introduced variables $V_{\text{BC}}$ is a subset of $V_{\text{C}} \setminus V_{\text{A}}$. Therefore, some of the gluing invariants in $M_{\text{C}}$ may not be specified in $M_{\text{B}}$. $M_{\text{B}}$'s gluing invariants can describe the relationship between $\tilde{V_{\text{A}}}$ and $V_{\text{BC}}$, but cannot describe the relationship between $V_{\text{AB}}$ and $V_{\text{BC}}$. Hence, $V_{\text{AB}}$ can be obtained as

$$V_{\text{AB}} = \{v \in V_{\text{A}} \setminus V_{\text{C}} \mid \exists i \in \text{ginv}(M_{\text{C}}) \,.\, v \in (\text{var}(i) \cap V_{\text{A}}) \wedge \text{var}(i) \cap \tilde{V_{\text{C}}} \neq \emptyset\}, \tag{4.1}$$

where ginv($M$) represents the gluing invariants in a machine $M$.

In other words, $\tilde{V_{\text{A}}} = ((V_{\text{A}} \setminus V_{\text{C}}) \setminus V_{\text{AB}})$ can be obtained as

$$\tilde{V_{\text{A}}} = \{v \in V_{\text{A}} \setminus V_{\text{C}} \mid \forall i \in \text{ginv}(M_{\text{C}}) \,.\, v \in (\text{var}(i) \cap V_{\text{A}}) \Rightarrow \text{var}(i) \cap \tilde{V_{\text{C}}} = \emptyset\}. \tag{4.2}$$

For example, let us assume that $a$ and $e$ are selected to be specified in $M_{\text{B}}$ ($V_{\text{B0}} = V_{\text{B}} \cap V_{\text{C}} = \{a, e\}$). In $M_{\text{C}}$, a gluing invariant `gluinv1:` $b = c + d$ describes replacement of $b$ (of $M_{\text{A}}$) with $c$ and $d$ (of $M_{\text{C}}$). By contrast, in $M_{\text{B}}$, `gluinv1` cannot describe replacement of $b$ since neither $c$ nor $d$ is selected to be specified in $M_{\text{B}}$. Therefore, $V_{\text{AB}} = \{b\}$; namely, $M_{\text{B}}$ should specify $b$ in addition to $a$ and $e$.

## Finding Certain Specifications through Slicing

For a predicate $p$ and a set of variables $V$, we say $p$ is *expressible* by $V$ if and only if var($p$) $\subseteq V$. The procedure described in this section finds predicates that are expressible by $V_{\text{B}}$.

Hereinafter, we use the following symbols:

$I_{\text{AB}}$ Members of $I_{\text{A}}$ that are expressible by $V_{\text{B}}$ (i.e., variables of $\tilde{V_{\text{A}}}$ do not appear in $I_{\text{AB}}$). Thus, $I_{\text{AB}} = \{i_A \in I_{\text{A}} \mid \text{var}(\{i_A\}) \subseteq V_{\text{AB}} \cup V_{\text{ABC}}\}$.

$\tilde{I_{\text{A}}}$ $I_{\text{A}} \setminus I_{\text{AB}}$. Variables of $\tilde{V_{\text{A}}}$ appear in all members of $\tilde{I_{\text{A}}}$. Thus, $\tilde{I_{\text{A}}} = \{i_A \in I_{\text{A}} \mid \text{var}(\{i_A\}) \cap \tilde{V_{\text{A}}} \neq \emptyset\}$.

$I_{\text{BC}}$ Members of $I_{\text{C}}$ that are expressible by $V_{\text{B}}$ (i.e., variables of $\tilde{V_{\text{C}}}$ do not appear in $I_{\text{BC}}$). Thus, $I_{\text{BC}} = \{i_C \in I_{\text{C}} \mid \text{var}(\{i_C\}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}}\}$.

$\tilde{I_{\text{C}}}$ $I_{\text{C}} \setminus I_{\text{BC}}$. Variables of $\tilde{V_{\text{C}}}$ appear in all members of $\tilde{I_{\text{C}}}$. Thus, $\tilde{I_{\text{C}}} = \{i_C \in I_{\text{C}} \mid \text{var}(\{i_C\}) \cap \tilde{V_{\text{C}}} \neq \emptyset\}$.

```
variables : a, b, e
```

```
typ_b : {a, b, e} ⊂ ℕ
inv_c1 : mod2(a + e) = 0 ⇒ a < 1
```

```
Event evt_b
  refines evt_a
  begin
    act_c1 : a :| a′ = a + 1
    act_a2 : b :| b′ = b + 2
  end
```

```
Event initialisation
  begin
    init_c1 : a :| a′ = 0
    init_a2 : b :| b′ = 0
    init_c5 : e :| e′ = 1
  end
```

Figure 4.3: mb0: machine obtained with slicing from ma and mc

| | |
|---|---|
| $I_{\mathrm{B}}$ | $I_{\mathrm{AB}} \cup I_{\mathrm{BC}}$. Invariants of the given machines that are expressible by $V_{\mathrm{B}}$. It becomes invariants of $M_{\mathrm{B}}$. |

$G_{\mathrm{AB}}$, $\tilde{G_{\mathrm{A}}}$, $G_{\mathrm{BC}}$, and $\tilde{G_{\mathrm{C}}}$ denote sliced guards in the same manner.

Moreover, since guards of the intermediate machine are augmented in Step 2 (Section 4.4.2), we denote the base of guards of intermediate machine as follows:

| | |
|---|---|
| $G_{\mathrm{B0}}$ | $G_{\mathrm{AB}} \cup G_{\mathrm{BC}}$. Final set of guards is constructed by augmenting $G_{\mathrm{B0}}$. |

Predicates in $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ that are expressible by $V_{\mathrm{B}}$ certainly express the properties of $V_{\mathrm{B}}$, which should be consistent with $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$. Therefore, in this step, invariants, guards, and BAPs in $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ that are expressible by $V_{\mathrm{B}}$ are specified in $M_{\mathrm{B}}$. [1] For example, mb0 (Figure 4.3) is constructed by collecting predicates that are expressible by $V_{\mathrm{B}} = \{a, b, e\}$ from ma and mc.

Note that an action mc/evt_c/act_c5, which assigns a value to $e (\in V_{\mathrm{B}})$, is not specified in mb0 because $f (\in \tilde{V_{\mathrm{C}}})$ occurs in its BAP.

The algorithm for slicing invariants is shown in Algorithm 1.

**Algorithm 1** Slicing invariants

```
1: $I_{\mathrm{B0}} \leftarrow \emptyset$
2: for all $i_{\mathrm{A}} \in I_{\mathrm{A}}$ do
3:     if $\mathrm{var}(i_{\mathrm{A}}) \subseteq (V_{\mathrm{A}} \cup V_{\mathrm{B}})$ then
4:         if $\mathrm{var}(i_{\mathrm{A}}) \cap V_{\mathrm{B}} \neq \emptyset$ then
5:             $I_{\mathrm{B0}} \leftarrow I_{\mathrm{B0}} \cup \{i_{\mathrm{A}}\}$
6:         end if
7:     end if
8: end for
9: for all $i_{\mathrm{C}} \in I_{\mathrm{C}}$ do
10:     if $\mathrm{var}(i_{\mathrm{C}}) \subseteq (V_{\mathrm{A}} \cup V_{\mathrm{B}})$ then
11:         if $\mathrm{var}(i_{\mathrm{C}}) \cap V_{\mathrm{B}} \neq \emptyset$ then
12:             $I_{\mathrm{B0}} \leftarrow I_{\mathrm{B0}} \cup \{i_{\mathrm{C}}\}$
13:         end if
14:     end if
15: end for
```

The procedure tries to find some of the invariants in the abstract machine (Lines 2–7) and the concrete machine (Lines 9–15). As the invariants of intermediate machine should be specified with variables of $V_{\mathrm{A}} \cup V_{\mathrm{B}}$, the procedure collects

[1]BAPs that are expressible by $V_{\mathrm{B}} \cup V'_{\mathrm{B}}$ are also specified, where $V'_{\mathrm{B}}$ represents the set of after-state variables of $V_{\mathrm{B}}$.

invariants that can be expressible with $V_\mathrm{A} \cup V_\mathrm{B}$ (Lines 3 and 10). In addition, because invariants of $M_\mathrm{B}$ should describe properties of $V_\mathrm{B}$, the procedure collects invariants that are related to $V_\mathrm{B}$ (Lines 4 and 11). The slicing procedure collects typing invariants of all variables in $V_\mathrm{B}$.

The algorithm for slicing events is shown in Algorithm 2.

**Algorithm 2** Slicing events

```
1: $E_{\mathrm{B0}} \leftarrow \emptyset$
2: for all $e_\mathrm{C} \in E_\mathrm{C}$ do
3:     $e_{\mathrm{B0}} \leftarrow$ (empty event)
4:     $E_{\mathrm{B0}} \leftarrow E_{\mathrm{B0}} \cup \{e_{\mathrm{B0}}\}$
5:     refines($e_{\mathrm{B0}}$) $\leftarrow$ refines($e_\mathrm{C}$)
6:     for all $g_{e_\mathrm{C}} \in$ guards($e_\mathrm{C}$) do
7:         if var($g_{e_\mathrm{C}}$) $\subseteq V_\mathrm{B}$ then
8:             guards($e_{\mathrm{B0}}$) $\leftarrow$ guards($e_{\mathrm{B0}}$) $\cup \{g_{e_\mathrm{C}}\}$
9:         end if
10:    end for
11:    for all $a_{e_\mathrm{C}} \in$ actions($a_{e_\mathrm{C}}$) do
12:        if var($a_{e_\mathrm{C}}$) $\subseteq V_\mathrm{B} \cup V'_\mathrm{B}$ then
13:            actions($e_{\mathrm{B0}}$) $\leftarrow$ actions($e_{\mathrm{B0}}$) $\cup \{a_{e_\mathrm{C}}\}$
14:        end if
15:    end for
16:    for all $e_\mathrm{A} \in$ refines($e_{\mathrm{B0}}$) do
17:        for all $g_{e_\mathrm{A}} \in$ guards($e_\mathrm{A}$) do
18:            if var($g_{e_\mathrm{A}}$) $\subseteq V_\mathrm{B}$ then
19:                guards($e_{\mathrm{B0}}$) $\leftarrow$ guards($e_{\mathrm{B0}}$) $\cup \{g_{e_\mathrm{A}}\}$
20:            end if
21:        end for
22:        for all $a_{e_\mathrm{A}} \in$ actions($a_{e_\mathrm{A}}$) do
23:            if var($a_{e_\mathrm{A}}$) $\subseteq V_\mathrm{B} \cup V'_\mathrm{B}$ then
24:                actions($e_{\mathrm{B0}}$) $\leftarrow$ actions($e_{\mathrm{B0}}$) $\cup \{a_{e_\mathrm{A}}\}$
25:            end if
26:        end for
27:    end for
28:    $unique \leftarrow true$
29:    for all $f_{\mathrm{B0}} \in (E_{\mathrm{B0}} \setminus \{e_{\mathrm{B0}}\})$ do
30:        if $f_{\mathrm{B0}} = e_{\mathrm{B0}}$ then
31:            refines($e_\mathrm{C}$) $\leftarrow f_{\mathrm{B0}}$
32:            $unique \leftarrow false$
33:        end if
34:    end for
35:    if $unique = false \vee$ e$_{\mathrm{B0}}$ = (empty event) then
36:        $E_{\mathrm{B0}} \leftarrow E_{\mathrm{B0}} \setminus e_{\mathrm{B0}}$
37:    else
38:        refines($e_\mathrm{C}$) $\leftarrow e_{\mathrm{B0}}$
39:    end if
40: end for
```

The procedure constructs an empty event $e_{\mathrm{B0}}$ of $M_{\mathrm{B0}}$ for each event $e_\mathrm{C}$ of $M_\mathrm{C}$ first (Lines 2–3). $e_{\mathrm{B0}}$ refines events that are refined by $e_\mathrm{C}$ (Line 5). Then, because events of $e_{\mathrm{B0}}$ should be specified with $V_\mathrm{B}$, guards of $e_\mathrm{C}$ that are expressible with $V_\mathrm{B}$ and actions of $e_\mathrm{C}$ that are expressible with $V_\mathrm{B} \cup V'_\mathrm{B}$ are collected as specification of $e_{\mathrm{B0}}$ (Lines 6–10 and 11–15). The same procedure is also applied to every $e_{\mathrm{A}}$ that is refined by $e_{\mathrm{B0}}$ (Lines 17–21 and 22–26). This procedure may construct an empty event as $e_{\mathrm{B0}}$ or multiple events with the same specification. Therefore, the procedure checks whether there is another event that has the same specification as $e_{\mathrm{B0}}$'s (Lines 28–34) and eliminates such an event (Line 36). If the $e_{\mathrm{B0}}$ is a unique and non-empty event, it is added as an event of $M_{\mathrm{B0}}$ (Line 38).

As a result of slicing of events (Algorithm 2), the refinement structure of events becomes as shown in Figure 4.4. Basically, one event of the intermediate machine is constructed for an event of the concrete machine (Figure 4.4a). Constructed intermediate events refine abstract events that are refined by corresponding original concrete events (Figures 4.4b and 4.4c). In the case of slicing constructed multiple identical intermediate events, duplicated events are eliminated (Line 36 of Algorithm 2, Figure 4.4d).

We implemented these steps as a plugin tool of Event-B's IDE, named SliceAndMerge. [2] Users of the tool can select $V_{\mathrm{B0}}$ with checkboxes and obtain a sliced machine $M_{\mathrm{B0}}$ (Figure 4.5). The tool also supports analysis of dependencies between invariants and variables, and merging of refinements (described in Section 4.5).

### 4.4.2 Step 2 of Decomposing Refinement: Complementing

In this section, we describe complementing of the base of intermediate machine $M_{\mathrm{B0}}$. We explain that the machine $M_{\mathrm{B0}}$ may not be consistent or may not be consistent with given machines (i.e., $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$). Next, we show properties of predicates that should be added to $M_{\mathrm{B0}}$ (*complementary predicates*). We then describe how to manually find complementary predicates and show possible heuristics that make the process automatic in specific situations.

#### Possible Lack of Consistency in $M_{\mathrm{B0}}$

Although all POs of $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ are discharged, $M_{\mathrm{B0}}$, which is constructed from fragments of the machines, is not ensured to be consistent. For example, some invariants in $M_{\mathrm{B0}}$ (from $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$) may not be preserved by events in $M_{\mathrm{B0}}$ because the specification of $M_{\mathrm{B0}}$'s events may be only part of the specification of $M_{\mathrm{A}}$'s and $M_{\mathrm{C}}$'s events.

For instance, the PO shown in Figure 2.6 (`mc/evt_c/inv_c1`/INV), which has a succedent specified with the after-states of $a$ and $e$, is provable because predicates, including `grd_c2` and `act_c5`, are in the antecedent.

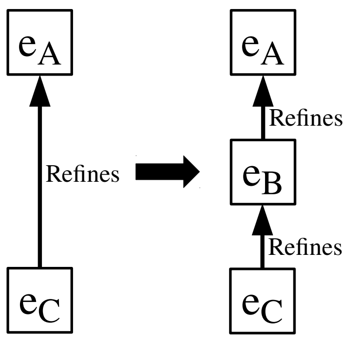
Although the preservation of the same invariant `inv_c1` by the event `evt_b` in `mb0` (`mb0/evt_b/inv_c1`/INV) should also hold, this is not provable because predicates `grd_c2` and `act_c5`, which are essential for proving that `inv_c1` is preserved, are not included in the antecedent (Figure 4.6) as they are predicates about variable $f$ ($\in \tilde{V_{\mathrm{C}}}$).

#### Complementary Predicates for Consistency

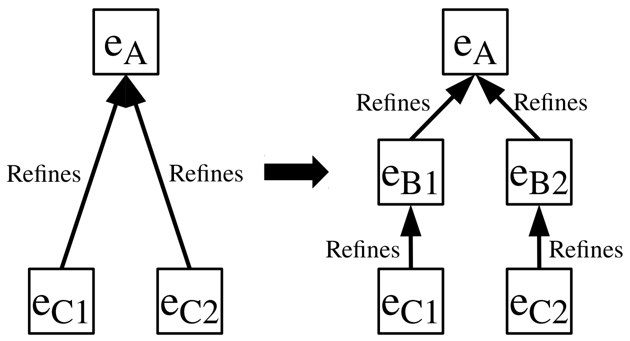
Since $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$ are consistent, they have predicates that are essential for the consistency.

When such predicates are expressible by $V_{\mathrm{B}}$, the consistency of $M_{\mathrm{B}}$ can be guaranteed by including such predicates in $M_{\mathrm{B}}$. Obviously in simple cases, this can be realized by slicing (Section 4.4.1).

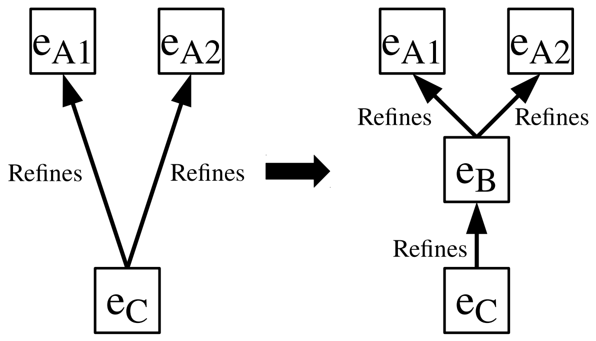
[2] Available at `http://tkoba.jp/software/slice_and_merge/`



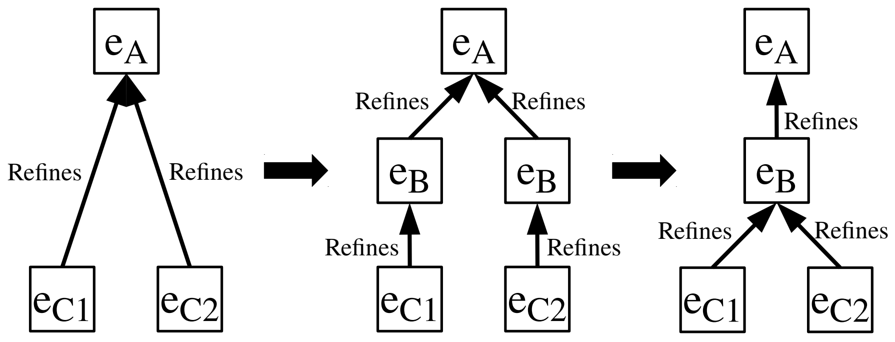
(a) Simple case



(b) Interpolation of multiple events that refine single abstract event

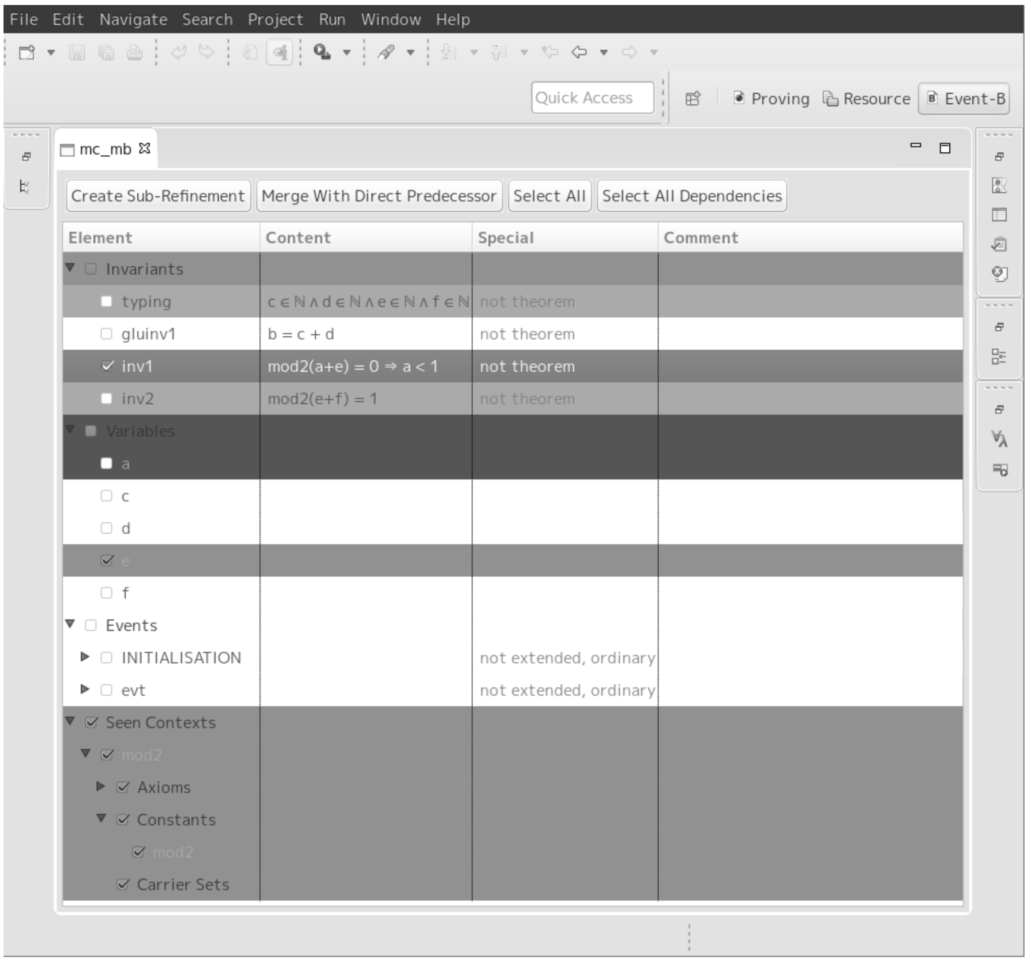


(c) Interpolation of single event that refines multiple abstract events (merging of events through refinement)



(d) Special case of Figure 4.4b: intermediate events become identical

Figure 4.4: Interpolation of refinement structure of events



| Element | Content | Special | Comment |
|---|---|---|---|
| Invariants | | | |
| typing | c ∈ ℕ ∧ d ∈ ℕ ∧ e ∈ ℕ ∧ f ∈ ℕ | not theorem | |
| gluinv1 | b = c + d | not theorem | |
| ✓ inv1 | mod2(a+e) = 0 ⇒ a < 1 | not theorem | |
| inv2 | mod2(e+f) = 1 | not theorem | |
| Variables | | | |
| a | | | |
| c | | | |
| d | | | |
| ✓ e | | | |
| f | | | |
| Events | | | |
| INITIALISATION | | not extended, ordinary | |
| evt | | not extended, ordinary | |
| ✓ Seen Contexts | | | |
| ✓ mod2 | | | |
| ✓ Axioms | | | |
| ✓ Constants | | | |
| ✓ mod2 | | | |
| ✓ Carrier Sets | | | |

Figure 4.5: Screenshot of SLICEANDMERGE

| | |
|---|---|
| BAP of `act_c1` | $a' = a + 1$ |
| ... | ... |
| $\vdash$ | $\vdash$ |
| Modified `inv_c1` | $\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$ |

Figure 4.6: Invariant preservation of `inv_c1` by `evt_b` in `mb0` (unprovable)

However, as described above, sometimes $M_{\mathrm{B0}}$ is inconsistent because $M_{\mathrm{B0}}$, which is obtained by a syntactic predicate-level slicing, sometimes lacks some of these predicates. In such cases, predicates that are essential for discharging POs need to be added to $M_{\mathrm{B0}}$ so that the resulting $M_{\mathrm{B}}$ is consistent. Moreover, such predicates need to be expressible by $V_{\mathrm{B}}$. We call such additional predicates *complementary predicates* (CPs).

Predicates that are essential for the consistency of original machines can often be found from the specifications or the proofs of consistency of $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$, and they can often be "translated" into $V_{\mathrm{B}}$ as CPs. We discuss how often CPs are required and how hard finding them is in Section 8.1.

Finding CPs essentially corresponds to Craig interpolation [26]. Thus, this process is feasible only for specific logics, such as first-order logic, since only such logics have interpolation property. For instance, an extension of first logic with a

Table 4.1: Types of POs newly generated by adding CPs

| Discharged PO | $CP$ is added to | Newly generated PO |
|---|---|---|
| $e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}\ \ (g_{\mathrm{A}i} \in \tilde{G_{\mathrm{A}}})$ | $G_{\mathrm{B}}$ | $e_{\mathrm{C}}/CP/\mathrm{GRD}$ |
| $e_{\mathrm{B}}/t_{\mathrm{A}i}/\mathrm{SIM}\ \ (t_{\mathrm{A}i} \in \tilde{T_{\mathrm{A}}})$ | $T_{\mathrm{B}}$ | $e_{\mathrm{C}}/CP/\mathrm{SIM}$ |
| $e_{\mathrm{B}}/i_{\mathrm{A}i}/\mathrm{INV}\ \ (i_{\mathrm{A}i} \in I_{\mathrm{AB}})$ | $T_{\mathrm{B}}$ | $e_{\mathrm{C}}/CP/\mathrm{SIM}$ |
| $e_{\mathrm{B}}/i_{\mathrm{C}i}/\mathrm{INV}\ \ (i_{\mathrm{C}i} \in I_{\mathrm{BC}})$ | $T_{\mathrm{B}}$ | $e_{\mathrm{C}}/CP/\mathrm{SIM}$ |

quantifier "there exists uncountably many" does not have interpolation property [49].

As we described in Sections 2.3 and 2.5, proof obligations we treat in this thesis — namely, invariant preservation (INV), guard strengthening (GRD), and action simulation (SIM) — are denoted as follows:

**INV** $A \land I \land G \land BA1 \Rightarrow i[\mathbf{v}'/\mathbf{v}]$

**GRD** $A \land I \land J \land H \land W \Rightarrow g$

**SIM** $A \land I \land J \land H \land W \land BA2 \Rightarrow a$,

where $G$, $H$, $BA1$, and $BA2$, respectively, denote guards of the abstract event, guards of the concrete event, before-after predicates of the abstract event, and before-after predicates of the concrete event.

After slicing, POs (of INV, GRD, and SIM) in $M_{\mathrm{B}0}$ are classified as follows:

- $e_{\mathrm{B}}/i_{\mathrm{B}}/\mathrm{INV}$, where $e_{\mathrm{B}}$ is an event of $M_{\mathrm{B}0}$ and $i_{\mathrm{B}}$ is an invariant of $M_{\mathrm{B}0}$
- $e_{\mathrm{B}}/g_{\mathrm{A}}/\mathrm{GRD}$, where $e_{\mathrm{B}}$ is an event of $M_{\mathrm{B}0}$ and $g_{\mathrm{A}}$ is a guard of an abstract event of $e_{\mathrm{B}}$ in $M_{\mathrm{A}}$
- $e_{\mathrm{C}}/g_{\mathrm{B}}/\mathrm{GRD}$, where $e_{\mathrm{C}}$ is an event of $M_{\mathrm{C}}$ and $g_{\mathrm{B}}$ is a guard of an abstract event of $e_{\mathrm{C}}$ in $M_{\mathrm{B}0}$
- $e_{\mathrm{B}}/a_{\mathrm{A}}/\mathrm{SIM}$, where $e_{\mathrm{B}}$ is an event of $M_{\mathrm{B}0}$ and $a_{\mathrm{A}}$ is a before-after predicate of an abstract event of $e_{\mathrm{B}}$ in $M_{\mathrm{A}}$
- $e_{\mathrm{C}}/a_{\mathrm{B}}/\mathrm{SIM}$, where $e_{\mathrm{C}}$ is an event of $M_{\mathrm{C}}$ and $a_{\mathrm{B}}$ is a before-after predicate of an abstract event of $e_{\mathrm{C}}$ in $M_{\mathrm{B}0}$

Among these, $e_{\mathrm{C}}/g_{\mathrm{B}}/\mathrm{GRD}$ and $e_{\mathrm{C}}/a_{\mathrm{B}}/\mathrm{SIM}$ can be trivially discharged, since $g_{\mathrm{B}}$ and $a_{\mathrm{B}}$ are obtained through slicing, and thus they are either specified in $e_{\mathrm{C}}$ (i.e., they are included in $H$ or $BA2$ in the hypothesis of POs) or specified in an abstract event of $e_{\mathrm{C}}$ in $M_{\mathrm{A}}$ (i.e., the POs are the same as $e_{\mathrm{C}}/g_{\mathrm{A}}/\mathrm{GRD}$ and $e_{\mathrm{C}}/a_{\mathrm{A}}/\mathrm{SIM}$, which are already proved in $M_{\mathrm{C}}$).

Some of POs of the other three types may require adding CPs to $M_{\mathrm{B}0}$ for proving them. For instance, if $g_{\mathrm{A}}$ is expressible by $V_{\mathrm{B}}$, $e_{\mathrm{B}}/g_{\mathrm{A}}/\mathrm{GRD}$ can be trivially proved, since $g_{\mathrm{A}}$ is included in $G_{\mathrm{B}0}$ (which corresponds to $H$ in the PO) through slicing. Otherwise, the PO $e_{\mathrm{B}}/g_{\mathrm{A}}/\mathrm{GRD}$ may be unprovable.

It is important to note that new POs about consistency of CPs are generated by adding CPs. The types of such POs are shown in Table 4.1. For example, the first row of Table 4.1 shows that when $CP$ is added to $M_{\mathrm{B}}$, it is added as a guard of the intermediate machine and a new PO $e_{\mathrm{C}}/CP/\mathrm{GRD}$ is generated because of

GLc0: $\mathit{mod2}(a' + e')=0 \Rightarrow a' < 1$ (modified `inv_c1`)

↳ GLc1: $\mathit{mod2}(a' + e') \neq 0$

↳ GLc2: $\mathit{mod2}((a + 1) + (f + 2)) \neq 0$ (by `act_c{1,5}`)

↳ GLc3: $\mathit{mod2}(a + f + 1) \neq 0$

↳ GLc4: $\mathit{mod2}(a + f)=0$

↳ GLc5: $\top$ (by `grd_c2`)

Figure 4.7: Proof of `mc/evt_c/inv_c1`/INV

GLb0: $\mathit{mod2}(a' + e')=0 \Rightarrow a' < 1$ (modified `inv_c1`)

↳ GLb1: $\mathit{mod2}(a' + e') \neq 0$

↳ GLb2: $\mathit{mod2}((a + 1) + e') \neq 0$ (by `act_c1`)

↳ GLb3: $\mathit{mod2}(a + e')=0$

↳ GLb4: $\top$ (by `act_NEW`)

Figure 4.8: Proof of `mb/evt_b/inv_c1`/INV

the addition. Therefore, developers need to find appropriate CPs such that such newly generated POs are provable.

The rest of this section describes ways to find CPs.

First, we describe a way to manually find CPs by analyzing proofs of given machines. Our decomposition method relies on this manual process, and thus it is used in case studies described in Section 6.1. Next, we show a possibility of heuristic for automation of this process with a rule-based analysis. Finally, we give another possible heuristic for automation that uses Craig interpolation.

Although our implementation SLICEANDMERGE currently does not support the heuristics described below, we are planning to implement partial support of them in the future.

### Manually Finding CPs from Existing Proofs

The essence of the consistency of $M_A$ and $M_C$ can be found by examining the proof of consistency and making an inference.

For example, the proof of `mc/evt_c/inv_c1`/INV can be summarized in terms of goals (succedent), as shown in Figure 4.7. The initial goal GLc0: $\mathrm{mod2}(a' + e') = 0 \Rightarrow a' < 1$ can be derived because GLc1: $\mathrm{mod2}(a' + e') \neq 0$ can be derived from hypotheses including a guard `grd_c2`.

A proof with the same root goal is possible using the vocabulary of `mb` if the goal GLb1: $\mathrm{mod2}(a' + e') \neq 0$ can be derived from an event-local predicate (guard or BAP) $p$ that is expressible by $V_B$. GLb1 can be transformed into GLb3: $\mathrm{mod2}(a + e') = 0$ by `act_c1`. We need to find $p$ such that GLb3 can be derived from $p$ because there is no predicate about $e'$ in `mb0`. A solution is to view GLb3 itself as $p$ and add an action, such as `act_NEW` : $e$ `:|` $\mathrm{mod2}(a + e') = 0$ to `mb0` (Figure 4.8).

### Heuristic for Finding CPs Using Rule-based Analysis

In some cases, part of a predicate is expressible by $V_{\mathrm{B}}$, but the remainder of it is not; thus, the predicate cannot be obtained through predicate-level slicing. Simple heuristics can be used to find parts of such predicates that are expressible by $V_{\mathrm{B}}$. For instance, a predicate $0 \leq a$ can be found by extracting a part that is expressible by $V_{\mathrm{B}}$ from `mc/evt_c/grd_c1` ( $0 \leq a \land 0 \leq c$ ). A possible implementation of this is to convert predicates into conjunctive normal form and extract clauses that are expressible by $V_{\mathrm{B}}$.

### Heuristic for Finding CPs as Craig Interpolant

In this section, we describe a heuristic to deal with a difficult problem of finding CPs by using Craig interpolation [26].

The goal of this heuristic is to find a formula $\phi$ such that an interpolant of $\phi$ will function as a CP, namely, adding $\phi$ to the specification of the intermediate machine discharges POs of the machine. Algorithms to obtain interpolant are not included in this thesis, and thus, this heuristic is not evaluated in the case studies (Section 6.1).

Because the applicability of the interpolation theorem is limited, this heuristic is applicable only to specific classes of expression of Event-B models. We discuss the applicability of this heuristic in Section 8.1.1.

We describe a general method to find CPs as Craig interpolants below.

One possible pattern of lack of consistency is about guard strengthening (GRD) PO. Let us consider a guard $g_{\mathrm{A}i}$ that is specified in an event $e_{\mathrm{A}}$ in $M_{\mathrm{A}}$. From the assumption on consistency of "$M_{\mathrm{C}}$ refines $M_{\mathrm{A}}$," a PO $M_{\mathrm{C}}/e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}$ has been successfully discharged, where $e_{\mathrm{C}}$ is an event that refines $e_{\mathrm{A}}$.

The formula of the PO $M_{\mathrm{C}}/e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}$ is as follows:

$$I_{\mathrm{A}} \land I_{\mathrm{C}} \land G_{\mathrm{C}} \Rightarrow g_{\mathrm{A}i}. \tag{4.3}$$

Formula 4.3 is equivalent to the following (Formula 4.4):

$$\tilde{I_{\mathrm{A}}} \land I_{\mathrm{AB}} \land I_{\mathrm{BC}} \land \tilde{I_{\mathrm{C}}} \land G_{\mathrm{BC}} \land \tilde{G_{\mathrm{C}}} \Rightarrow g_{\mathrm{A}i}, \tag{4.4}$$

and thus, it is also equivalent to the following (Formula 4.5):

$$I_{\mathrm{AB}} \land \tilde{I_{\mathrm{C}}} \land \tilde{G_{\mathrm{C}}} \Rightarrow g_{\mathrm{A}i} \lor \neg\tilde{I_{\mathrm{A}}} \lor \neg I_{\mathrm{BC}} \lor \neg G_{\mathrm{BC}}. \tag{4.5}$$

Let $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ be the interpolant of Formula 4.5.

The problem is that the guard strengthening PO of the same guard by $M_{\mathrm{B}}$ ($M_{\mathrm{B}}/e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}$) may not be provable due to lack of necessary predicates.

Here, $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ works as a CP for the PO $M_{\mathrm{B}}/e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}$. In other words, $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ obtained as the interpolant of Formula 4.5 is expressible by $V_{\mathrm{B}}$ (we show this as Theorem 4.6.1 in Section 4.6), and thus, it can be added to the event $e_{\mathrm{B}}$ as a guard.

If $\mathcal{I}$ is a Craig interpolant of a formula $A \Rightarrow B$, then $A \Rightarrow \mathcal{I}$, $\mathcal{I} \Rightarrow B$, and $\mathrm{var}(\mathcal{I}) \subseteq \mathrm{var}(A) \cap \mathrm{var}(B)$. Hence, if $\mathcal{J}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ is a Craig interpolant of Formula 4.3, then $\mathrm{var}(\mathcal{J}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}) \subseteq \mathrm{var}(I_{\mathrm{A}} \land I_{\mathrm{C}} \land G_{\mathrm{C}}) \cap \mathrm{var}(g_{\mathrm{A}i})$. As we described above, $g_{\mathrm{A}i}$ cannot be expressible by $V_{\mathrm{B}}$ (i.e., $\mathrm{var}(g_{\mathrm{A}i}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset$) when we need to find a CP to discharge the PO. Moreover, $\mathrm{var}(I_{\mathrm{C}}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset$ because $I_{\mathrm{C}}$ includes gluing invariants of $\tilde{V_{\mathrm{A}}}$. Thus, in this case, $\mathrm{var}(\mathcal{J}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset$ may hold (i.e., the interpolant $\mathcal{J}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ may be *inexpressible* by $V_{\mathrm{B}}$). Therefore, converting Formula 4.3 into Formula 4.5 before interpolating is necessary to make sure that the interpolants are expressible by $V_{\mathrm{B}}$, and thus they work as CPs to discharge GRD POs [3].

Moreover, as we show as Theorems 4.4.1 and 4.4.2, $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ satisfies the following conditions:

- The PO $M_{\mathrm{B}}/e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}$ becomes provable by adding $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ as an additional guard of $e_{\mathrm{B}}$
- A GRD PO ($M_{\mathrm{C}}/e_{\mathrm{C}}/\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}/\mathrm{GRD}$) that is generated (as shown in the first row of Table 4.1) by adding $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ as a guard of $M_{\mathrm{B0}}$ is provable

**Theorem 4.4.1** *By adding a predicate $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$, which is obtained as a Craig interpolant of Formula 4.5, to an intermediate machine $M_{\mathrm{B}}$, the PO $M_{\mathrm{B}}/e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}$ becomes dischargeable.*

**Proof** Because $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ is an interpolant of Formula 4.5, the following is true:

$$\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}} \Rightarrow g_{\mathrm{A}i} \vee \neg\tilde{I_{\mathrm{A}}} \vee \neg I_{\mathrm{BC}} \vee \neg G_{\mathrm{BC}}. \quad (4.6)$$

From Formula 4.6, the following is true:

$$I_{\mathrm{AB}} \wedge G_{\mathrm{AB}} \wedge \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}} \Rightarrow g_{\mathrm{A}i} \vee \neg\tilde{I_{\mathrm{A}}} \vee \neg I_{\mathrm{BC}} \vee \neg G_{\mathrm{BC}}. \quad (4.7)$$

Formula 4.7 is equivalent to the following formula (Formula 4.8):

$$\tilde{I_{\mathrm{A}}} \wedge I_{\mathrm{AB}} \wedge I_{\mathrm{BC}} \wedge G_{\mathrm{AB}} \wedge G_{\mathrm{BC}} \wedge \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}} \Rightarrow g_{\mathrm{A}i}. \quad (4.8)$$

Thus, it is also equivalent to the following formula (Formula 4.9):

$$I_{\mathrm{A}} \wedge I_{\mathrm{B}} \wedge G_{\mathrm{B0}} \wedge \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}} \Rightarrow g_{\mathrm{A}i}. \quad (4.9)$$

This formula (Formula 4.9) is nothing but the PO $M_{\mathrm{B}}/e_{\mathrm{B}}/g_{\mathrm{A}i}/\mathrm{GRD}$ after adding $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ as an additional guard of event $e_B$ in $M_{\mathrm{B}}$. Thus, the PO becomes dischargeable by adding $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ as an additional guard of event $e_B$ in $M_{\mathrm{B}}$. □

**Theorem 4.4.2** *The event $e_{\mathrm{C}}$ in $M_{\mathrm{C}}$ strengthens the additional guard $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ in $e_{\mathrm{B}}$ in $M_{\mathrm{B}}$.*

**Proof** Because $\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}$ is an interpolant of Formula 4.5, the following is true:

$$I_{\mathrm{AB}} \wedge \tilde{I_{\mathrm{C}}} \wedge \tilde{G_{\mathrm{C}}} \Rightarrow \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}. \quad (4.10)$$

From Formula 4.10, the following is true:

$$(\tilde{I_{\mathrm{A}}} \wedge I_{\mathrm{BC}} \wedge G_{\mathrm{BC}}) \wedge I_{\mathrm{AB}} \wedge \tilde{I_{\mathrm{C}}} \wedge \tilde{G_{\mathrm{C}}} \Rightarrow \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}. \quad (4.11)$$

Formula 4.11 is equivalent to the following formula (Formula 4.12):

$$I_{\mathrm{A}} \wedge I_{\mathrm{B}} \wedge I_{\mathrm{C}} \wedge G_{\mathrm{C}} \Rightarrow \mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}. \quad (4.12)$$

This formula (Formula 4.12) is nothing but the PO $M_{\mathrm{C}}/e_{\mathrm{C}}/\mathcal{I}_{e_{\mathrm{C}}/g_{\mathrm{A}i}/\mathrm{GRD}}/\mathrm{GRD}$. Thus, the PO is dischargeable. □

Similar ways to obtain interpolants can be used for action simulation (SIM) POs and invariant preservation (INV) POs.

In addition, all newly generated POs (Table 4.1) by adding CPs obtained by the heuristic are provable.

[3] Conversion in the same manner is also necessary for SIM POs but not necessary for INV POs.

## 4.5 Restructuring

We call a sequence of machines $[M_n, M_{n+1}, \dots M_{m-1}, M_m]$ a *refinement chain* (RC) if $M_{i+1}$ refines $M_i$ for every natural number $i$ such that $n \leq i < m$.

In addition to decomposing, we can *merge* refinements as follows: When there is an RC $[M_0, M_1, M_2]$, merging $M_1$ and $M_2$ constructs a new machine $M_{12}$ such that $M_{12}$ refines $M_0$. $M_{12}$'s variables, invariants, and events are composed of the unions of the variables, invariants, and events of $M_1$ and $M_2$.

Suppose that an RC $[M_\mathrm{A}, M_\mathrm{B}, M_\mathrm{C}]$ is provided. The algorithm of merging the RC to construct a machine $M_\mathrm{D}$ such that an RC $[M_\mathrm{A}, M_\mathrm{D}]$ is formed is shown in Algorithm 3.

**Algorithm 3** Merging of machines

```
1: $V_\mathrm{D} \leftarrow V_\mathrm{B} \cup V_\mathrm{C}$
2: $I_\mathrm{D} \leftarrow I_\mathrm{B} \cup I_\mathrm{C}$
3: $E_\mathrm{D} \leftarrow \emptyset$
4: for all $e_\mathrm{A} \in E_\mathrm{A}$ do
5:     for all $e_\mathrm{B} \in E_\mathrm{B}$ s.t. $e_\mathrm{A} \in \mathrm{refines}(e_\mathrm{B})$ do
6:         for all $e_\mathrm{C} \in E_\mathrm{C}$ s.t. $e_\mathrm{B} \in \mathrm{refines}(e_\mathrm{C})$ do
7:             $e_\mathrm{D} \leftarrow$ (empty event)
8:             $\mathrm{refines}(e_\mathrm{D}) \leftarrow \{e_\mathrm{A}\}$
9:             $\mathrm{guards}(e_\mathrm{D}) \leftarrow \mathrm{guards}(e_\mathrm{B}) \cup \mathrm{guards}(e_\mathrm{C})$
10:            $\mathrm{actions}(e_\mathrm{D}) \leftarrow \mathrm{actions}(e_\mathrm{B}) \cup \mathrm{actions}(e_\mathrm{C})$
11:            $E_\mathrm{D} \leftarrow E_\mathrm{D} \cup \{e_\mathrm{D}\}$
12:        end for
13:    end for
14: end for
15: for all $e_\mathrm{D} \in E_\mathrm{D}$ do
16:     for all $f_\mathrm{D} \in (E_\mathrm{D} \setminus e_\mathrm{D})$ do
17:         if $\mathrm{guards}(e_\mathrm{D}) = \mathrm{guards}(f_\mathrm{D}) \wedge \mathrm{actions}(e_\mathrm{D}) = \mathrm{actions}(f_\mathrm{D})$ then
18:             $\mathrm{refines}(e_\mathrm{D}) \leftarrow \mathrm{refines}(e_\mathrm{D}) \cup \mathrm{refines}(f_\mathrm{D})$
19:             $E_\mathrm{D} \leftarrow E_\mathrm{D} \setminus f_\mathrm{D}$
20:         end if
21:     end for
22: end for
```

All variables and invariants of $M_\mathrm{B}$ and $M_\mathrm{C}$ are collected to be specified in $M_\mathrm{D}$ (Lines 1 and 2). In order to preserve refinement structures of events in the original RC $[M_\mathrm{A}, M_\mathrm{B}, M_\mathrm{C}]$, an event of $M_\mathrm{D}$ is created for each pair of $e_\mathrm{B}$ and $e_\mathrm{C}$ such that $e_\mathrm{C}$ refines $e_\mathrm{B}$ (Lines 5–13). Each event in $M_\mathrm{D}$ that is created from a pair of events $(e_\mathrm{B}, e_\mathrm{C})$ refines events of $M_\mathrm{A}$ that are refined by $e_\mathrm{B}$ (Line 8) and have all guards and actions of $e_\mathrm{B}$ and $e_\mathrm{C}$ (Lines 9–10). This procedure may create events $E$ with different 'refines' clauses and the same guards and the same actions. Such events are combined into an event that refines all events of $M_\mathrm{A}$ that are refined by $E$ (Lines 17–20).

Refinements can be restructured by merging and decomposing refinements. Suppose that an RC $[M_n, \cdots, M_m]$ is given. First, machines $(M_i)_{i=n+1}^{m}$ are merged as $M'_m$, which directly refines $M_n$. Then, an RC $[M_n, M'_m]$ is decomposed by constructing new machines $(\tilde{M}_i)_{i=k+1}^{l}$ that reflect the user's preference of aspects in terms of $V_{\mathrm{B}0}$. As a result, the refinement is restructured into an RC $[M_n = \tilde{M}_k, \tilde{M}_{k+1}, \dots, \tilde{M}_{l-1}, \tilde{M}_l = M'_m]$. Then, as a result of restructuring refinements, the understandability of a specification increases because the meaning of each refinement step can be changed as the user likes. In Section 6.1.2, we describe an application of the restructuring method to extract parts of an existing model for reuse.

In addition, refinement restructuring enables us to reuse models. Suppose that a user has a system $S$, an RC $[M_{S0}, M_{S1}, \ldots, M_{S(n-1)}, M_{Sn}]$ for $S$, and another system $\tilde{S}$ such that $S$ and $\tilde{S}$ has several states and properties in common. In this case, several abstract machines for $S$ may be reused to construct machines for $\tilde{S}$ such that an RC $[M_{S0}, M_{S1}, \ldots, M_{Sl}, M_{\tilde{S}(l+1)}, \ldots, M_{\tilde{S}m}]$ exists. Our refinement restructuring method increases the reusable parts of machines, regardless of the original refinement structure, by extracting platform-independent machines (or $\tilde{S}$-related aspects of $S$'s machines) as abstract machines.

## 4.6 Feasibility of Finding CPs as Craig Interpolant

In this section, we show that Craig interpolants obtained by the heuristic described in Section 4.4.2 are always expressible by $V_{\mathrm{B}}$ as Theorem 4.6.1. To this end, we first investigate the possibility of occurrences of variable in each symbol of 4.5 as lemmas and then use them to show the theorem.

As we discuss in Section 8.1.1, the heuristic is applicable only if Craig's interpolation theorem is applicable to the predicates in proofs of given Event-B models. For instance, the heuristic is applicable if predicates in the proofs are reducible to first-order ones. Therefore, in this section, we assume that the given models are expressed only with predicates of such classes.

**Lemma 4.6.1** $\mathrm{var}(\tilde{I_{\mathrm{A}}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{AB}} \cup V_{\mathrm{ABC}}$.

**Proof** Because $\tilde{I_{\mathrm{A}}}$ is a subset of $I_{\mathrm{A}}$,

$$\mathrm{var}(\tilde{I_{\mathrm{A}}}) \subseteq \mathrm{var}(I_{\mathrm{A}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{AB}} \cup V_{\mathrm{ABC}}.$$

□

**Lemma 4.6.2** $\mathrm{var}(I_{\mathrm{AB}}) \subseteq V_{\mathrm{AB}} \cup V_{\mathrm{ABC}}$.

**Proof** Because $I_{\mathrm{AB}}$ is a subset of $I_{\mathrm{A}}$,

$$\mathrm{var}(I_{\mathrm{AB}}) \subseteq \mathrm{var}(I_{\mathrm{A}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{AB}} \cup V_{\mathrm{ABC}}. \tag{4.13}$$

By definition, variables of $\tilde{V_{\mathrm{A}}}$ do not appear in $I_{\mathrm{AB}}$. Thus,

$$\mathrm{var}(I_{\mathrm{AB}}) \cap \tilde{V_{\mathrm{A}}} = \emptyset. \tag{4.14}$$

From 4.13 and 4.14,

$$\mathrm{var}(I_{\mathrm{AB}}) \subseteq V_{\mathrm{AB}} \cup V_{\mathrm{ABC}}.$$

□

**Lemma 4.6.3** $\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap V_{\mathrm{AB}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{C}}} \neq \emptyset$.

**Proof** From Assumption 4.3.1, all invariants in $M_{\mathrm{C}}$ that include variables in $V_{\mathrm{AB}}$ are gluing invariants. Thus,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap V_{\mathrm{AB}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap (V_{\mathrm{BC}} \cup \tilde{V_{\mathrm{C}}}) \neq \emptyset. \tag{4.15}$$

From the definition of $V_{\mathrm{AB}}$ (Formula 4.1), gluing invariants that include variables in $V_{\mathrm{AB}}$ also include variables in $\tilde{V_{\mathrm{C}}}$. Therefore, from 4.1 and 4.15,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap V_{\mathrm{AB}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{C}}} \neq \emptyset.$$

□

**Lemma 4.6.4** $\mathrm{var}(I_{\mathrm{BC}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{ABC}} \cup V_{\mathrm{BC}}$.

**Proof** Because $I_{\mathrm{BC}}$ is a subset of $I_{\mathrm{C}}$,

$$\mathrm{var}(I_{\mathrm{BC}}) \subseteq \mathrm{var}(I_{\mathrm{C}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{AB}} \cup V_{\mathrm{ABC}} \cup V_{\mathrm{BC}} \cup \tilde{V_{\mathrm{C}}}. \tag{4.16}$$

From Lemma 4.6.3 (variables of $V_{\mathrm{AB}}$ can appear only in gluing invariants of $M_{\mathrm{C}}$ such that the gluing invariants replace $V_{\mathrm{AB}}$ with $\tilde{V_{\mathrm{C}}}$),

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap V_{\mathrm{AB}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{C}}} \neq \emptyset,$$

and thus,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{C}}} = \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap V_{\mathrm{AB}} = \emptyset. \tag{4.17}$$

In addition, by definition, variables of $\tilde{V_{\mathrm{C}}}$ do not appear in $I_{\mathrm{BC}}$. Thus,

$$\mathrm{var}(I_{\mathrm{BC}}) \cap \tilde{V_{\mathrm{C}}} = \emptyset. \tag{4.18}$$

Because $I_{\mathrm{BC}} \subseteq I_{\mathrm{C}}$, from 4.17 and 4.18,

$$\mathrm{var}(I_{\mathrm{BC}}) \cap V_{\mathrm{AB}} = \emptyset. \tag{4.19}$$

From 4.16, 4.18, and 4.19,

$$\mathrm{var}(I_{\mathrm{BC}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{ABC}} \cup V_{\mathrm{BC}}.$$

□

**Lemma 4.6.5** $\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap V_{\mathrm{BC}} \neq \emptyset$.

**Proof** From Assumption 4.3.1, all invariants in $M_{\mathrm{C}}$ that include variables in $\tilde{V_{\mathrm{A}}}$ are gluing invariants. Thus,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap (V_{\mathrm{BC}} \cup \tilde{V_{\mathrm{C}}}) \neq \emptyset. \tag{4.20}$$

From the definition of $\tilde{V_{\mathrm{A}}}$ (Formula 4.2), every gluing invariant that includes variables in $\tilde{V_{\mathrm{A}}}$ does *not* include variables in $\tilde{V_{\mathrm{C}}}$. Therefore, from 4.2 and 4.20,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset \Rightarrow \mathrm{var}(\{i_C\}) \cap V_{\mathrm{BC}} \neq \emptyset.$$

□

**Lemma 4.6.6** $\mathrm{var}(\tilde{I_{\mathrm{C}}}) \subseteq V_{\mathrm{AB}} \cup V_{\mathrm{ABC}} \cup V_{\mathrm{BC}} \cup \tilde{V_{\mathrm{C}}}$.

**Proof** Because $\tilde{I_{\mathrm{C}}}$ is a subset of $I_{\mathrm{C}}$,

$$\mathrm{var}(\tilde{I_{\mathrm{C}}}) \subseteq \mathrm{var}(I_{\mathrm{C}}) \subseteq \tilde{V_{\mathrm{A}}} \cup V_{\mathrm{AB}} \cup V_{\mathrm{ABC}} \cup V_{\mathrm{BC}} \cup \tilde{V_{\mathrm{C}}}. \tag{4.21}$$

From Lemma 4.6.5 (in $M_{\mathrm{C}}$, variables of $\tilde{V_{\mathrm{A}}}$ can appear only in gluing invariants of $M_{\mathrm{C}}$ such that the gluing invariants replace $\tilde{V_{\mathrm{A}}}$ with $V_{\mathrm{BC}}$),

$$\tilde{V_{\mathrm{A}}} = \{\tilde{v_A} \in (V_{\mathrm{A}} \setminus V_{\mathrm{C}}) \,|\, \forall i_C \in I_{\mathrm{C}} \,.\, \tilde{v_A} \in \mathrm{var}(\{i_C\}) \Rightarrow \mathrm{var}(\{i_C\}) \subseteq (V_{\mathrm{A}} \cup V_{\mathrm{B}})\}.$$

Therefore,

$$\forall i_C \in I_{\mathrm{C}} \,.\, \mathrm{var}(\{i_C\}) \cap \tilde{V_{\mathrm{A}}} \neq \emptyset \Rightarrow i_C \in I_{\mathrm{BC}}. \tag{4.22}$$

Because $V_{\text{BC}} \cap \tilde{V_{\text{C}}} = \emptyset$ and 4.22,

$$\forall i_C \in I_{\text{C}} \,.\, \text{var}(\{i_C\}) \cap \tilde{V_{\text{A}}} \neq \emptyset \Rightarrow i_C \notin \tilde{I_{\text{C}}}.$$

Thus,

$$\text{var}(\tilde{I_{\text{C}}}) \cap \tilde{V_{\text{A}}} = \emptyset. \tag{4.23}$$

From 4.21 and 4.23,

$$\text{var}(\tilde{I_{\text{C}}}) \subseteq V_{\text{AB}} \cup V_{\text{ABC}} \cup V_{\text{BC}} \cup \tilde{V_{\text{C}}}.$$

□

**Lemma 4.6.7** $\text{var}(\tilde{G_{\text{A}}}) \subseteq \tilde{V_{\text{A}}} \cup V_{\text{AB}} \cup V_{\text{ABC}}$.

**Proof** Because $\tilde{G_{\text{A}}}$ is a subset of $G_{\text{A}}$,

$$\text{var}(\tilde{G_{\text{A}}}) \subseteq \text{var}(G_{\text{A}}) \subseteq \tilde{V_{\text{A}}} \cup V_{\text{AB}} \cup V_{\text{ABC}}. \tag{4.24}$$

□

**Lemma 4.6.8** $\text{var}(G_{\text{AB}}) \subseteq V_{\text{AB}} \cup V_{\text{ABC}}$.

**Proof** Because $G_{\text{AB}}$ is a subset of $G_{\text{A}}$,

$$\text{var}(G_{\text{AB}}) \subseteq \text{var}(G_{\text{A}}) \subseteq \tilde{V_{\text{A}}} \cup V_{\text{AB}} \cup V_{\text{ABC}}. \tag{4.25}$$

By definition, variables of $\tilde{V_{\text{A}}}$ do not appear in $G_{\text{AB}}$. Thus,

$$\text{var}(G_{\text{AB}}) \cap \tilde{V_{\text{A}}} = \emptyset. \tag{4.26}$$

From 4.25 and 4.26,

$$\text{var}(G_{\text{AB}}) \subseteq V_{\text{AB}} \cup V_{\text{ABC}}.$$

□

**Lemma 4.6.9** $\text{var}(G_{\text{BC}}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}}$.

**Proof** Because $G_{\text{BC}}$ is a subset of $G_{\text{C}}$,

$$\text{var}(G_{\text{BC}}) \subseteq \text{var}(G_{\text{C}}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}} \cup \tilde{V_{\text{C}}}. \tag{4.27}$$

In addition, by definition, variables of $\tilde{V_{\text{C}}}$ do not appear in $G_{\text{BC}}$. Thus,

$$\text{var}(G_{\text{BC}}) \cap \tilde{V_{\text{C}}} = \emptyset. \tag{4.28}$$

From 4.27 and 4.28,

$$\text{var}(G_{\text{BC}}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}}.$$

□

**Lemma 4.6.10** $\text{var}(\tilde{G_{\text{C}}}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}} \cup \tilde{V_{\text{C}}}$.

**Proof** Because $\tilde{G_{\text{C}}}$ is a subset of $G_{\text{C}}$,

$$\text{var}(\tilde{G_{\text{C}}}) \subseteq \text{var}(G_{\text{C}}) \subseteq V_{\text{ABC}} \cup V_{\text{BC}} \cup \tilde{V_{\text{C}}}.$$

□

**Theorem 4.6.1** *Let $W_{\text{B}}$ be the set of variables that occur before "⇒" in Formula 4.5 (which is equivalent to formula of PO $M_{\text{C}}/e_{\text{C}}/g_{\text{A}i}/\text{INV}$) and $W_{\text{A}}$ be the set of variables that occur after "⇒" in Formula 4.5. Then, the intersection of $W_{\text{B}}$ and $W_{\text{A}}$ is a subset of $V_{\text{B}}$. Thus, a Craig interpolant $\mathcal{I}_{e_{\text{C}}/g_{\text{A}i}/\text{GRD}}$ can always be expressible by $V_{\text{B}}$.*

**Proof** Let $\alpha$ be the part before "⇒" of Formula 4.5 and $\beta$ be the part after "⇒" of the formula. Thus,

$$\alpha = I_{\text{AB}} \wedge \tilde{I_{\text{C}}} \wedge \tilde{G_{\text{C}}},$$
$$\beta = g_{\text{A}i} \vee \neg\tilde{I_{\text{A}}} \vee \neg I_{\text{BC}} \vee \neg G_{\text{BC}}.$$

$g_{\text{A}i}$ is an arbitrary member of $\tilde{G_{\text{A}}}$, and thus,

$$\text{var}(\{g_{\text{A}i}\}) \subseteq \text{var}(\tilde{G_{\text{A}}}) \subseteq \tilde{V_{\text{A}}} \cup V_{\text{AB}} \cup V_{\text{ABC}}.$$

From Lemmas 4.6.2, 4.6.6, and 4.6.10,

$$\text{var}(\alpha) \subseteq \text{var}(I_{\text{AB}}) \cup \text{var}(\tilde{I_{\text{C}}}) \cup \text{var}(\tilde{G_{\text{C}}}) \tag{4.29}$$
$$\subseteq V_{\text{AB}} \cup V_{\text{ABC}} \cup V_{\text{BC}} \cup \tilde{V_{\text{C}}}. \tag{4.30}$$

Moreover, from Lemmas 4.6.7, 4.6.1, 4.6.4, and 4.6.9,

$$\text{var}(\beta) \subseteq \text{var}(\tilde{G_{\text{A}}}) \cup \text{var}(\tilde{I_{\text{A}}}) \cup \text{var}(I_{\text{BC}}) \cup \text{var}(G_{\text{BC}}) \tag{4.31}$$
$$\subseteq \tilde{V_{\text{A}}} \cup V_{\text{AB}} \cup V_{\text{ABC}} \cup V_{\text{BC}}. \tag{4.32}$$

From 4.30 and 4.32,

$$\begin{aligned}\text{var}(\{\mathcal{I}_{e_{\text{C}}/g_{\text{A}i}/\text{GRD}}\}) &\subseteq \text{var}(\alpha) \cap \text{var}(\beta)\\ &\subseteq V_{\text{AB}} \cup V_{\text{ABC}} \cup V_{\text{BC}}\\ &\subseteq V_{\text{B}}.\end{aligned}$$

□

# Chapter 5

# Refinement Planning

## 5.1 Approach

Our view of the planning problem and an approach to solve it are as follows.

Although our planning method is applicable to plan events in refinement steps, since we aim to search in solution spaces of high-layered sketches of models, we focus on essential information of Event-B models, namely context-wide and machine-wide elements and properties. Thus, we treat variables and constants as elements. Properties we treat are written as axioms and invariants.

Because our planning approach is intended to be used before actual modeling, given information about the target system is poorer than the information for refactoring. Given information (dependencies) are as follows:

- List of properties (a subset of direct and implied properties of the most concrete model) without actual expressions of them
- Elements that appear in expressions of each property
- Dependencies between elements such as lists of carrier sets and constants that are required for typing variables
- Specifications about introduction order of elements such as an order to introduce element of a composed system earlier than elements of its subsystems

The feasibility of obtaining the information above is discussed in Chapter 7.

Our goal is to show possible ways to introduce elements and properties (refinement plans) through refinement. Thus, according to the input information, our method calculates possibilities of sequences $[(E_0, P_0), (E_1, P_1), \ldots, (E_n, P_n)]$ such that $E_i$ and $P_i$, respectively, denote the sets of elements and properties of the $i$'th refinement step. In other words, our method eliminates obviously meaningless sequences according to the input.

The relationships of dependencies we pay attention to are as follows:

**Elements occurrence** (Relationship between a property and elements.) In order to specify a property, elements that appear in the property need to be specified in the model.

**Static elements for typing** (Relationship between an element and elements.) For typing of elements, some elements require other elements to be included in the model.

**Abstract before concrete** (Relationship between an element and an element.) If an element $e$ is considered to be an abstract version of another element $f$, then $e$ should not be introduced later than $f$.

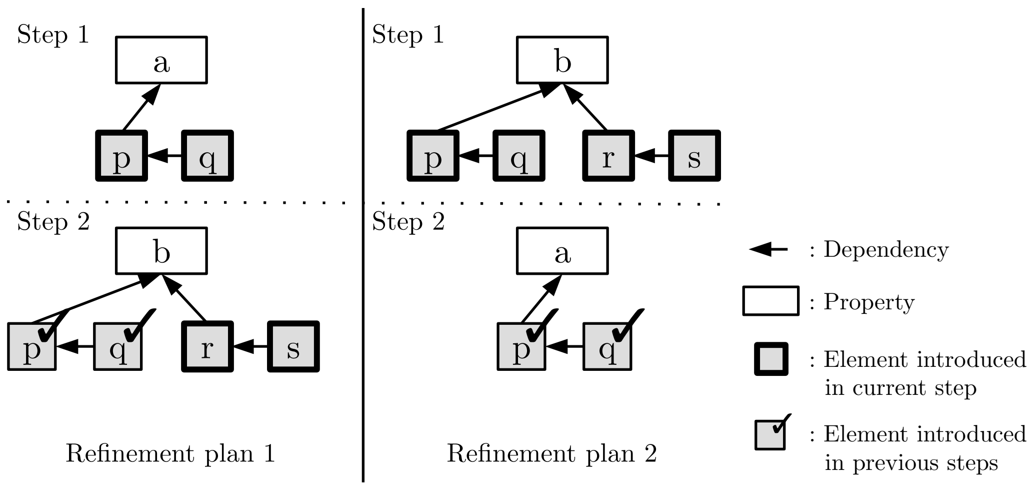


Figure 5.1: Difference between two refinement plans

**Conceptually similar properties** (Relationship between a property and properties.) If elements that are required to introduce property $p$ in a model $M$ allow specifying other properties $P$, then $P$ should also be introduced to $M$.

## 5.2 Rationales

### 5.2.1 Core Rationale for Refinement Planning: Avoiding Meaningless Refinements

This section describes a general rationale for deriving "good" refinement strategies. The rationale is important for mitigating complexity without relying on knowledge of experienced developers. It is also important for avoiding failures to discharge proof obligations.

Suppose a simple system is composed of two properties, $p$ and $q$, as illustrated in Figure 5.1. Sets of elements are necessary to specify and verify the properties, as the arrows in Figure 5.1 shows. There are also dependencies between elements. Refinement plan 1 shows one refinement plan in which $a$ is introduced in the first step and $b$ in the second step. The first step introduces elements $p$ and $q$ according to the dependencies. The second step introduces (only) $r$ and $s$ (enclosed by bold lines) under the dependencies and the fact that $p$ and $q$ have already been introduced (with the check mark). Refinement plan 2 has the opposite order of plan 1. In this case, the second step becomes *meaningless* since the first one introduces all the elements. The second step only requires one to verify $p$ without extending the previous model (the verification can actually be done in the first step at the same time). Plan 2, thus, fails to mitigate complexity.

In order to pass type checking of Event-B, state variables and constants need the carrier sets used to type them. For example, a variable that represents the state of a traffic light requires a carrier set of "colors of traffic lights." The carrier set is specified by the constants of *green* and *red*. These are dependencies between elements, i.e., between the state variable and the typing carrier set (typed_using($e$)), as well as between the carrier set and constants (contained_by($e$)).

Given the dependencies, a property $p$ depends on elements dep($p$), introduced in the same step, such that
$\text{dep}(p) = \bigcup_{e \in \text{inc}(p)} (\text{typed\_using}(e) \cup \text{contained\_by}(e))$, where inc($p$) denotes el-

ements that directly appear in $p$. A set of properties $P$ depends on elements $\mathrm{DEP}(P) = \bigcup_{p \in P} \mathrm{dep}(p)$.

In the refinement plan $PL = [P_1, P_2, \cdots, P_n, \cdots]$, the elements introduced in step $n$ are $\mathrm{intro}(PL, n) = \mathrm{DEP}(P_n) - \bigcup_{i=1 \cdots n-1} \mathrm{DEP}(P_i)$. This is the set of elements required for the properties $P_n$ that have not yet been introduced.

As a result, some plans include meaningless steps that do not introduce any elements when the required ones have already been introduced in the previous steps. Such plans are considered meaningless, failing to mitigate complexity. In the case of Figure 5.1, $\mathrm{intro}([\{q\}, \{p\}], 2) = \mathrm{DEP}(\{p\}) - \mathrm{DEP}(\{q\}) = \emptyset$ as $\mathrm{DEP}(\{p\}) \subset \mathrm{DEP}(\{q\})$. Suppose now a different situation where $\mathrm{DEP}(\{p\}) = \mathrm{DEP}(\{q\})$. In this case, $\mathrm{intro}([\{p\}, \{q\}], 2) = \mathrm{intro}([\{q\}, \{p\}], 2) = \emptyset$; thus, only the choice of plan $[\{p, q\}]$ remains valid.

By using the information of $\mathrm{dep}(p)$, our method avoids generating meaningless steps as follows: Suppose there is a set of properties $Q$ such that $\mathrm{DEP}(Q) \subset \bigcup_{i=1 \cdots n} \mathrm{DEP}(P_i)$. This means a step that introduces $Q$ later than the $n$th step ($[P_1, \cdots, P_n, \cdots, Q, \cdots]$) makes the $(n+1)$th step meaningless. Such a situation can be avoided by introducing $Q$ with other properties not included in $Q$, namely by taking plans such as (1) $[P_1, \cdots, (P_n \cup Q), \cdots]$ (where $\mathrm{DEP}(Q) \subset \bigcup_{i=1 \cdots n} \mathrm{DEP}(P_i)$ and $\mathrm{DEP}(Q) \not\subset \bigcup_{i=1 \cdots n-1} \mathrm{DEP}(P_i)$) or (2) $[P_1, \cdots, P_n, \cdots, (Q \cup R), \cdots]$. However, case (2) becomes meaningless since $Q$ has nothing to do with $R$. In contrast, by merging in case (1)'s way, properties are introduced as soon as all elements required for them are introduced, and thus, all steps introduce elements.

Elimination based on the above rationale leads to refinement plans that

- introduce abstract properties (depend on fewer elements) earlier and concrete properties (depend on more elements) later, and
- introduce conceptually similar properties (i.e., ones depending on the same set of elements) together.

Thus, although the rules themselves may seem to be artificial, they derive intuitively "good" plans.

### 5.2.2 Rationales for Common Refinement Strategies

Further rationales are presented here to eliminate "meaningless" plans in terms of common refinement intentions. These rationales are represented as *merging* or *ordering* rules of elements or properties. An ordering rule "$x$ no later than $y$" eliminates plans that introduce $y$ earlier than $x$. A merging rule "$x$ and $y$ together" eliminates plans that introduce them in different steps.

#### Merging of Constrained and Constraining Elements

If an element represents a state (variable) and has a specific domain or constraint, the element should be introduced together with the elements that represent the domain. In the island example, the number of cars on the bridge and island should be introduced together with their upper bound (capacity). If this merging is not used, the refinement steps seem to be too granular to delay introducing constraints on one variable. This rule is concrete enough so as not to require suggestions from experienced persons; it only requires a systematic check of state elements and the domains or constraints on each of them. On the other hand, it might be acceptable to introduce the state first, by constructing an abstract model in which the state changes freely (more freely than the actual case), then introduce the constraint later. The other ordering is meaningless as the first model would include constraining elements (capacity) not used for anything.

#### Ordering by Subsystem Decomposition

If some elements represent states or behaviors inside a subsystem, they should be introduced no earlier than those about the states or behaviors of the enclosing system. In the island example, the element "the number of cars on the island" should be introduced later than the element "the number of cars outside the mainland." If this ordering is not used (and the opposite order is used instead), a meaningless step appears where the internal states of a subsystem can change freely without considering the global constraints. The subsequent steps for introducing the global constraints also become complex because the details of the subsystems have already been introduced. This rule can be identified with a systematic check if the specification calls for a decomposition of the system into subsystems, as is common in various specification and modeling formats.

#### Ordering by Controlled and Controlling Elements

If some of the elements represent the "controlling" states of specific means, such as sensors or actuators, they should be introduced no earlier than those about the "controlled" states that directly appear in the top-level requirements (often invariants). In the island example, "traffic light on the mainland side" controls "the number of cars outside the mainland." Thus, the former should be introduced no earlier than the latter. If this ordering is not used (and the opposite order is used instead), application constraints, usually about controlled elements, will be introduced after the details of the controlling elements. This will increase the complexity. This rule can also be identified by making a systematic check because developers find it easy to spot means-end relationships.

#### Merging for Proof

It may be necessary to examine certain properties at the same time in order to discharge application-specific proof obligations about invariants. Obvious cases include a guard condition to ensure an invariant. In the island example, the guard condition of "the number of cars is incremented only if it is less than the capacity" is obviously necessary to prove the invariant "the number of cars is no more than the capacity" This guard-invariant merging is easy for developers to spot (as it is the intention of the guard), and it can be identified in a systematic and manual check. Without this merging, the developer will fail to discharge some of the proof obligations on the constructed models. However, the above case is only one part of the application-specific proof logic. The next section discusses how to handle unobvious cases.

## 5.3 Planning Method

This section describes our method (refinement planner tool and refinement view generator tool) as proof of concept of the rationales presented in Section 5.2.

### 5.3.1 Overview

The overview of our planning method is shown in Figure 5.2.

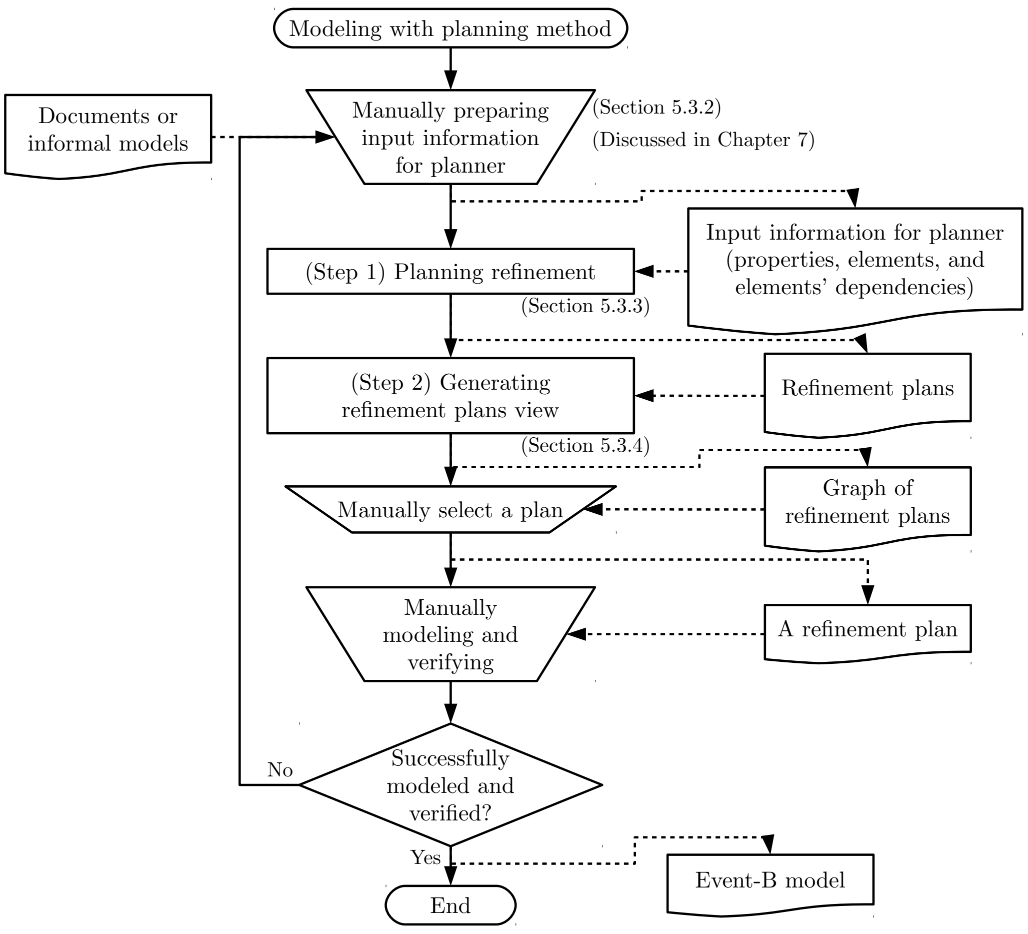

Modeling with planning method
Documents or informal models
Manually preparing input information for planner
(Section 5.3.2)
(Discussed in Chapter 7)
Input information for planner (properties, elements, and elements' dependencies)
(Step 1) Planning refinement
(Section 5.3.3)
(Step 2) Generating refinement plans view
Refinement plans
(Section 5.3.4)
Graph of refinement plans
Manually select a plan
Manually modeling and verifying
A refinement plan
Successfully modeled and verified?
No
Yes
End
Event-B model

Figure 5.2: Overview of using our planning method

Our method requests developers prepare input information of a list of properties, elements that appear in each property, dependencies about static elements for typing, and specification. The feasibility of this non-straightforward process is discussed in Chapter 7.

The refinement planner analyzes the input information and generates possible refinement plans as a list.

Then, the refinement view generator constructs a simplified view of generated refinement plans so that users can easily compare possible refinement plans.

If users are not satisfied with the refinement plans, they may modify the input information, rerun the planner, and rerun view generator again.

Users can follow one of refinement plans shown by the view generator to try constructing models. Although our planner does not guarantee that users can construct valid models by following generated plans, it helps developers to explore the space of possible refinement plans in a systematic and semi-automated manner.

### 5.3.2 Preparing the Input

For refinement planning, the following information is necessary: (1) properties (correspond to invariants), (2) elements, and (3) dependency relationships between elements. To elicit information of properties corresponding to invariants (1), developers need to know what events (i.e., their necessary conditions and effects) change the system's state since invariants are properties that hold before and after an event's occurrence. For elements (2) and dependency relationships between elements (3), developers need to know the elements. Consequently, the process of elicitation is twofold: First, developers acquire general information, namely the events (and their necessary conditions and effects), invariants, and some of the elements. Second, developers consider Event-B specific information, namely the dependency relationships and properties/elements that do not explicitly appear in the normal requirements analysis. This may be facilitated by assuming specific semi-formal specifications, such as constrained natural language, Problem Frames, or UML. However, this thesis does not pick any specific one.

### 5.3.3 Generating Refinement Plans

The previous foundations urge developers to only consider essential rationales for refinement planning. In contrast, actually, there are often high-level, intuitive guidelines that developers have established through experience, as described in [86]. Developers may intuitively merge or order sets of properties. For example, the elements "traffic light on the mainland side" and "one on the island side" seem to be conceptually the same and may suggest a merging rule.

It is also necessary to remind developers of the necessity of merging properties in order to discharge application-specific proof obligations (about invariants). "Merging for Proof" in Section 5.2.2 only showed one of the concrete rules that can be obviously introduced and checked. However, it is a very application-dependent issue as to which set of properties composes a completed proof. This aspect also requires good suggestions from people with experience.

It is desirable but very difficult for developers to give helpful suggestions before the actual modeling and proof trials. As a result, it is necessary to consider an iterative procedure for developers to modify their suggestions after looking at the derived plans or even after trying modeling and proofs. This point also holds for the input properties and elements, which may be wrong or missing. Below, we present a method for refinement planning, especially a search algorithm as it would be if it were a waterfall process. Actually, various modifications are made that may be reflected by rolling back and replanning or by modifying the constructed models (this point will be discussed in the case study).

To realize the core rationale presented in Section 5.2.1 in an efficient way, we implemented a search method that derives only valid plans by constructing a tree of refinement plans. That is, we treat the possibilities of a refinement plan as a tree. The tree has a root node that corresponds to the plan without any steps and leaves that correspond to completed refinement plans (i.e., all of the properties have been introduced). A parent node may have children that correspond to plans with an additional refinement step introducing properties. The method is based on a depth-first search of the tree.

Algorithm 4 describes the core of the search method. The function NEXT-PLANS generates the next steps of a plan. This corresponds to generating the child nodes of a node in a search tree. Let $P_{\text{all}}$ be the set of all properties and $P$ be the set of currently introduced properties. First, an unintroduced property $p \in P_{\text{all}} \setminus P$ is selected (Line 3 in NEXTPLANS) to be introduced in the next step.

**Policy 1: Continue to introduce properties as long as they do not increase the complexity of the model.** If there is another unintroduced property $q \in P_{\text{all}} \setminus (P \cup \{p\})$ such that $\text{dep}(q) \subseteq \bigcup_{x \in (P \cup \{p\})} dep(x)$, introducing $q$ after $p$ does not cause new elements. As we described in Section 5.2.1, a step such as this is "meaningless." Therefore, after selecting an unintroduced property $p$ (Line

**Algorithm 4** Core of our planner tool

```
1:  function NEXTPLANS($plan$)
2:      $next\_plans \leftarrow \emptyset$
3:      for all property $p$ not introduced do
4:          $next\_step \leftarrow \{p\} \cup$ PROPSTOGETHER($plan, p$)
5:          $next\_plan \leftarrow plan +\!\!+ [next\_step]$
6:          if $next\_plan$ does not violate any specified ordering rules then
7:              Add $next\_plan$ to $next\_plans$
8:          end if
9:      end for
10:     return $next\_plans$
11: end function
12:
13: function PROPSTOGETHER($plan, property$)
14:     $together \leftarrow \emptyset$
15:     $new\_plan \leftarrow plan +\!\!+ [\{property\}]$
16:     for all property $p$ not introduced s.t. $p \neq property$ do
17:         $new\_plan' \leftarrow plan +\!\!+ [\{property, p\}]$
18:         if ELEMENTS($new\_plan$) = ELEMENTS($new\_plan'$) then
19:             Add $p$ to $together$
20:         end if
21:     end for
22:     return $together$
23: end function
```

3 in NEXTPLANS), our method selects unintroduced properties $Q$ such that any property in $Q$ does not lead to new elements when it is introduced after $p$ (Lines 16 – 21 in PROPSTOGETHER). Accordingly, the method prevents a meaningless refinement step by introducing $Q$ together with $p$ (Line 4 in NEXTPLANS). The elements that arise by following a plan are determined using a function ELEMENTS . This function determines not only dep($p$) (for all properties $p$ in the plan), but also the elements that should be introduced with it (**Policy 2: Consider elements grouping specifications**).

After deciding the properties to be introduced in the next step (Line 4 in NEXTPLANS), the child nodes of the current node in the search tree are generated only if the elements introduced by following the plan are in a desirable order (Line 6 in NEXTPLANS) (**Policy 3: Search considering the order of introduction of elements**) Specifically, the method checks whether (1) the elements of the systems are introduced before their subsystems and whether (2) controlled elements are introduced before or together with their controller elements.

The policies described above are able to decrease the search space and narrow down the results. Policy 1 decreases the number of properties introduced separately, whereas Policy 2 increases the number of properties introduced by Policy 1. Thus, both policies decrease the depth of the search tree. Policy 3 decreases the width of the search tree since it causes pruning.

In this way, the search proceeds efficiently by generating only plans complying with the core rationale and rules. The search procedure and the checking of the result (Section 5.3.4) can be executed iteratively one after another.

For instance, assume that properties $p_1, p_2, \ldots, p_7$ such that

$$\begin{aligned}
&\mathrm{dep}(p_1) = \{n_{\text{Outside}}, \text{Cap}\},\\
&\mathrm{dep}(p_2) = \{n_{\leftarrow}, n_{\text{Island}}, n_{\rightarrow}, \text{Cap}\},\\
&\mathrm{dep}(p_3) = \{\mathit{ml\_tl}, \mathit{green}, n_{\leftarrow}, n_{\text{Island}}, n_{\rightarrow}, \text{Cap}\},\\
&\mathrm{dep}(p_4) = \{n_{\leftarrow}, n_{\rightarrow}\},\\
&\mathrm{dep}(p_5) = \{\mathit{ml\_tl}, \mathit{green}, n_{\rightarrow}\},\\
&\mathrm{dep}(p_6) = \{\mathit{ml\_tl}, \mathit{green}, n_{\text{Outside}}, \text{Cap}\},\\
&\mathrm{dep}(p_7) = \{n_{\text{Outside}}, n_{\leftarrow}, n_{\text{Island}}, n_{\rightarrow}\},
\end{aligned}$$

and the following ordering rules

**rule1** $n_{\text{Outside}}$ should be introduced earlier than $n_{\leftarrow}$, $n_{\text{Island}}$, and $n_{\rightarrow}$;

**rule2** $n_{\text{Outside}}$ should be introduced together with Cap;

**rule3** $n_{\text{Outside}}$ should be introduced no later than *ml_tl*; and

**rule4** *green* depends on *red*.

are given.

The generated search tree is shown in Figure 5.3.

The root node of the search tree is the plan with no step ([]). NextPlans([]) is calculated as follows. First, one of properties that is not yet introduced is selected ($p$), and the planner sees what the next step will be if $p$ will be introduced in the next step (Lines 3–9 of Algorithm 4). In this example, PropsTogether([], $p_1$) = $\emptyset$, because elements that are required to express properties in the new plan ($\{\text{Elements}\}([\{p_1\}])$) are $\{n_{\text{Outside}}, \text{Cap}\}$, and there are no properties that can be expressed with $\{n_{\text{Outside}}, \text{Cap}\}$ other than $p_1$. Thus, *next_plan* is $[\{p_1\}]$, and this plan does not violate the ordering rules. PropsTogether([], $p_2$) is $\{p_4\}$ because $\{\text{Elements}\}([\{p_2\}]) = \{n_{\leftarrow}, n_{\text{Island}}, n_{\rightarrow}, \text{Cap}\}$, and $p_4$ is also expressible with $\{\text{Elements}\}([\{p_2\}])$ since $\mathrm{dep}(p_4) = \{n_{\leftarrow}, n_{\rightarrow}\}$. However, *next_plan*, in this case, violates order rules 1 and 2 because $n_{\text{Outside}}$ will be introduced after $n_{\leftarrow}$, $n_{\text{Island}}$, $n_{\rightarrow}$ and Cap by following the plan. Therefore, this plan is removed from the stack for search. PropsTogether([], $p_3$) is $\{p_2, p_4, p_5\}$ because $\{\text{Elements}\}([\{p_3\}]) = \{\mathit{ml\_tl}, \mathit{green}, n_{\leftarrow}, n_{\text{Island}}, n_{\rightarrow}, \text{Cap}\}$ (*green* requires *red* because of ordering rule 4), and $p_2$, $p_4$, and $p_5$ are also expressible with $\{\text{Elements}\}([\{p_3\}])$. However, *next_plan*, in this case, is also removed from the stack since it violates order rules 1, 2, and 3.

By repeating such a procedure, the planner finds that NextPlans([]) = $\{[\{p_1\}]]\}$. Thus, the root node of the search tree [] has only one child node ($[\{p_1\}]$).

Next, the planner calculates NextPlans($[\{p_1\}]$).

PropsTogether($[\{p_1\}]$, $p_2$) is $\{p_4, p_7\}$, so *next_plan* is $[\{p_1\}, \{p_2, p_4, p_7\}]$ and valid if $a = p_2$. PropsTogether($[\{p_1\}]$, $p_3$) is $\{p_2, p_4, p_5, p_6, p_7\}$, so *next_plan* is $[\{p_1\}, \{p_3, p_2, p_4, p_5, p_6, p_7\}]$ and valid if $p = p_3$. This plan becomes a leaf node of the search tree since this plan has all properties. By repeating such a procedure, the planner finds that NextPlans($[p_1]$) = $\{[\{p_1\}, \{p_2, p_4, p_7\}]$, $[\{p_1\}, \{p_3, p_2, p_4, p_5, p_6, p_7\}]$, $[\{p_1\}, \{p_4\}]$, $[\{p_1\}, \{p_5, p_6\}]$, $[\{p_1\}, \{p_6\}]$,
$[\{p_1\}, \{p_7, p_2, p_4\}]$ $\}$.

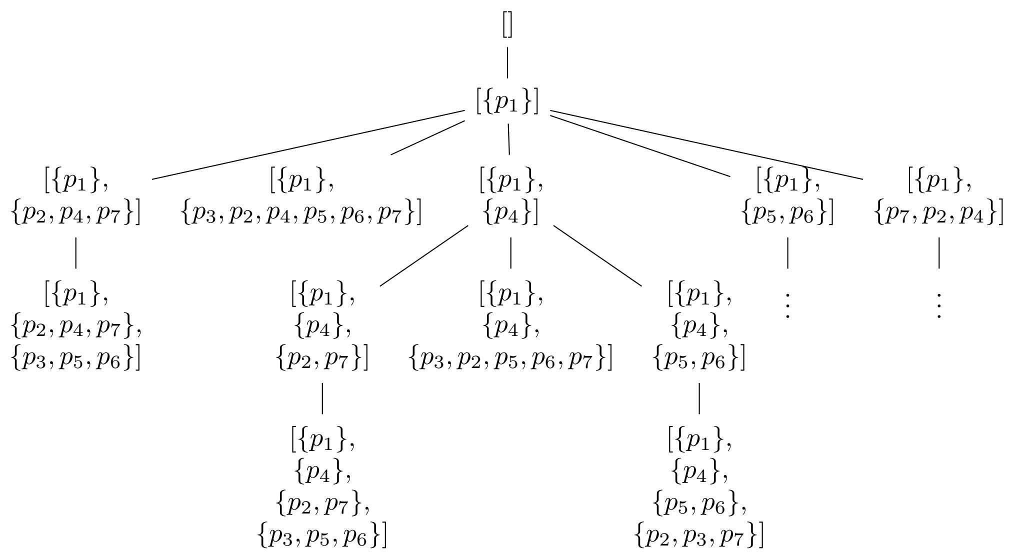

$[]$
$[\{p_1\}]$
$[\{p_1\}, \{p_2, p_4, p_7\}]$
$[\{p_1\}, \{p_3, p_2, p_4, p_5, p_6, p_7\}]$
$[\{p_1\}, \{p_4\}]$
$[\{p_1\}, \{p_5, p_6\}]$
$[\{p_1\}, \{p_7, p_2, p_4\}]$
$[\{p_1\}, \{p_2, p_4, p_7\}, \{p_3, p_5, p_6\}]$
$[\{p_1\}, \{p_4\}, \{p_2, p_7\}]$
$[\{p_1\}, \{p_4\}, \{p_3, p_2, p_5, p_6, p_7\}]$
$[\{p_1\}, \{p_4\}, \{p_5, p_6\}]$
$\vdots$
$\vdots$
$[\{p_1\}, \{p_4\}, \{p_2, p_7\}, \{p_3, p_5, p_6\}]$
$[\{p_1\}, \{p_4\}, \{p_5, p_6\}, \{p_2, p_3, p_7\}]$

Figure 5.3: Example of search tree generated by planner

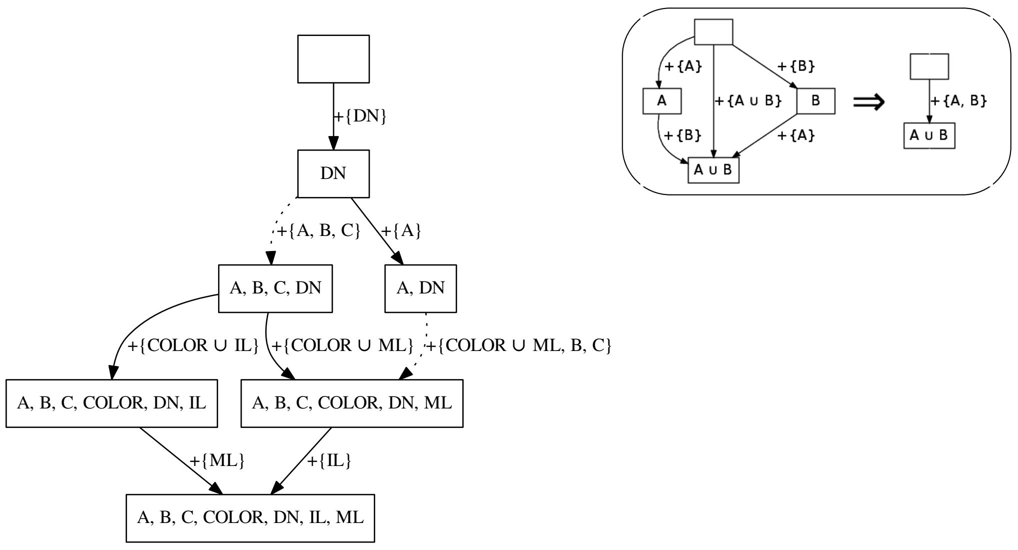


Figure 5.4: Refinement plans view (introduced elements so far) for island example

### 5.3.4 Simplification of Result View

Although the raw result of the search method is composed of sequences of sets of properties, the plans also define elements introduced in each step. The result is shown to users in the elements-introduction order using terms of constituents of the target system The result is shown this way because this form is easier for users to comprehend the model to be constructed, compared to the property-introduction order using terms of properties of the target system. Specifically, the elements are introduced as a directed graph, such as Figure 5.4. The vertices of the graph denote elements introduced so far, and labels on the arcs denote the specific element sets introduced in each step. Thus, the graph has a source that corresponds to no elements and a sink that corresponds to all elements. Moreover, as right-hand side of Figure 5.4 illustrates, the view is presented with a simplification such that the elements are introduced in arbitrary order.

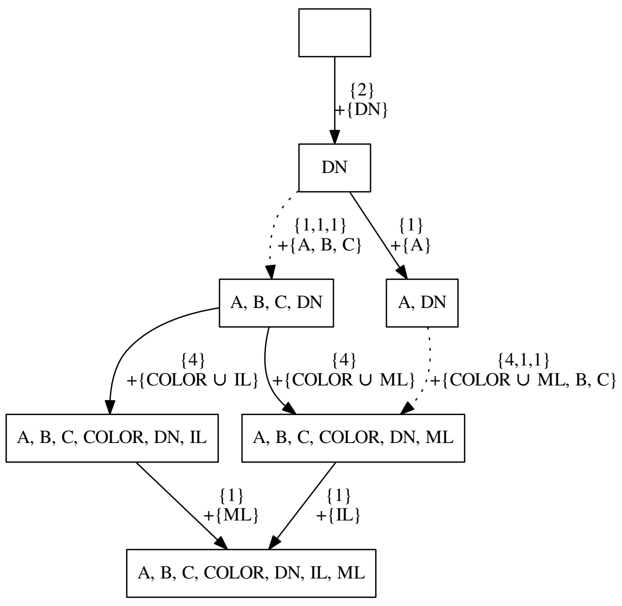


Figure 5.5: Refinement plans view for island example with scores of edges

### 5.3.5 Further Filtering with Heuristics

It is also possible to narrow down generated plans by using heuristics.

One effective heuristic focuses on a primary purpose of the refinement mechanism, namely distributing the complexity of modeling. The heuristic views the number of introduced elements in each step as the complexity of the step.

Heuristics can be used to find "the best" plan in the generated set. For example, in general, modelers want to avoid large refinement steps by effectively using refinement. Therefore, when a set of plans $(p_i)(0 \leq i \leq n)$ is given, it may be helpful to find a plan $p_m(0 \leq m \leq n)$ that minimizes the maximum number of introduced elements in steps of $p_m$.

The index $m$ can be obtained as follows:

$$m = \arg\min_{0 \leq i \leq n}(\max_{1 \leq j < \text{nsteps}(p_i)} |\text{intro}(p_i, j)|),$$

where nsteps$(p)$ is the number of steps of plan $p$.

Another usage of heuristics is to score a plan or some steps of a plan. For instance, Figure 5.5 shows a result view of traffic examples (Figure 5.4) with scores of complexity for every step. Labels on an edge of the diagram show the number of elements introduced in the step. Although reducing complexity is an important aim of using the refinement mechanism, considering only complexity (i.e. finding "the best" plan using the heuristic) may show a too fine-grained plan. Not only reducing complexity but also other points, such as conceptual integrity, are important for refinement planning. Figure 5.5 shows that a modeler may introduce elements of A, B, and C at the same time in the second step. Because A, B, and C are separated but closely related elements, a modeler may find this plan better than the most complexity-reducing plan. Thus, adding this information to the result view is considered helpful.

# Chapter 6

# Case Studies

## 6.1 Evaluation of Refactoring

In this section, we describe two case studies to evaluate our refactoring method described in Chapter 4. In the first case study, we decomposed large refinement steps by using our decomposition method (Section 4.4) to evaluate whether the method is effective for improving maintainability. In the second case study, we aimed to evaluate whether our refactoring method improves reusability of existing models. We first refactored the refinement structure of given machines by using our restructuring method (Section 4.5) and extracted predicates of reusable parts as an abstract machine. We then constructed Event-B machines for another target system as a refinement of the extracted machine.

### 6.1.1 Decomposing Large Refinement Steps

#### Aims and Settings

This case study tried to determine whether we can improve maintainability of existing machines by decomposing refinements.

The author of this thesis decomposed refinements in a large-scale Event-B model with several intermediate machines by following our decomposition method and verified their consistency. The implementation SliceAndMerge was used for slicing (described in Section 4.4.1) in this case study, but it should be noted that SliceAndMerge does not support complementing. Thus, the author used the manual method of finding CPs (described in Section 4.4.2) for complementing intermediate machines.

The target model was a specification about an autonomous satellite flight formation system [78], and it was constructed by a computer scientist who had over four years of experience in modeling in Event-B. The target system was a controller for two spacecraft (leader and follower), which run autonomously while maintaining two-layered communication, namely a higher-layer mode communication and a lower-layer phase communication.

The model has an RC of five steps $[\mathtt{m0}, \mathtt{m1}, \dots, \mathtt{m5}]$. The second refinement ($[\mathtt{m1}, \mathtt{m2}]$) and the third refinement ($[\mathtt{m2}, \mathtt{m3}]$) were selected to be decomposed, because they were larger than the other steps. The row of `m2` in Table 6.1a and the row of `m3` in Table 6.1b show statistics of `m2` and `m3`, respectively. The $N_{\mathrm{V}}$ and $N_{\mathrm{I}}$ [1] in Table 6.1, respectively, list the number of variables and invariants of the models. In `m2`, seven variables and 46 invariants were introduced to specify mode transitions and communications in the spacecraft. In `m3`, two variables have

[1]For the sake of simplicity, we did not count invariants for typing.

Table 6.1: Results of case study 1 of refactoring method

(a) Decomposition of second refinement

| | $N_V$ | $N_I$ | $N_{CP}$ | $N_{UCP}$ | $N_{PO}$ | $N_{MPO}$ |
|---|---|---|---|---|---|---|
| m2 | +7 | 46 | – | – | 454 | 53 |
| m2_1 | +4 | 12 | 21 | 5 | 112 | 12 |
| m2_2 | +1 | 9 | 10 | 6 | 87 | 0 |
| m2_3 | +1 | 8 | 12 | 6 | 80 | 5 |
| m2_4 | +1 | 17 | 0 | 0 | 218 | 33 |
| Sum of m2_* | +7 | 46 | 43 | 17 | 497 | 50 |

(b) Decomposition of third refinement

| | $N_V$ | $N_I$ | $N_{CP}$ | $N_{UCP}$ | $N_{PO}$ | $N_{MPO}$ |
|---|---|---|---|---|---|---|
| m3 | $-2+10$ | 72 | – | – | 1127 | 175 |
| m3_1 | $-1+3$ | 7 | 17 | 4 | 112 | 6 |
| m3_2 | $-1+3$ | 17 | 17 | 8 | 261 | 30 |
| m3_3 | +2 | 14 | 3 | 2 | 202 | 30 |
| m3_4 | +2 | 34 | 0 | 0 | 584 | 81 |
| Sum of m3_* | $-2+10$ | 72 | 37 | 14 | 1159 | 147 |

disappeared, 10 variables were introduced ($N_V$ is "$-2 + 10$"), and 72 invariants were introduced to specify the phase transitions in modes of spacecraft.

**Results**

We selected slicing criteria $V_{B0}$ to obtain the sliced machines. After that, we found CPs with the approach described in Section 4.4.2. Both of the refinements were decomposed with four intermediate machines (m2_1, ..., m2_4, m3_1, ..., and m3_4). Thus, the machines form an RC [m1, m2_1, ..., m2_4, m3_1, ..., m3_4]. The most concrete intermediate machines, m2_4 and m3_4, were semantically the same [2] as the corresponding original machines, m2 and m3. We selected slicing criteria so that the slicing would distribute aspects in the original machines into small and meaningful sets of concepts. For example, the properties and behavior regarding communication failures, the follower's incoming buffer for mode messages, the leader's outgoing buffer, and the acknowledgement message were specified and verified in m2_1, m2_2, m2_3, and m2_4, respectively.

The results of decomposition are shown in Table 6.1. The number of introduced invariants was reduced significantly through the decomposition, and the intermediate machines were more comprehensible than the originals. The replacement of the variables in m3 was also split into two steps. In both m3_1 and m3_2, one variable has disappeared ($N_V$ of both machines is "-1+3").

[2]There were differences in the actual specifications because several invariants were moved in order to abstract the intermediate machines, and the refinement structures of the events were changed.

```
variables:  $cur\_mode\_leader, cur\_mode\_follower, modeDeliveryReport,$
            $cur\_phase\_leader, cur\_phase\_follower,$
            $phaseOutgoingLeader, phaseIncomingLeader,$
            $phaseOutgoingFollower, phaseIncomingFollower$
```

```
inv22:  $phaseOutgoingFollower \neq \emptyset \land cur\_mode\_leader = cur\_mode\_follower$
        $\Rightarrow phaseIncomingLeader = \emptyset$
inv29:  $cur\_phase\_leader = cur\_phase\_follower \land cur\_phase\_leader \neq PHASE0$
        $\Rightarrow cur\_mode\_leader = cur\_mode\_follower$
inv40:  $phaseOutgoingLeader \neq \emptyset \Rightarrow phaseOutgoingFollower = \emptyset$
inv74:  $phaseIncomingLeader = \{P1\} \land modeDeliveryReport = \emptyset$
        $\Rightarrow cur\_phase\_follower = PHASE1$
...
```

```
Event LeavePhase1
  refines RemainCurrentModeLeader
  when
    grd1:  $cur\_phase\_leader = PHASE1$
    grd3:  $modeDeliveryReport = \emptyset$
    grd5:  $phaseIncomingLeader = \{P1\}$
    ...
  then
    act1:  $phaseOutgoingLeader := \{P2\}$
    act2:  $phaseIncomingLeader := \emptyset$
  end
```

Figure 6.1: Specification related to proof of `m3/LeavePhase1/inv40`/INV

We needed to manually find CPs and add them to the intermediate machines except the most concrete ones. Although the author was not familiar with the target model at first, he became familiar with it through finding CPs because the process made implicit properties in concrete machines explicit in simpler vocabularies. The $N_{\mathrm{CP}}$ in Table 6.1 lists the numbers of added CPs. Similar events in $M_{\mathrm{B0}}$ — such as the events of entering phase 1, phase 2, and phase 3 — often had the same kind of inconsistency and thus required the same kind of CPs. The numbers of unique CPs ($N_{\mathrm{UCP}}$) show the actual burden of finding CPs.

The procedure for finding CPs was as follows:

1. We examined original proofs. For instance, Figure 6.1 shows part of the specification of `m3` that is related to a PO `LeavePhase1/inv40`/INV (Figure 6.2). We examined the proof log of the original machine (`m3`) and found that several predicates (`inv22`, `inv29`, `inv74`, `grd1`, `grd3`, `grd5`, and BAP of `act2`) work as hypotheses (Figure 6.3).

2. We found predicates of hypotheses that cannot be written in the intermediate machine. Variables *cur_phase_leader* and *cur_phase_follower*, which are specified in `m3`, were not specified in intermediate machine `m3_2`. Therefore, predicates about the variables are dropped through slicing. `inv29`, `inv74`, and `grd1`, which work as hypotheses in the proof of `LeavePhase1/inv40`/INV, were not specifiable in intermediate machine `m3_2`.

3. We found lemmas in the original proof that are consequences of dropped hypotheses. Figure 6.3 shows that a predicate of a goal GL5 ($cur\_mode\_leader = cur\_mode\_follower$) is a consequence of `inv29`,

| | |
|---|---|
| `inv22` | $phaseOutgoingFollower \neq \emptyset$ $\land cur\_mode\_leader = cur\_mode\_follower$ $\Rightarrow phaseIncomingLeader = \emptyset$ |
| `inv29` | $cur\_phase\_leader = cur\_phase\_follower$ $\land cur\_phase\_leader \neq PHASE0$ $\Rightarrow cur\_mode\_leader = cur\_mode\_follower$ |
| `inv74` | $phaseIncomingLeader = \{P1\} \land modeDeliveryReport = \emptyset$ $\Rightarrow cur\_phase\_follower = PHASE1$ |
| `grd1` | $cur\_phase\_leader = PHASE1$ |
| `grd3` | $modeDeliveryReport = \emptyset$ |
| `grd5` | $phaseIncomingLeader = \{P1\}$ |
| BAP of `act2` | $phaseOutgoingLeader' = \{P2\}$ |
| ... | ... |
| $\vdash$ | $\vdash$ |
| Modified `inv40` | $phaseOutgoingLeader' \neq \emptyset \Rightarrow phaseOutgoingFollower = \emptyset$ |

Figure 6.2: PO `m3/LeavePhase1/inv40`/INV

GL0: $phaseOutgoingLeader' \neq \emptyset \Rightarrow phaseOutgoingFollower{=}\emptyset$ (modified `inv40`)

- GL1: $phaseOutgoingLeader' \neq \emptyset$
  - GL2: $phaseOutgoingLeader'{=}\{P2\}$ (BAP of `act2`)
- GL3: $phaseOutgoingFollower{=}\emptyset$
  - GL4: $phaseOutgoingFollower \neq \emptyset$ $\land cur\_mode\_leader{=}cur\_mode\_follower$ $\Rightarrow phaseIncomingLeader{=}\emptyset$ (`inv22`)
  - GL5: $cur\_mode\_leader{=}cur\_mode\_follower$
    - GL6: $cur\_phase\_leader{=}cur\_phase\_follower$ $\land cur\_phase\_leader \neq PHASE0$ $\Rightarrow cur\_mode\_leader{=}cur\_mode\_follower$ (`inv29`)
    - GL7: $cur\_phase\_leader{=}cur\_phase\_follower{=}PHASE1$
      - GL8: $cur\_phase\_leader{=}PHASE1$ (`grd1`)
      - GL9: $cur\_phase\_follower{=}PHASE1$
        - GL10: $phaseIncomingLeader{=}\{P1\} \land modeDeliveryReport{=}\emptyset$ $\Rightarrow cur\_phase\_follower{=}PHASE1$ (`inv74`)
        - GL11: $phaseIncomingLeader{=}\{P1\}$ (`grd5`)
        - GL12: $modeDeliveryReport{=}\emptyset$ (`grd3`)
  - GL13: $phaseIncomingLeader \neq \emptyset$
    - GL14: $phaseIncomingLeader{=}\{P1\}$ (`grd5`)

Figure 6.3: Proof tree of `m3/LeavePhase1/inv40`/INV

| | |
|---|---|
| `inv29` | $\mathit{cur\_phase\_leader} = \mathit{cur\_phase\_follower}$ $\land \mathit{cur\_phase\_leader} \neq \mathit{PHASE0}$ $\quad\Rightarrow \mathit{cur\_mode\_leader} = \mathit{cur\_mode\_follower}$ |
| `inv74` | $\mathit{phaseIncomingLeader} = \{\mathit{P1}\} \land \mathit{modeDeliveryReport} = \emptyset$ $\quad\Rightarrow \mathit{cur\_phase\_follower} = \mathit{PHASE1}$ |
| `grd1` | $\mathit{cur\_phase\_leader} = \mathit{PHASE1}$ |
| `grd3` | $\mathit{modeDeliveryReport} = \emptyset$ |
| `grd5` | $\mathit{phaseIncomingLeader} = \{\mathit{P1}\}$ |
| ... | ... |
| $\vdash$ | $\vdash$ |
| $\neg$ `inv22` | $\neg\ (\mathit{phaseOutgoingFollower} \neq \emptyset$ $\land \mathit{cur\_mode\_leader} = \mathit{cur\_mode\_follower}$ $\quad\Rightarrow \mathit{phaseIncomingLeader} = \emptyset)$ |
| $\neg$ BAP of `act2` | $\neg\ \mathit{phaseOutgoingLeader}' = \{\mathit{P2}\}$ |
| Modified `inv40` | $\mathit{phaseOutgoingLeader}' \neq \emptyset \Rightarrow \mathit{phaseOutgoingFollower} = \emptyset$ |

Figure 6.4: A sequent inferred from PO `m3/LeavePhase1/inv40`/INV

`inv74`, and `grd1` in the original proof. The predicate is expressible with variables of `m3_2`.

4. We added those lemmas to the machine constructed through slicing. $\mathit{cur\_mode\_leader} = \mathit{cur\_mode\_follower}$ is added to guard of `LeavePhase1` in `m3_2`. We succeeded in discharging `m3_2/LeavePhase1/inv40`/INV with the addition.

This process is just a slight extension of what developers usually do when they try to understand proofs in original machines (step 1). Although we needed to find which expressions of the original machine correspond to particular hypotheses due to lack of traceability support in the proof tree, this process can be easily automated. Thus, we found that this process is easy for modelers, and they need to carry out similar processes in maintenance of the specification, even if they do not use our refactoring method.

We found that CPs could be inferred using Craig's interpolation. We applied inference rules to the sequent of original PO so that the intersection of variables in the antecedent and those in the succedent is a subset of variables in the intermediate machine. A Craig interpolant of such a sequent can be expressed only with variables that are common between the succedent and the antecedent. Although such interpolants are not always minimal, one can be added to the intermediate machine. For example, we applied inference rules to the sequent of `m3/LeavePhase1/inv40`/INV (Figure 6.2) to obtain `m3_2`(e.g., Figure 6.4). In this case, `inv22` and the BAP of `act2` are removed from antecedent and, $\neg$ `inv22` and $\neg$ BAP of `act2` are added to the succedent through the inference. A predicate

$$\mathit{cur\_mode\_leader} = \mathit{cur\_mode\_follower} \land \mathit{phaseIncomingLeader} \neq \emptyset$$

was obtained as an interpolant of the inferred sequent. We added the interpolant as a guard of `LeavePhase1` in `m3_2` to successfully discharge
`m3_2/LeavePhase1/inv40`/INV.
Because the event has a guard $\mathit{phaseIncomingLeader} = \{P1\}$,
a weaker predicate $\mathit{cur\_mode\_leader} = \mathit{cur\_mode\_follower}$ can be added instead of the interpolant.

The $N_{\mathrm{PO}}$ and $N_{\mathrm{MPO}}$ in Table 6.1 respectively list the numbers of all POs and the numbers of POs that were manually discharged, including those POs

**Step 1** ($M_{\text{O1}}$): Persons somehow *move* between locations according to the *authorization of persons to locations.*

**Step 2** ($M_{\text{O2}}$): *Physical connections* between locations are introduced. Persons *move* between physically connected locations.

**Step 3** ($M_{\text{O3}}$): *Doors* with red/*green* lights are introduced. Doors somehow authenticate persons.

**Step 4** ($M_{\text{O4}}$): ID cards are introduced. *Doors* read cards and communicate with a controller by messages to authenticate.

**Step 5** ($M_{\text{O5}}$): Physical movements of *doors*, *persons*, and lights are considered. Communication is a reaction to a physical event.

Figure 6.5: Aspects introduced in each step of original model $\mathbf{M_O}$ [3]

- Persons are in locations but do not move to other locations.
- Locations have monitors and consoles with card readers.
- Authenticated persons log in to the server by inserting their ID cards in a reader.
- A red light indicates an authentication failure.
- The controller tries to find an unoccupied monitor in the room.
- Consoles communicate with a controller by sending messages.

Figure 6.6: Aspects of new model $\mathbf{M_N}$ [3]

related to CPs. Most of POs are usually discharged by automatic provers of the IDE for Event-B. Thus, the number of manually discharged POs ($N_{\text{MPO}}$ in Table 6.1) corresponds to the actual amount of effort for verification. The results show that our decomposition method decreased the labor of verification. For example, rows of `m3` and "sum of `m3_*`" in Table 6.1b show that the number of manually discharged POs decreased from 175 to 147 through decomposition, despite that the number of all POs increased from 1127 to 1159. This appears to be because direct inclusion of CPs added lemmas to the set of hypotheses. Our future work includes a detailed analysis of this effect.

We discuss the possibility of having large-scale refinements and the meaning of the results in Section 8.1.

### 6.1.2 Extracting Reusable Parts of Machines

#### Aims and Settings

This case study tried to determine whether we can extract reusable parts of existing machines by using restructuring (Section 4.5).

We used a model of a "location access controller" (from [2, Chapter 16]) as the original model $\mathbf{M_O}$ with an RC $[M_{\text{O1}}, \ldots, M_{\text{O5}}]$ (Figure 6.5). The model is about a controller of doors between locations according to persons' permission to enter.

We constructed a new model $\mathbf{M_N}$ by reusing parts of $\mathbf{M_O}$. Aspects shown in Figure 6.6 are specified in $\mathbf{M_N}$.

[3] Aspects that should be extracted from $\mathbf{M_O}$ to construct $\mathbf{M_N}$ are underlined and those that should be omitted from $\mathbf{M_O}$ are *slanted.*

First, we constructed a machine $M_{\mathrm{mrg}}$ by merging all the machines of $\mathbf{M_O}$. Next, by slicing $M_{\mathrm{mrg}}$, we extracted aspects that were common to $\mathbf{M_O}$ and $\mathbf{M_N}$. Thus, we extracted specifications related to authentication using communication between card readers and a controller (from $M_{\mathrm{O4}}$ and $M_{\mathrm{O5}}$), persons (from $M_{\mathrm{O1}}$), locations (from $M_{\mathrm{O1}}$), and red lights (from $M_{\mathrm{O3}}$ and $M_{\mathrm{O5}}$). In other words, we omitted aspects that would not be included in $\mathbf{M_N}$. That is, we omitted authorization of persons to locations (from $M_{\mathrm{O1}}$), physical connection of locations (from $M_{\mathrm{O2}}$), doors (from $M_{\mathrm{O3}}$), and green lights (from $M_{\mathrm{O3}}$ and $M_{\mathrm{O5}}$), in addition to movement of persons (from $M_{\mathrm{O1}}$), which is the primary aspect of $\mathbf{M_O}$.

**Results**

As a result, we succeeded in automatically extracting the reusable parts from $M_{\mathrm{mrg}}$ because the reusable machine constructed by slicing was consistent, and thus, we did not need to manually find CPs to make the reusable parts consistent. After that, we successfully augmented the reusable parts with specifications that were unique to $\mathbf{M_N}$. We also succeeded in discharging all POs.

The extracted machine $M_{\mathrm{mrg}}$ describes aspects that are common in $\mathbf{M_O}$ and $\mathbf{M_N}$ (depicted as underlined parts in Figures 6.5 and 6.6). The RC $[M_{\mathrm{mrg}}]$ was decomposed into $[M_{\mathrm{mrg1}}, M_{\mathrm{mrg2}}]$ first.

$M_{\mathrm{mrg1}}$ has the following aspects:

- Location of persons
- Card readers that accepted a person
- List of card readers that did not accept a person
- Blocked card readers (card readers in operation of authentication)
- Messages sent from card readers to the controller when a card is inserted
- Messages sent from the controller to card readers after all operations (acknowledgement messages)

Through refinement, the following aspects were introduced to $M_{\mathrm{mrg1}}$ as $M_{\mathrm{mrg2}}$:

- Messages sent from the controller to card readers if authentication was successful
- Messages sent from the controller to card readers if authentication was not successful
- Card readers that emit red light
- Messages sent from card readers to the controller after red light is turned off

We augmented $M_{\mathrm{mrg1}}$ by adding the following specifications as $M_{\mathrm{N1}}$:

- The controller refers to authentication data of server (specified as a constant relationship) for authentication
- After a successful authentication, the controller eventually sends an acknowledgement message to a card reader

As a refinement of $M_{\mathrm{N1}}$, we constructed another machine $M_{\mathrm{N2}}$. The following aspects related to finding available monitors in the room are introduced through this refinement:

- Persons who are accepted by authorization are in one of the following states:
  - The controller has not yet checked whether there is an available monitor for the person
  - The controller successfully found an available monitor for the person
  - The controller failed to find an available monitor for the person
- If the person's current location has a monitor that is not used by anyone, it is registered to be used by the person
- If an available monitor is found, the person eventually finishes using it
- If no available monitor is found, the process is aborted

Next, as a refinement of $M_{\mathrm{N2}}$, we constructed the most concrete machine $M_{\mathrm{N3}}$ by augmenting $M_{\mathrm{mrg2}}$. The following aspects related to messages displayed on a terminal, and messages sent between card readers and the controller are introduced through this refinement:

- A terminal shows messages ("accepted," "closed," "not found") according to messages from the controller
- A terminal and the controller communicate by sending messages about finding monitors, finishing, and aborting

Figures 6.7 and 6.8 show synchronizations of events in $\mathbf{M_O}$ and $\mathbf{M_N}$, respectively. An event sequence begins when a person inserts a card (CARD event) and ends when a device with a card reader (a door or a terminal) receives an acknowledgement message from the controller (ACKN event). Most of the events involve communication between a device and the controller. As Figures 6.7 and 6.8 show, we succeeded in extracting common parts of events — namely authentication using ID cards, communication between a device and the controller, and an entire event sequence for the case of refusal — by using our restructuring method.

Note that not only omitted aspects in $\mathbf{M_O}$ but also extracted aspects were scattered over several refinement steps in the original specification. Therefore, simply copying a single step, such as $M_{\mathrm{O3}}$, and modifying it is not an effective way of reusing such aspects. In contrast, we succeeded in extracting aspects in a cross-refinement manner by slicing after merging refinement steps.

We discuss the importance of systematic extraction and the feasibility of automatic extraction in Section 8.1.

## 6.2 Evaluation of Planning

In this section, we describe a case study to evaluate our planning method described in Chapter 5. We applied our planning method to four systems (Island, FTP, Location Access Controller, and Storage). The Island example is the example we described in Section 2.1.3. FTP [2] is an example of file transfer protocol with functionalities of re-transmission on error. Location Access Controller [2] is an example of controller that regulates movements of people according to permissions. Storage is an example from industry that controls management, data migration, and load-balancing of a data storage system. We elicited properties, elements, and relationships between elements. Then we examined refinement plans generated by the planner and simplified by the view constructor. Finally we constructed Event-B models according to the plans.

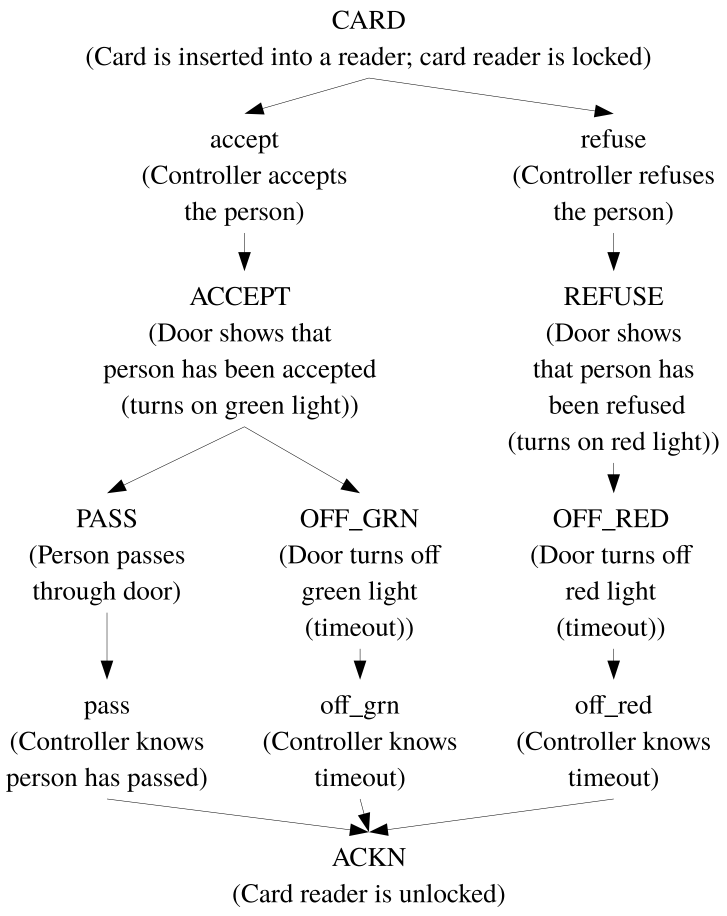

CARD
(Card is inserted into a reader; card reader is locked)
accept
(Controller accepts the person)
refuse
(Controller refuses the person)
ACCEPT
(Door shows that person has been accepted (turns on green light))
REFUSE
(Door shows that person has been refused (turns on red light))
PASS
(Person passes through door)
OFF_GRN
(Door turns off green light (timeout))
OFF_RED
(Door turns off red light (timeout))
pass
(Controller knows person has passed)
off_grn
(Controller knows timeout)
off_red
(Controller knows timeout)
ACKN
(Card reader is unlocked)

Figure 6.7: Synchronization of events of $\mathbf{M_O}$. Events of the controller are named in lower case, whereas events of the environment are named in upper case.

Table 6.2: Results of case study of planning. Numbers of properties, elements, relationships between elements, generated plans, and simplified plans

| Example | $\|P\|$ | $\|E\|$ | $\|P\|/\|E\|$ | $\|Rels\|$ | $\|PL_{\text{raw}}\|$ | $\|PL_{\text{simplified}}\|$ |
|---|---|---|---|---|---|---|
| Island | 16 | 9 | 1.78 | 4, 1, 1 | 63 | 2 |
| FTP | 19 | 18 | 1.06 | 1, 2, 1 | 7206 | 7 |
| LAC | 19 | 23 | 0.83 | 10, 0, 7 | 9564 | 9 |
| Storage | 8 | 14 | 0.57 | 2, 2, 2 | 165 | 54 |

### 6.2.1 Analysis of Refinement Design Space Exploration

The first analysis targeted theoretical understanding of how the planning method explores the design space of refinement. In other words, this trial does not consider realistic situations or practical usages but purely focuses on the characteristics of the planning method. This is done by considering an extreme situation where no smart or experienced human suggestions are given.

By invoking the refinement planner tool, the following data is acquired.

In Table 6.2, Column $|P|$ shows numbers of properties. Columns $|E|$ and $|Rels|$ show the numbers of elements and relationships between elements (corresponding to optional rationales described in Section 5.2.2). By running the search algorithm, 63, 7206, 9564, and 165 plan candidates are derived, respectively (Column

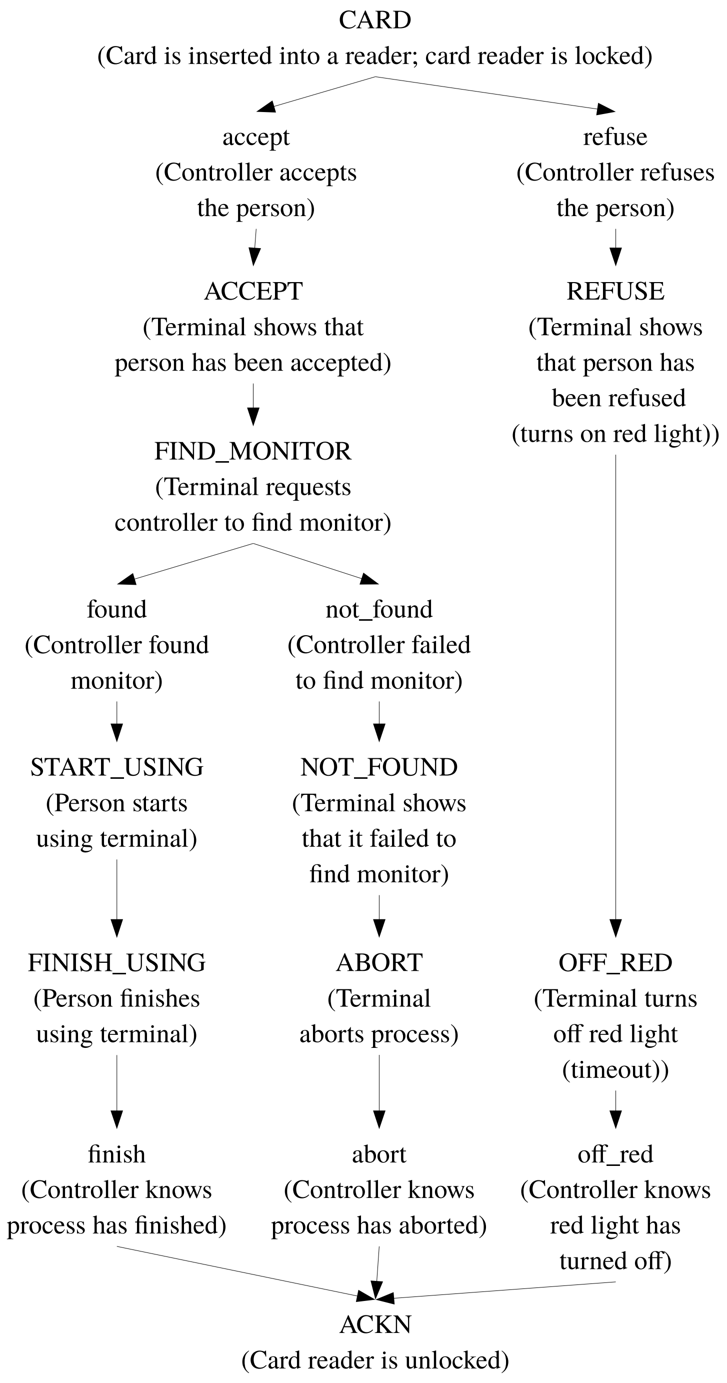

CARD
(Card is inserted into a reader; card reader is locked)
accept
(Controller accepts the person)
refuse
(Controller refuses the person)
ACCEPT
(Terminal shows that person has been accepted)
REFUSE
(Terminal shows that person has been refused (turns on red light))
FIND_MONITOR
(Terminal requests controller to find monitor)
found
(Controller found monitor)
not_found
(Controller failed to find monitor)
START_USING
(Person starts using terminal)
NOT_FOUND
(Terminal shows that it failed to find monitor)
FINISH_USING
(Person finishes using terminal)
ABORT
(Terminal aborts process)
OFF_RED
(Terminal turns off red light (timeout))
finish
(Controller knows process has finished)
abort
(Controller knows process has aborted)
off_red
(Controller knows red light has turned off)
ACKN
(Card reader is unlocked)

Figure 6.8: Synchronization of events of $\mathbf{M_N}$

$|PL_{\text{raw}}|$). This number is not so large given the fact that it includes some arbitrary ordering. Actually, the view generator simplified them into 2, 7, 9, and 54 results, respectively (Column $|PL_{\text{simplified}}|$). The number of simplified plans for the Storage example is larger than others. We estimated this has to do with the ratio of the number of properties and number of elements (Column $|P|/|E|$). Since our planning method follows the dependency between properties and elements, when the dependencies are dense (e.g., Island), the introduction of properties tends to introduce many elements at once. Thus, introduced elements tend to be similar. In contrast, when the dependencies are sparse (e.g., Storage), introduced elements tend to have larger variations.

### 6.2.2 Analysis of Generated Plans

The second analysis looked at the details of generated plans in each system. The results on the Island example show that the space is limited and that the results are similar. Figure 5.4 shows the simplified elements view of the result for the Island example. Figure 5.4 is also simplified by labels to denote a meaningful set of elements, as follows. $N$ denotes a set that includes an element corresponding to the variable of the number of cars outside the mainland. $D$ denotes elements related to constants for a bound of $N$. $A$ and $C$ denote elements related to cars on the bridge heading to the island and to the mainland, respectively. $B$ denotes elements related to cars on the island. $IL$ and $ML$ denote elements related to traffic lights at the island side and the mainland side, respectively. $COLOR$ denotes elements related to the colors of traffic lights.

Figure 5.4 illustrates choices already decided and to be decided. There are some arbitrary choices about merging and ordering of $A, B, C$ and $(COLOR \cup ML), B, C$. Anyway, $N$ is introduced first with $D$ constraining it, as $N$ is a controlled variable used for the global constraint. Regarding traffic lights, either $ML$ or $IL$, the first time it is necessary to talk about $COLOR$. Another point is that the controlling traffic light $IL$ does not precede the controlled $A, B, C$, thus appearing at later steps. The controlling traffic light $ML$ does not precede the controlled $A$ and $N$.

The last point explains the choices of the second step, ordering of $B, C$ and $ML$ (with $COLOR$). The probably natural choice (for human developers) is to consider $A, B, C$ earlier than $ML$ as $A, B, C$ are all about the conceptually same "number of cars" and also compose a complete decomposition of the state of global system $N$. This choice appears in the left-hand side of the second step, which then leaves arbitrary ordering of $IL$ and $ML$ in the remaining steps. Finally, the ordering and merging of $A, B, C$ and $IL, ML$ (with $COLOR$) are up to human developers, which depends on feelings on granularity. It may be natural to consider $A, B, C$ at the same step as $IL, ML$ if not too complex as these two sets represent the same concepts of "number of cars" and "signals." The obtained plan is two steps of $A, B, C$ and $IL, ML$, which is actually the same as the one explained in a textbook of Event-B [2] (Figure 2.1).

The other choices are also possible and valid, expect for the above feeling of "natural." For example, a choice of $A$ and $COLOR \cup ML$ first may be explained as "first refine the interface with the external part while keeping the inside as a blackbox." Though this time the strategy is not natural as the external part (mainland) does not have any state or control. If so, such a strategy allows for multitasking of two parts after first refining the boundary. In this way, the generated plans could be interpreted as different design choices of refinement in the system.

In order to derive plans of the FTP example, we invoked the planner several times while modifying input data iteratively according to the result of previous runs. We tried iterative invocation of the planner by incrementally introducing the rationales while checking that the derived plans (i.e., narrowed design space) were convincing. Even though only the default rationales are used in the case studies, the potentials were demonstrated for interactive and iterative support to make a design choice and see the narrowed design space.

Thus, it was possible to systematically derive refinement plans only with general, primitive rationales to some extent and, to enough extent, with some somewhat-easy human suggestions.

### 6.2.3 Analysis of Input Robustness

There were also trials to delete some of the input properties, elements, or dependencies to represent actual situations where some of the input may be missing. It is difficult to generalize the results clearly, but in many cases, the results are not affected very much. If some dependencies or rules are missing, more plans are derived by the search algorithm. This does not increase the complexity of the result view because the view simplification works well with more arbitrary choices. Thus, an iterative process is practical by finding "strange" plans and thus missing rules. If models are constructed with a meaningless plan derived by incomplete input, at some point, the problem of the input will be found and modified. At this point, replanning may be tried to see what the effect of correction is or just modeling may be continued. The latter is possible as partial correction by adding something to a previous step (if ordering is missing) or merging steps (if merging is missing).

# Chapter 7

# Preliminary Analysis for Finding Information on Abstraction

## 7.1 Problem and Motivation

In Chapter 5, we proposed a method for planning the refinement structure of a target system. The planning method can be used to explore design spaces of a target system in terms of refinement, after documentation or analysis of the target system and before constructing actual models. In Section 6.2, we confirmed that our planning method supports refinement planning for complex problems by analyzing information on dependency, which is given as input.

However, it is not straightforward to prepare input information of our planning method. Although the list of properties, list of elements, and dependency between properties and elements can be easily obtained by documentation or requirements analysis, it requires a special way of thinking to find *information related to refinement*, such as abstract properties, abstract elements, and gluing invariants.

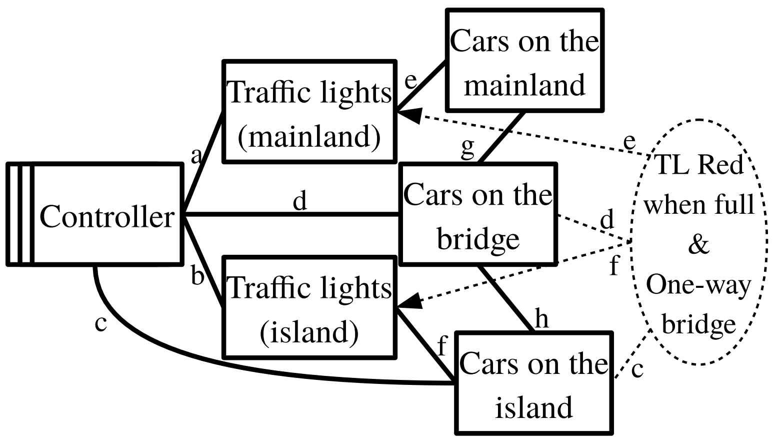
Most existing methods for requirements analysis, such as analysis on natural language document [34] and object-oriented requirements analysis [24], basically analyze target systems in a single abstraction layer and thus are not applicable for our goal. Although goal-oriented requirements analysis methods such as KAOS [27] consider multiple abstractions of goals with and/or decomposition of goals, they do not deal with our target, namely additional information and division of goals for comprehensive abstraction.

In order to discuss the feasibility of finding refinement-specific information, we constructed a preliminary method for elicitation of such information from diagrams of the problem structure of the target system.

Our elicitation method aims to find refinement-specific information by analyzing a diagram for a widely used problem analysis method named Problem Frames [51]. To this end, we defined patterns of common refinements and information related to them by analyzing existing models of Event-B. Our elicitation method helps developers to find refinement-specific information through iterative abstraction of a diagram based on our patterns.

## 7.2 Background on Problem Analysis

Problem Frames [51] is one of the most trusted methods for requirements analysis. In the beginning of the software development process, the approach supports careful analysis of problems about software and its environment. Concretely, phenomena related to a problem are identified and arranged as a problem diagram. The problem diagram is then decomposed into subproblem diagrams, which are



|  |  |
|---|---|
| a: CT!{PulseMTL} | e: TM!{MLRed, MLGreen} |
| b: CT!{PulseITL} | f: TI!{ILRed, ILGreen} |
| c: CI!{#I} | g: CB!{EnterML}, CM!{LeaveML} |
| d: CB!{#L, #R} | h: CB!{EnterIL}, CI!{LeaveIL} |

Figure 7.1: Problem diagram of traffic example

viewed as instances of problem classes (problem frames) and analyzed through argumentation for corresponding problem frames.

A problem diagram of the traffic example is shown in Figure 7.1. The diagram describes the relationship between a *machine domain* (Controller) that represents implementation, *problem domains* (other rectangles) that represent related parts of the world, and *problem requirements* (oval). Labeled lines between domains represent interfaces, which are phenomena shared between domains. A prefix with an exclamation mark of an interface shows a domain that controls the interface. The goal of the analysis is to construct a specification of the machine that satisfies the requirements.

In the Problem Frames approach, the constructed problem diagram is decomposed into *subproblem diagrams*, which are matched to well-known diagram forms (*problem frames*) and analyzed through arguments for the matched frames. Our elicitation method requires a problem diagram as an input, but it does not require the decomposition of it nor argumentation on it.

## 7.3 Requirements Information for Refinement

In this section, we describe refinement-specific information we want to obtain.

### 7.3.1 Gluing Properties

We call some properties *gluing properties*. Gluing properties have both abstract specification elements and concrete specification elements. Thus, gluing properties semantically "glue" together the state spaces of two specifications. For instance, in the traffic example, a specification for the second step (a specification depicted in the middle of Figure 2.1) has a gluing property, such that "the number of cars outside the mainland equals to the sum of the number of cars on the bridge, and the number of cars on the island." This gluing property describes the semantic relationship between elements of the first step (the number of cars outside the mainland) and elements of the second step (the number of cars on the bridge and the number of cars on the island). Gluing properties are important for considering and planning refinements of formal specifications.

### 7.3.2 Abstract Elements

An abstraction of a concept is expressed as a gluing property — namely, as a relationship between the abstract elements and the concrete elements.

Several elements that are not in the concrete natural language document can be introduced for abstraction. For instance, in the traffic example, when elements ("the number of cars on the bridge" and "the number of cars on the island") and a property ("the sum of the number of cars on the bridge and the number of cars on the island is less than or equal to the capacity of outside") are provided in the original document, the developer can *design* a new element ("the number of cars *outside* the mainland") for abstraction of the elements ("the number of cars on the bridge" and "the number of cars on the island"). This allows the developer to verify the property in a simple manner.

Additionally, when considering abstraction, a developer can distinguish abstract elements from concrete ones by deciding the *introduction order of elements*. Because abstract elements are simpler than concrete ones, abstract elements should be introduced earlier for simplicity of modeling and verification. For example, "the number of cars outside the mainland" should be introduced earlier than "the number of cars on the bridge" and "the number of cars on the island" since "the number of cars outside the mainland" is an abstraction of the two elements.

### 7.3.3 Abstract Properties

Non-gluing properties in a concrete specification may be rephrased or weakened to their abstract versions by using abstract elements in an abstract specification. For instance, in the traffic example, the property $p$ ("the sum of the number of cars on the bridge and the number of cars on the island is less than or equal to the capacity of outside") is equivalent to its abstract version $p'$ ("the number of cars outside the mainland is less than or equal to the capacity of outside"), according to the gluing property $p_g$ ("the sum of the number of cars on the bridge and the number of cars on the island is equal to the number of cars outside the mainland"). By rephrasing $p$ to an equivalent or weakened property $p'$, the developer can verify $p$ in an abstract specification by using only abstract elements.

### 7.3.4 Abstraction Fragment

We call a pair consisting of a gluing property and the ordering rules of elements an *abstraction pair*. For instance, a pair $PA$ $\langle$ "The sum of the number of cars on the bridge and the number of cars on the island is equal to the number of cars outside the mainland," "the number of cars outside the mainland" should be introduced earlier than "the number of cars on the bridge" and "the number of cars on the island" $\rangle$ is an abstraction pair. In our elicitation methods, an abstraction pair can be obtained as an instance of information in patterns.

A refinement chain of formal specification is planned by arranging several abstraction pairs. In other words, the order of consideration of the abstraction pairs decides the refinement chain.

Given these facts, to consider refinement of a specification from flat descriptions of the concrete properties of a target system, developers need to consider the possible abstractions of the given properties. Our approach uses abstraction

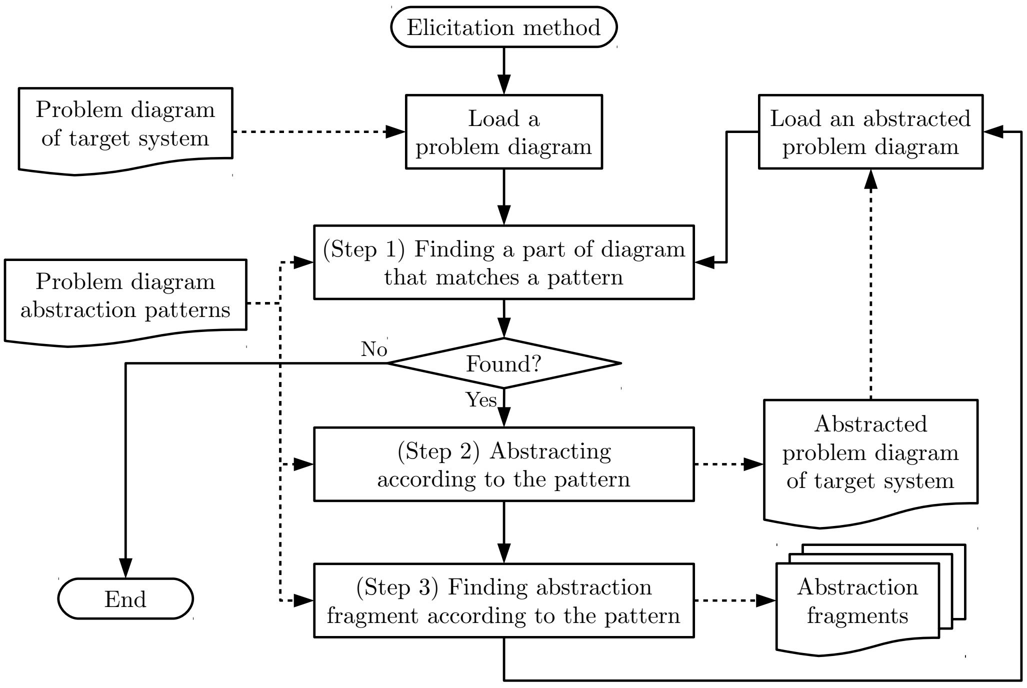

Elicitation method
Problem diagram of target system
Load a problem diagram
Load an abstracted problem diagram
(Step 1) Finding a part of diagram that matches a pattern
Problem diagram abstraction patterns
No
Found?
Yes
Abstracted problem diagram of target system
(Step 2) Abstracting according to the pattern
End
(Step 3) Finding abstraction fragment according to the pattern
Abstraction fragments

Figure 7.2: Overview of our elicitation method

pairs for this process. For example, assume that we can elicit the property "when the traffic light is green, the number of cars on the island is greater than zero" in the concrete document. This property can be rephrased to "when the traffic light is green, the number of cars outside the mainland is greater than zero." using the abstraction pair $PA$. When creating a specification using refinement, abstract properties should be specified in an abstract specification. Therefore, after getting concrete properties and abstraction pairs, a developer needs to acquire their abstract properties.

We call a pair of information described in Section 7.3 *abstraction fragment.* Namely, an abstraction fragment consists of an abstraction pair, abstract properties, and abstract elements (acquired through finding the introduction order of elements).

## 7.4 Abstraction of Problem Diagrams Using Patterns

The process of our elicitation method is shown in Figure 7.2.

First, developers need to construct a problem diagram of Problem Frames. In this analysis process, developers can elicit properties and elements of target systems.

This elicitation method stepwisely constructs abstract versions of a given diagram by using patterns of modification on problem diagrams. New information related to an abstraction is obtained in this process.

This step should be repeated until there is no part of the diagram that can be a target of abstraction. An example of stepwise abstraction is shown in Figure 7.3. The concrete diagram is identical to one shown in Section 7.1. The intermediate diagram shows that the controller directly informs cars about the number of cars in every area in the intermediate problem. In the abstract diagram, $n_{\text{Island}}$, $n_{\leftarrow}$, and $n_{\rightarrow}$ are grouped as "the number of cars outside the mainland" ($n_{\text{Outside}}$).

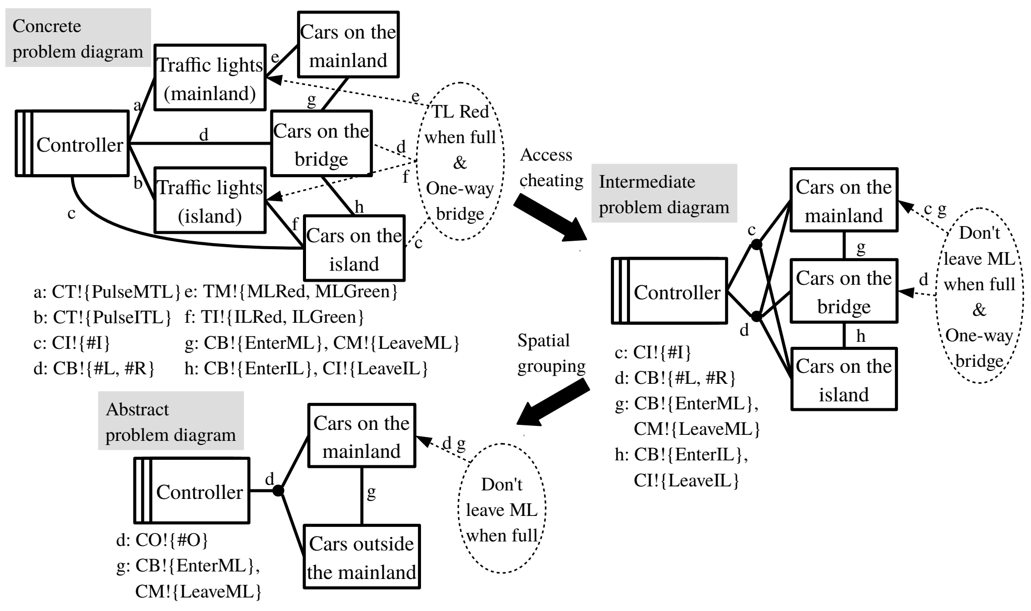


Figure 7.3: Stepwise abstraction of problem diagram

A pattern comprises a target, concrete elements, abstract elements, gluing concerns, and diagram modifications. *Target* describes parts in a concrete problem diagram that can be changed through an abstraction using the pattern. Modifications on a problem diagram using the pattern are described in terms of elements, namely domains and interfaces. Through the abstraction, developers omit *concrete elements* and add *abstract elements*. Relationships between abstract elements and concrete elements are described in *gluing concerns*, which are instantiated as gluing descriptions. Modifications on a problem diagram through the abstraction are described in *diagram modifications*. Developers need to convert requirements that refer to concrete elements into abstract requirements by considering gluing descriptions. Abstract requirements can be equivalent or weaker than the concrete versions, but cannot be stronger than the concrete versions.

For example, a pattern named *spatial grouping* is defined as follows:

**Target** Several domains of the same class.

**Concrete elements** Domains in the target and their interfaces.

**Abstract elements** A domain that represents a combination of the domains and interfaces that represent combinations of concrete interfaces of the same class.

**Gluing concerns** For every abstract element $a$ that represents combinations of concrete element $(c_i)$, the value of $a$ is equal to the (numerical, set-theoretical, etc.) sum of values of $(c_i)$.

**Diagram modifications** Concrete elements are replaced with abstract elements. Domains that have interfaces with concrete domains come to have interfaces with the abstract domain.

An example of applying the spatial grouping pattern is illustrated as abstraction of the intermediate problem diagram in Figure 7.3. In this case, first, the domains "Cars on the bridge"and "Cars on the island"are found as the target because they are about the same class. Then, the target domains are replaced with an abstract domain ("Cars outside the mainland"). Moreover, concrete elements of the same class $n_{\text{Island}}$, $n_{\leftarrow}$, and $n_{\rightarrow}$ are grouped as an abstract element ($n_{\text{Outside}}$). As an instantiation of a gluing concern, developers obtain a gluing description ($n_{\text{Outside}} = n_{\text{Island}} + n_{\leftarrow} + n_{\rightarrow}$). Concrete elements EnterIL and LeaveIL are just omitted because they are shared within the target domains. Requirement $n_{\text{Island}} + n_{\leftarrow} + n_{\rightarrow} \leq \text{Cap}$ is changed to its abstract version ($n_{\text{Outside}} \leq \text{Cap}$) by considering the gluing description. As a result, an abstract element, an abstract property, and a gluing property are obtained.

We analyzed 14 problems that comprise safety properties as their key properties and defined five patterns. We describe the other patterns in the rest of this section.

### 7.4.1 Temporal Grouping

This pattern is for grouping multiple sequential event elements as a single event element. Assuming that a problem is related to a sequence of consecutive events $(e_0, \ldots, e_{n+1})$ , events $\{e_1, e_2, \ldots, e_n\}$ are considered as concrete elements and replaced with an abstract event element ($e'$). Thus the sequence becomes $(e_0, e', e_{n+1})$. Gluing concerns are (1) $e'$'s guard is equivalent to the guard of $e_1$, and (2) $e'$'s state change is equivalent to state change by a sequence of events $(e_1, \ldots, e_n)$. States that are changed only by event $e_1, e_2, \ldots$, and $e_n$ are omitted in the abstract problem.

For instance, a possible scenario of cars' movement in the example is as follows:

1. The controller sends a pulse to a traffic light on the mainland (MPulseG).
2. The traffic light turns green (MTLColor := green).
3. Cars on the mainland go on the bridge (LeaveMainland / #L++).
4. The cars on the bridge (going left) eventually enter the island because the bridge is controlled to be one way (EnterIsland / #L - -; #I++).
5. Steps 3 and 4 are repeated until #L becomes zero.
6. To allow cars on the island returning, the controller sends a pulse to the traffic light on the mainland to turn it red (MPulseR).

Developers can design an event (EnterIslandDirectly / #I becomes positive; #L becomes zero) by applying the temporal grouping pattern to a concrete problem with this scenario. Thus, in the abstract problem, the scenario is represented by a consecutive sequence of events (MPulseG, EnterIslandDirectly, MPulseR). In this way, the pattern makes argumentation simpler.

### 7.4.2 Access Cheating

In problem analysis, developers frequently need to analyze a domain $C$ that connects other domains $A$ and $B$. Typical examples are sensors and actuators that connect the machine and the world. Such a domain $C$ is called *connection domain* and considered important since it may be related to requirements and effects, including delay, unreliability, and conversion of data [51].

The access cheating pattern aims to construct an abstract problem without a connection domain and arguments over properties of connection domains later (i.e., in concrete problems). Thus, in an abstract problem, permission of logical access or physical access is intentionally violated, and developers concentrate on effect without considering means. The gluing concern of this pattern is *"Property of means $\Rightarrow$ effect"* or *"Property of means $\Leftrightarrow$ effect."* For example, the intermediate problem in Figure 7.3 can be abstracted from a concrete problem using this pattern.

For instance, in our example, the controller of the concrete problem needs to regulate cars' traffic by controlling traffic lights. Controlling traffic lights corresponds to informing a number of cars in all areas because the color of traffic lights depends on the number of cars (e.g., a green traffic light on the mainland means that $n_{\rightarrow} = 0 \land n_{\text{Island}} < \text{Cap}$). By applying this pattern to the example, developers can construct an abstract problem such that the controller directly informs cars about the number of cars. In this case, the gluing concern is "traffic light is green $\Leftrightarrow n_{\rightarrow} = 0 \land n_{\text{Island}} < \text{Cap}$."

### 7.4.3 State Transition Limiters

In this pattern, developers find additional elements that limit the transitions of some states. If an element represents a state (variable) and has a specific domain or constraint, the element should be introduced together with the elements that represent the domain. In the traffic system, "the number of cars on the bridge" and "the number of cars on the island" should be introduced together with its upper bound ("capacity"). If this merging is not used, the refinement steps seem to be too fine-grained to delay introducing constraints on one variable. This rule is sufficiently concrete so that it does not require suggestions from experienced persons. It only requires a systematic check of state elements, and the domains or constraints on each of them. Although it might be acceptable to introduce the state first, constructing an abstract specification that the state changes freely (more than the actual), then later the constraint, is introduced. The other ordering is meaningless as then the first specification includes the constraining elements (capacity) not used for anything. For instance, in the traffic system, the transition of state "the number of cars outside the mainland" is limited by a constant "capacity outside the mainland."

### 7.4.4 Realization of Properties

Another common strategy is to start from specifications of the logical properties and behaviors of the system and to introduce specifications of the means afterward. In the traffic system example, the specifications of the properties and behaviors of the system are only about the number of cars in the abstract model. The concrete models introduce the properties and behaviors of traffic lights, and they introduce sensors. One of the gluing properties for the number of cars and the state of the traffic light is as follows: "If traffic light is green, the number of cars outside is less than capacity." Gluing properties for this pattern have the form "(*property of means* `implies` (*logical property*)" or "(*property of means*) `iff` (*logical property*)."

### 7.4.5 Other Patterns

In addition to generic patterns provided by us, a developer can reflect ad-hoc guidelines from experiences in this elicitation method. For instance, assume that there is a guideline that recommends developers introduce sensors earlier than actuators. That guideline can be reflected as a pattern for the elimination of actuators from a problem diagram that includes sensors and actuators.

Table 7.1: Target systems in user experiment

| Target system | #properties | #elements | #orders |
|---|---|---|---|
| Traffic | 16 | 10 | 15 |
| LAC | 19 | 23 | 54 |
| Storage | 8 | 19 | 22 |

## 7.5 Preliminary Experiment on Finding Abstraction Fragments

In this section, experiments to evaluate our research questions are described. Problems used in experiments are Traffic (described in Section 1.1), LAC [2, Section 16] (a hardware/software system to regulate movement of people between rooms considering authorization), and Storage (from an industrial project, a controller of storage systems considering load balancing and virtual storage), as Table 7.1 shows.

We carried out a user experiment with the following settings.

In the first experiment, we let users elicit abstraction fragments by using our elicitation method of pattern-based analysis. This experiment serves as a preliminary phase of the experiment of refinement planning (Section 6.2).

We briefly described our elicitation methods and target systems to participants, and we gave participants concrete problem diagrams of the target systems. The concrete problem diagrams were constructed by us so that participants could concentrate on our elicitation methods without being perplexed by details of the Problem Frames approach. Participants followed our elicitation method (Section 7.4) to derive abstract problem diagrams until they derived the most abstract problem diagram. They also found gluing descriptions as instantiations of gluing concerns during the derivation.

The participants were four computer science students without knowledge of Problem Frames. We assumed they could not supplement our elicitation methods with their insights because of their lack of experience.

Participants applied patterns to problem diagrams without being puzzled and acquired abstract problem diagrams and gluing descriptions of all target systems in less than one hour in total. For each of the analysis results by the four participants, a refinement plan was derived by using the planner. Table 7.2 summarizes how the complexity is mitigated into multiple steps. #steps and #maxphen denote the number of steps and the maximum number of introduced elements, respectively, in each step. All the participants succeeded in mitigating the complexity at a similar level of granularity of the steps.

As a second experiment, we succeeded in constructing Event-B models in multiple abstraction levels in all cases by using the information acquired in the first experiment. Table 7.3 shows the number of invariants and number of variables in every step of every case. We also successfully showed that arguments over each model are consistent and that concrete arguments are consistent with their abstract versions.

Table 7.2: Refinement plans comparison

| Plan | $P$ | $P_1$ | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|
| #steps (Traffic) | 3 | 3 | 3 | 3 | 3 |
| #steps (LAC) | 5 | 6 | 5 | 3 | 4 |
| #steps (Storage) | 3 | 3 | 2 | 3 | 2 |
| #maxphen (Traffic) | 3 | 3 | 3 | 4 | 3 |
| #maxphen (LAC) | 5 | 5 | 5 | 7 | 6 |
| #maxphen (Storage) | 4 | 4 | 5 | 4 | 4 |

Table 7.3: Number of requirement clauses and elements in Experiment 2 of elicitation method

| Target | Traffic | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Sort | req | | | | phen | | | |
| Participant | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
| Abst. problem | 1 | 1 | 3 | 1 | 3 | 3 | 4 | 3 |
| 1st refinement | 4 | 4 | 7 | 4 | 7 | 7 | 6 | 7 |
| 2nd refinement | 7 | 7 | 7 | 7 | 11 | 11 | 10 | 11 |
| 3rd refinement | - | - | 7 | - | - | - | 11 | - |
| Target | LAC | | | | | | | |
| Sort | req | | | | phen | | | |
| Participant | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
| Abst. problem | 2 | 2 | 5 | 1 | 3 | 3 | 8 | 5 |
| 1st refinement | 3 | 3 | 5 | 5 | 6 | 6 | 10 | 10 |
| 2nd refinement | 5 | 7 | 7 | 7 | 10 | 18 | 18 | 18 |
| 3rd refinement | 7 | - | - | - | 18 | - | - | - |
| Target | Storage | | | | | | | |
| Sort | req | | | | phen | | | |
| Participant | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
| Abst. problem | 4 | 4 | 4 | 1 | 8 | 7 | 8 | 3 |
| 1st refinement | 9 | 7 | 9 | 7 | 17 | 12 | 17 | 12 |
| 2nd refinement | - | 9 | - | 9 | - | 17 | - | 17 |
| 3rd refinement | - | - | - | - | - | - | - | - |

# Chapter 8

# Discussion

## 8.1 Discussion on Refactoring

In this section, we discuss our refactoring method (Chapter 4) and application of the method considering the results of the case study for the method (Section 6.1).

### 8.1.1 Discussion on Refactoring Methods

#### Deriving CPs

All POs originate from specifications. Hypotheses essential to discharging POs are also inferred from specifications. We call predicates that raise a PO $\phi$ *raisers* of $\phi$ and predicates that provide hypotheses for discharging $\phi$ *hypothesis providers* of $\phi$.

Suppose that $\phi$ is a PO in a concrete machine. If the raisers of $\phi$ are expressible by $V_{\mathrm{B}}$, the hypotheses required to discharge $\phi$ should also be expressible by $V_{\mathrm{B}}$. However, hypotheses providers are not always specified with vocabulary of $V_{\mathrm{B}}$. Sometimes, a PO $\phi$ that is expressible by $V_{\mathrm{B}}$ is discharged with hypotheses including a hypothesis $h$ that is expressible by $V_{\mathrm{B}}$, and $h$ is implied by hypotheses providers $P$ that are expressible by $V_{\mathrm{C}}$ but not expressible by $V_{\mathrm{B}}$. In other words, in this case, $h$ is not directly specified in the machine but rather implicitly specified by $P$. In such cases, $\phi$ is raised but cannot be discharged in the intermediate machine since the intermediate machine lacks some of the hypotheses providers for $\phi$. Thus, users need to add CPs that are expressible by $V_{\mathrm{B}}$ and able to imply hypothesis $h$.

However, developers tend to directly specify hypotheses in practice, because hypotheses raisers for POs are usually important properties of a target system; thus, directly specifying hypotheses to discharge the POs is usually a meaningful way of describing the system. Therefore, users do not need to add CPs frequently. For instance, we did not need to add CPs in the second case study (Section 6.1.2), because all of the hypotheses providers were specified in $V_{\mathrm{B}}$ for all of the POs that were expressible by $V_{\mathrm{B}}$.

Specifying a hypothesis provider in the form $h \wedge \mathit{predicate}$ to imply hypothesis $h$ is another common case. Although users need to add CPs, they can be found with simple rules. In other cases, CPs can be found by reviewing the proofs for the original machines, as described in Section 4.4.2. This task is easy for users who are familiar with Event-B. In our first case study (Section 6.1.1), all of the CPs were found by conducting a rule-based analysis of the concrete machine or analyzing the proof of the machine's consistency.

Therefore, we conclude that finding CPs is neither frequently required nor difficult. As a primary part of our future work, however, we are planning to construct systematic and complete methods for deriving CPs so that developers can easily derive consistent intermediate machines. We will further investigate relationships between CPs and Craig interpolation of the completed proof.

**Selecting Slicing Criteria**

Users of our decomposition method can select a slicing criterion, namely variables that are specified in the intermediate machine. Users may consider aspects of the intermediate machine and select some of the variables of the concrete machine, or they may consider properties that should be verified in the intermediate machine and select some of the invariants of the concrete machine. In the latter case, the slicing criterion is a set of variables required to specify selected invariants. Users can select an arbitrary $V_{\mathrm{B}0}$ so long as $V_{\mathrm{A}} \cap V_{\mathrm{C}} \subseteq V_{\mathrm{B}} \subseteq V_{\mathrm{A}} \cup V_{\mathrm{C}}$.

**Feasibility of Consistent Decomposition of Refinement**

In Section 4.4.2, we described a heuristic for mending consistency by finding CPs as Craig interpolants of formulas that are equivalent to POs in given machines $M_{\mathrm{A}}$ and $M_{\mathrm{C}}$. Obviously, the applicability of this heuristic depends on the applicability of interpolation to formulas of POs.

From the point of view of languages used in expressions, we cannot claim that the heuristic is applicable to arbitrary predicates in Event-B models. The mathematical language of Event-B has a high expressive power because it is first-order logic extended with set theoretic notation that enables advanced expressions, such as quantification over all subsets of a set [4]. In addition, the language can be further extended with additional operators and proof rules by using the theory extension mechanism [19]. The theory extension has been used to support advanced mathematical theories, such as those of real numbers and graphs, on Event-B models. Considering these facts, we cannot guarantee that the heuristic is applicable to every Event-B model.

However, the heuristic is applicable to predicates if they are reducible to first-order ones because the interpolation theorem holds for first-order logic, and such a class of predicates is expressive enough for most practical use. For example, as we described in Section 6.1.1, the expressions of the models used in case studies of refactoring were in such a class. Most of the set-theoretic notations used in the models are basic ones, such as $\mathit{phaseOutgoingFollower} \neq \emptyset$ and $\mathit{phaseIncomingLeader} = \{\mathit{P1}\}$. In the models, there is also a predicate with quantification over subsets of sets. It is a predicate of an invariant that states several variables are singletons, as follows:

$$\begin{aligned}
&\forall S\, \forall s\,.\\
&S \in \{\mathit{phaseOutgoingLeader}, \mathit{phaseOutgoingFollower},\\
&\mathit{phaseIncomingLeader}, \mathit{phaseIncomingFollower},\\
&\mathit{phaseDeliveryReportLeader}, \mathit{phaseDeliveryReportFollower}\}\\
&\land S \neq \emptyset \land s \in S\\
&\Rightarrow S = \{s\},
\end{aligned}$$

where *phaseOutgoingLeader* and other variables about phases are a subset of a finite set of constants. However, it is equivalent to the following:

$$\begin{aligned}
&\forall s\,.\mathit{phaseOutgoingLeader} \neq \emptyset \land s \in \mathit{phaseOutgoingLeader}\\
&\ \Rightarrow \mathit{phaseOutgoingLeader} = \{s\}\\
&\land\\
&\forall s\,.\mathit{phaseOutgoingFollower} \neq \emptyset \land s \in \mathit{phaseOutgoingFollower}\\
&\ \Rightarrow \mathit{phaseOutgoingFollower} = \{s\}\\
&\land\\
&\ \ldots\\
&\land\\
&\forall s\,.\mathit{phaseDeliveryReportFollower} \neq \emptyset \land s \in \mathit{phaseDeliveryReportFollower}\\
&\ \Rightarrow \mathit{phaseDeliveryReportFollower} = \{s\}
\end{aligned}$$

Thus, it is reducible to a first-order expression.

Therefore, although developers can specify highly mathematical predicates for developments, the heuristic is considered to be applicable for most practical uses of Event-B.

### Adding New Concepts of Abstraction

A user can add new concepts of abstraction to the machines, by decomposing refinement after adding new specifications for abstraction to the concrete machine.

One way is adding new variables. For example, in Figure 2.5, by creating an intermediate machine that has $\{g\}$ as $V_{\mathrm{B0}}$ after adding a variable $g$, an invariant $g = a + e$, and other predicates, a user can construct an intermediate machine for specification of variables $b$ and $g$ instead of variables $b$, $a$, and $e$.

The other way is adding new events. Assume that a concrete machine has several events $E$ that have common guards and actions. By selecting variables that occur in common predicates in $E$ as $V_{\mathrm{B0}}$, a user can construct an intermediate machine with an abstract event, which is refined by all events of $E$.

These appear to be useful for restructuring refinement of existing models.

## 8.1.2 Discussion on Application of Refactoring

### Further Elaboration of Intermediate Machines

After slicing and complementing, users may want to add properties to a consistent intermediate machine, in order to make it richer. In this case, users need to find concrete machine properties and abstract them so that they are expressible by $V_{\mathrm{B}}$.

### Improvement of Maintainability by Decomposition

In our first case study (Section 6.1.1), we decomposed large refinements into smaller ones. The primary benefit of reducing the size of specifications is the support of maintaining machines. According to a study conducted in industries [76], activities for formal specifications' maintenance include impact analysis, refactoring identification, and validation. Our decomposition method makes such activities easier because it shrinks the size of the state space and the number of predicates, and reveals implicit properties of concrete machines as CPs. In particular, reducing the size of specifications can significantly reduce the cost of verification [61] in maintenance. The study [76] also reports that refactoring steps should be small. This becomes easier with smaller refinement steps. Thus, our decomposition method improves maintainability of each single refinement step. Our future work will include evaluation of the trade-off between this and the maintainability of the whole model.

It should be noted that the improvement of understandability and the reduction of the cost of manual proofs confirmed in the case study is a result of finding CPs with a manual analysis by the author. Thus, we did not confirm whether our refactoring method with the heuristic with interpolation is effective for improving the maintainability. Confirmation of this will be included in our primary future work.

### Large Refinement Steps

Large refinement steps such as ones used in our first case study (Section 6.1.1), are common. Developers design refinements on the basis of properties that should be verified or subjects that should be considered in each step. Usually, such properties or subjects are about multiple aspects of the target system. Therefore, including many aspects in one refinement step may seem natural for developers when they construct machines and are in fact common, despite that smaller refinements are easier to comprehend. Thus, we believe our decomposition method is effective for most existing Event-B machines.

### Effectiveness of Systematic Extraction of Reusable Parts

In our second case study (Section 6.1.2), we automatically extracted reusable parts of an existing model. Manually extracting such parts without using our refactoring method is not impossible — namely, developers can extract such parts by examining several machines of the original model and copying and pasting. However, the number of predicates that should be examined is large. In addition, such predicates are usually scattered over several machines. Therefore, manual examination is tedious and error-prone. Our refactoring method makes this process more systematic. Moreover, if a machine constructed with slicing is consistent (i.e., there is no need to find CPs), it is possible to automatically extract reusable parts. If the implementation of heuristics described in Section 4.4.2 is possible, it will further help this process of extracting reusable parts.

It should be noted that even if heuristics for finding CPs (Section 4.4.2) can be implemented, CPs that are found by such heuristics may not be suitable for reuse because they may not be natural for humans. We need to check whether such heuristics are effective to obtain comprehensible and reusable parts of machines by implementing them. It will be interesting to exploit sophisticated interpolation algorithms tailored to obtain simple interpolants, such as one proposed in [8].

### Feasibility of Automatic Extraction of Reusable Parts

In our second case study (Section 6.1.2), we extracted aspects of "authentication using communication between card readers and a controller" as reusable parts of the original machines. In the original machines, these aspects were introduced through several refinement steps, and it seemed that they were dependent on other parts. However, they were actually independent of other parts, and we succeeded in automatically extracting them. We often see this kind of independence of parts embedded in machines. Our refactoring method provides an automatic extraction of such parts. Although users sometimes need to add CPs, most of the predicates can be found with rules, as we described in this section.

## 8.2 Discussion on Refinement Planning

In this section, we discuss the capability and potential of our planning method (Chapter 5). On the basis of the results of the case study for the method (Section 6.2), we discuss the contribution to the activity of refinement design and whether our planning method can be a foundation of tools for planning refinement in Event-B.

### 8.2.1 What Roles in Refinement Design Can the Planning Method Play?

As the planning method offers a kind of filtering function, primary concerns are about what and how much the planning method excludes. Regarding this point, the approach in this thesis adopted only part of essential and general rationales to eliminate plans that seem absolutely meaningless. In other words, this thesis avoided using more high-level, intuitive guidelines, which is possibly controversial, without logical validation.

#### Does the Planner Eliminate Only Meaningless Refinement Plans?

We defined "validity" of the refinement structure and took an approach to eliminate clearly meaningless plans from possibilities of arbitrary refinement plans. Thus, we consider this question as equivalent to whether our definition of validity is valid.

As described in Section 5.2, our definition of validity is in a broad sense; we consider not only mathematical correctness and consistency but also generic refinement strategies based on the intention of refinement. For example, as described in Section 5.2.2, our planning method considers that controlling elements (e.g., elements of traffic lights) are more concrete than controlled elements (e.g., number of cars). Therefore, arguing validity of the refinement structure is controversial. Although we consider our rationale as not strange based on discussions with experts of Event-B modeling and comments from reviewers, empirical study should be conducted for thorough validation.

#### Does the Planner Eliminate All the Meaningless Refinement Plans?

This is expected to be true, for validity defined by the primary rules of Event-B (see Section 5.2). The case studies showed that more definitions are necessary in order to eliminate meaningless plans not excluded by the Event-B rules. In addition, there are inevitable limitations that make it necessary to modify refinement plans later (e.g., merging properties for proofs).

#### Do Derived Plans Deal with Various Kinds of Refinement?

The main kinds of refinement in Event-B are horizontal (superposition) refinement and vertical (data) refinement. Our view of refinement as incremental addition of properties and elements naturally corresponds to horizontal refinement. In addition, our planning method can handle vertical refinement by considering gluing invariants (i.e., invariants that semantically "glue" together state spaces over multiple abstraction levels) as properties. In fact, we were able to derive refinement plans that consider vertical refinements. There are also purely operational refinements, which are sometimes useful for proof localization, but they are outside our scope.

### Does the Planning Method Derive a Small Enough Number of Refinement Plans?

The essential rationales, as well as the view simplification, worked well enough in the case studies. The Storage example is an exceptional case, but we estimate that examples with a low ratio of #properties/#elements are not difficult-to-derive plans since they tend to allow refinement plans with a large amount of freedom. In other words, we found that our planning method is effective, especially for complicated examples (i.e., examples with a high ratio of #properties/#elements) that are difficult to derive plans intuitively. We need to note, however, that it is not likely that the rationales presented in this thesis are actually sufficient for ensuring a positive answer to this question in general. Good suggestions from various corners will be necessary in many cases. In particular, developers can apply their intuition and knowledge in specific ways, e.g., by merging $A$, $B$, and $C$. Nonetheless, the approach in this thesis can potentially apply to a variety of other rationales and high-level guidelines, possibly domain-specific ones. Moreover, very large systems are inevitably handled by dividing them into subsystems or sublayers and having developers focus on these parts (this is also done in modeling and analysis). The last point is that an iterative process is facilitated if the plans are too large, as discussed in Section 6.2. Therefore, we think that we should not overlook the experience of developers or the proper decomposition of large systems.

It is also possible to sort derived plans and propose the best plan to developers. A simple but promising criterion for sorting is minimizing the maximum number of introduced elements in each step. For example, a plan $PL_A$ where $\min_{i=1\cdots n}|\text{intro}(PL_A, i)| = 3$ is preferable to $PL_B$ where $\min_{i=1\cdots n}|\text{intro}(PL_B, i)| = 5$. By sorting in this way, developers can easily find plans that effectively distribute the complexity of modeling.

### Summary

The planning method does not mean full automated support of all the aspects in refinement planning. It can support the minimum tasks to extract plans that can be judged as valid with confidence by common rationales. This point means the planning method can eliminate simple but bothersome, error-prone human tasks (e.g., collecting a set of necessary properties and elements that should be introduced together). In addition, the planning method allows for more aggressive planning by incorporating human design choices or specific high-level rationales. This can be done through interaction with the developers, as well as through incorporating more aggressive automated decisions.

### 8.2.2 Can the Planning Method Serve as a Foundation for Practical Tool Support on Refinement?

### Is the Cost of the Planning Method Acceptable?

The large cost of modeling and proving Event-B models is a primary problem. Although our planning method prevents refinement-plan-level rollback, our planning method may not optimize the actual cost. However, our experience has taught us that enabling a comparison of several refinement plans reduces the overall cost. Our future work will include a user test that involves a measurement of the actual costs in terms of the time and/or the number of proofs discharged for validation.

**Is the Cost Required to Prepare the Input to the Planning Method Acceptable?**

Input preparation of our planning method is not a trivial task. We need to consider the cost of following our elicitation process in our future work. However, some form of requirements analysis is necessary for the modeling tasks, though maybe implicitly, but should be explicitly for sharing and tracking in actual use. Moreover, as discussed in Section 6.2, developers do not need to analyze the system completely the first time because the planning method can be applied iteratively. In the future, it would also be interesting to integrate our planning method with object-oriented domain modeling. A practical tool with a good interface would also be helpful for supporting specific workflows and interactions. Further experiments with developers are also necessary.

**Can the Planning Method Assume Good Input and Guarantee Good Output?**

In the case studies, we carried out the input creation process and evaluated the derived plans ourselves. Although we did not use any knowledge other than generic patterns provided by our planning method, if other developers were to follow our planning method, they could fail to find properties that do not explicitly appear in the normal requirements analysis. However, as discussed in Section 6.2.3, the planning method is still effective even if the input is incomplete or wrong. It is possible that some developers might feel that some of the output is meaningless.

**Summary**

At the current point, there are no obvious obstacles that prevent elaborating the planning method into a practical tool. Our future work will include a user experiment involving people of different backgrounds and different Event-B modeling abilities. Establishing a requirements analysis method and concrete evaluation criteria for plans will also be important.

## 8.3 Discussion on Preliminary Phases

In this section, we discuss our preliminary method for elicitation of information on abstraction described in Chapter 7. We focus on whether our elicitation method facilitates systematic elicitation of information, valid construction of Event-B models, and smooth refinement in constructing models.

### 8.3.1 Does the Elicitation Method Facilitate Systematic Elicitation of Abstractions to Mitigate the Complexity?

In the first experiment, the participants could successfully elicit the abstractions by using the elicitation method for pattern-based analysis. This was systematic, without depending on their expertise or previous experience on formal specification methods or Problem Frames. There was also no significant difference in the results in terms of how the derived abstractions mitigate the complexity. The time they used for the three systems (at most one hour) was short enough. Thus, we believe that the elicitation method is effective for facilitating the elicitation of abstractions.

Although none of the participants of the first experiment had any experience in problem analysis in Problem Frames, they successfully derived abstract diagrams and gluing descriptions in a short enough time (less than an hour for three problems). Derived information reflected natural abstraction for each participant and thus varied with participants, but all information was successfully used to construct Event-B models over multiple abstraction levels in the second experiment. Thus, we believe that our elicitation method is so systematic that it helps even non-experts of problem analysis to derive information of abstraction of problems.

Notably, in the first experiment, the problem diagrams are given by us to purely focus on the elicitation method. We believe this preliminary part does not limit the applicability of the elicitation method as it is supported by the trusted method of Problem Frames, or more generally, the essential task to identify properties and elements is general in requirements analysis.

### 8.3.2 Do the Refinement Plans Derived by the Elicitation Method Facilitate Valid Formalization in Each Refinement Step?

In the second experiment, we succeeded in formalizing each of the steps in the refinement plan. Our approach focuses on the dependencies of properties and elements so that the specification of each step is comprehensive. The patterns and the planner are responsible for this point, and developers do not need to face this complex or at least bothersome issue. This advantage held in the experiment as well.

In the sense discussed above, the refinement plans derived by the elicitation method can support valid formalization of each refinement step. Developers who work on the target formal specification method are responsible for the other aspects of valid formalization.

For example, our elicitation method does not include any mechanism to support validity in terms of syntax correctness specific to each formal specification method. In the first experiment, the participants were experienced enough to have responsibility for this point. It is notable that this point is often supported by mechanisms for model translation or generation, which may be added as plug-ins to our elicitation method to be tailored for specific formal specification methods.

Another notable point is that we cannot ensure the constructed model is valid in the sense that the application logic passes verification (testing, theorem proving, etc.). Developers could need to modify the models to add assumptions in a certain step to pass tests or complete proofs. We cannot avoid this possibility beforehand as investigating this possibility is what we want to do after formalization.

### 8.3.3 Do the Refinement Plans Derived by the Elicitation Method Facilitate a Smooth Process through the Refinement Steps?

In the second experiment, we succeeded in completing a stepwise refinement process just by following the plan derived by the approach.

The refinement planner can suggest the most granular plans by considering the dependencies. Thus it is easy for human developers to reflect their preferences on the granularity of steps by merging the granular steps. This does not break the comprehensiveness of each step discussed in the previous discussion. This is the difficulty that our approach resolves. The difficulty otherwise comes to the surface as dividing large steps by intuitive preferences can easily break the comprehensiveness in terms of dependencies.

In this thesis, we have focused on a generic method applicable regardless of formal specification methods to be used. It is attractive to support and exemplify plugin extensions to be tailored for specific formal specification methods. The starting point is, for example, to integrate method-specific, expression-level support, such as patterns to generate specification templates (e.g., [81]).

In the first experiment, we prepared problem diagrams and metadata for abstraction. Although it is possible that he created arbitrary diagrams with prior knowledge of possible refinement of the system, we are confident the diagrams can be prepared by usual analysis in problem frames. This issue should be examined in our future work through another user experiment.

We believe effectiveness of the method for the abstraction of problems is confirmed through our experiments. Although the set of three patterns we provided is not our main contribution, all of them were used by every participant; thus, we estimate that we found important patterns. Finding other patterns through more experiments will be included in our future work.

The patterns we provided may be polished through analyzing more formal specifications. However, our user experiments showed that the patterns were effective for planning refinement. In addition, although our elicitation method helped participants to easily construct specifications, our elicitation method may not optimize the actual cost of construction.

We need to consider the cost of applying our elicitation methods. However, some analysis of the target system is needed before constructing formal specifications. Moreover, by using refinement planners, users can prevent refinement that obviously causes reworking of constructing specifications. Future work will also include measurement of the actual cost in terms of time or the number of proofs discharged for validation through user experiments.

## 8.4 Summary of Evaluation and Discussion

On the basis of our definition of refinement structure, we constructed methods that systematically manipulate dependency of properties and consistency of models. Through case studies of applying our methods, we confirmed that our methods support difficult processes of constructing and reconstructing refinement structure, which have been handled with artisanship and trial-and-errors up to now.

In the evaluation of refactoring methods (Section 6.1), we showed that our refactoring method can improve maintainability by decomposing large refinement steps into meaningful small refinement steps. We also showed that our refactoring methods can improve reusability of existing models by extracting reusable elements and properties without being confined to existing refinement structure.

In the evaluation of planning methods (Section 6.2), we succeeded in showing expression of possible refinement plans by using our refinement planner. Refinement steps of obtained plans were conceptually united, and the expression of plans for three out of four problems were small enough to be comprehensible. Moreover, by analyzing characteristics of problems and obtained plans, we found that our planning method is particularly effective for complicated problems, for which it is difficult to intuitively construct valid plans.

Even with our planning method, it is still necessary to design essential information on abstraction such as gluing invariants and abstract variables. In order to discuss feasibility of this process, we constructed a preliminary method for elicitation of such information on abstraction and carried out a preliminary case study (Chapter 7). From the result of the case study, participants, who were not experienced in modeling in Event-B successfully found information on refinement.

There are various future works of evaluation. First, in Section 4.4.2, we showed a possibility of obtaining CPs as Craig interpolants from proofs of consistency of given machines. In order to confirm this, we are planning to implement it with existing algorithms for interpolation. We also should carry out case studies to evaluate the effectiveness of our refactoring method with the heuristic using Craig interpolation and compare the results with those obtained by manually found CPs. Second, for a thorough evaluation of maintainability and reusability, we need to carry out additional evaluations, including user studies. Next, our planning method has inherent weakness of state explosion because it is based on exhaustive search. We are planning to extend the planning method to support partial search. Finally, for thorough evaluation of early-phase processes, we should carry out holistic case studies, such as user studies of eliciting abstraction information, planning, and constructing specification.

# Chapter 9

# Related Work

## 9.1 Support of Modeling in Event-B

Decomposition of Event-B machines (in "shared variable" style [5] and "shared event" style [18]) is one of the primary mechanisms to deal with complexity of modeling in Event-B. The aim of these methods is decomposing a large single machine into several components. In contrast, our goal is decomposing and merging the refinement structure of multiple machines.

Generic instantiation [73] is also considered to be a central method to deal with complexity in Event-B. This feature facilitates reuse of Event-B contexts by supporting the definition of abstract data types and instantiation of them as specifications in contexts. Our refactoring method takes a different approach to facilitate reuse of Event-B by extracting specifications about a certain set of elements. Our method can be used together with a generic instantiation approach.

## 9.2 Support of Refinement of Event-B Models

The way of refinement follows the informal system description. The studies in [70, 55, 17] generated Event-B specifications from UML diagrams, SysML models, or BPMN models. The way of refinement was described in these notations or originally embedded in the notation language (in the case of BPMN). All the above studies relied on the developers' intuitions about the refinements given in the preceding semi-formal models and did not explain how well they work in Event-B. In contrast, our approach proposes a systematic method to derive good refinement plans without relying on developer intuition.

The study in [86] provides domain-specific guidelines for control problem systems. However, they are not general and do not have primitive rationales that allow for general explanation.

The case studies in [15] presented six different modeling attempts in Event-B for one example. Although they describe different refinement plans and some of them even describe their replanning, they do not provide systematic methods for planning refinements.

This thesis has shown a different objective and approach that focuses on essential rationales and a viewpoint of potentially possible plans, rather than an approach that relies on intuitions that are often domain-specific or difficult to justify logically. On the other hand, this work would be complemented by such high-level intuitions, as well as by usable notations for the preceding phase.

Problem Frames [51] is a reliable method for analysis of problems that are solved by software systems. In the process, the location of the problem in the world, the structure of problem, and decomposition of the problem are analyzed.

Problem Frames is also combined with Event-B in several studies. In the study [59], the decomposition structure was defined (intuitively) and connected to refinement and *shared-variable decomposition* [5] of Event-B.

The study described in [41] used the WRSPM reference model [39] to describe requirements as properties.

KAOS and Event-B have been combined to provide mapping patterns between behaviors specified in an Event-B model and a KAOS goal model [62], and to provide templates of Event-B specifications [81]. These studies provide good candidates for representation of refinement plans, but do not discuss how to make decisions on the plans.

Hoang et al. proposed an approach and a tool to use patterns to facilitate systematic and modular modeling in Event-B [46]. This study can be viewed as support for modeling and reuse of Event-B models since the approach effectively uses refinement mechanism to integrate reusable patterns into Event-B models. Moreover, this approach also partially guides refinement because patterns prescribe part of refinement strategies and thus help developers to intuitively narrow down appropriate refinement strategies. Our planning method can systematically construct plans of whole refinement considering usage of such patterns by giving input information that corresponds to patterns. In addition, as we show in Section 6.1.2, our interpolation method can be used to extract patterns from existing machines. Thus, our methods are considered to be compatible with Hoang et al.'s approach and complementary to it.

In [71], Sato et al. proposed a method to plan refinement structure before constructing Event-B models. Their goal is similar to our planning method's, but their approach differs from ours. They define a tree of evidence about requirements, properties, and behaviors, which are derived through an analysis. Our planning method also accepts information from such analyses, and thus, we consider this approach as complementary to ours.

## 9.3 Arrangement and Refactoring of Formal Specification

There have been many studies on refactoring software models for the purpose of organizing and understanding them. Refactoring rules for UML/OCL [25, 60], ASM [85], Alloy [37], and Object-Z [76, 63] have been proposed. Most of them provide rules that are similar to Fowler's popular rules [36]. Most of these rules are similar to popular refactoring rules, such as move and modification, as well as rules for parameterization of expressions and introduction of inheritance and polymorphism. The goal of our work is similar to theirs, but we take a different approach based on refinement — namely by manipulating refinement structure according to criteria of the vocabulary of a machine.

## 9.4 Arrangement and Refactoring of Proof

From the point of view of refactoring of verifications, a study by Whiteside [83] has a similar goal to ours. The study manipulates proofs in proof assistants by providing a proof script framework that handles proof trees in a hierarchical way. One of the study's primary contributions is refactoring of proof scripts, including manipulating expressions of proof scripts, changing styles of proof, and generalizing tactics. Our approach, namely refinement refactoring considering the vocabulary of a module, is different from Whiteside's study.

## 9.5 Arrangement of Requirements and Informal Models

Russo et al. [69] introduced a method for restructuring requirements specifications to find inconsistencies and other useful insights about them. The study provides several *viewpoints* on restructuring that are tailored for various usages. However, its target is natural language specifications.

Stepwise refinement can be seen as a kind of evolution of software models. Although many researchers have proposed methods related to the evolution of software models [82], most of them focus only on the evolution of object models and metamodels. There is a study related to the evolution of object models and constraints on them (in OCL) [29], but it aimed to realize co-evolution of constraints and object models according to evolution of each other. In our method, we provide property-centric methods for planning the evolution and constructing specifications by starting from an abstract specification — and making it more and more concrete.

In [40], a notion of *problem transformation* is proposed in a formal way. Problem transformation is an operation to derive a problem P1 from another problem P2, such that P1 describes solution of P2 in a more detailed way. It corresponds to ascending a "proof tree" of an argument over a problem. On the basis of problem transformation, an approach for deriving specification from requirements (*problem progression* [51, p.103]) has been proposed [66]. This process is important because requirements are not necessarily controlled or observed by the machine, and thus not easy for developers to interpret from the point of view of implementation. Moreover, in [72], another progression technique (requirement progression) also produces descriptions of relationships between before and after a step of progression (breadcrumbs). These approaches are similar to ours. Our methods can be classified as a problem transformation, and the concept of gluing descriptions and breadcrumbs are similar. However, these methods are for reduction and simplification of the problem in a single level of abstraction. Our methods are complementary to these progression techniques. Integrating our methods into the foundation of these techniques will be included in our future work.

Goal-oriented requirements analysis methods, such as KAOS [27], have been widely used to consider multiple levels of abstraction in arranging and analyzing requirements. In KAOS, a goal of the target system is decomposed through and/or decomposition, and allocated to agents to connect them with the implementation. Formal analysis of goal models has also been performed for a long time. Although studies on the KAOS method consider and/or decomposition of goals, they do not deal with our target, namely additional information and division of goals for comprehensive abstraction.

## 9.6 Bridging Informal Artifacts and Formal Artifacts

The ProR approach [53] proposed by Jastram helps developers to systematically handle informal requirements so that developers have traceability of requirements and system descriptions, such as formal specifications. With this approach, consistent system descriptions are constructed from a set of requirements through formal and informal reasoning. The approach has been integrated with Event-B and supports incremental elaboration of system descriptions by using refinement. Integrating our planning method, which aims to examine strategies for refinement, with ProR approach and its tool implementation will be interesting. Moreover, enhancing our refactoring method with ProR's feature for traceability will help developers to carry out more comprehensible refactoring of existing Event-B models.

Liu proposed a method named SOFL [58], which orchestrates specification language, method, and software process to bridge requirements analysis, design, and coding. In the process, informal specification, semi-formal specification, formal specification of abstract design, formal specification of detailed design, and program code are created sequentially. Conversely, our elicitation method and planning method are intended to facilitate the construction of comprehensive and rigorous initial specification.

## 9.7 Application of Craig Interpolation

A number of significant studies on formal methods have used Craig interpolation of logic formulas, which we found to be important for finding CPs. One of the primary applications of interpolation is counterexample-guided abstraction refinement [22] in model checking, which constructs a series of interpolants from the spurious behaviors of an abstract model and uses them to refine the model. One study [28] used interpolation to automatically construct a behavior model of a system from its goal model. The approach described therein updates a behavior model by using interpolants of counterexamples and goals. In the future, we believe we can use Craig interpolation in a similar way to systematically find CPs from models expressed in certain classes of expressions.

# Chapter 10

# Conclusion

## 10.1 Summary

Our goal was solving various problems related to the application of Event-B by explicitly manipulating the refinement structure of models, namely what aspects are introduced in each step of refinement. Based on our view on the refinement structure of Event-B models, we designed our approach so that it controls usability, comprehensibility, and complexity of Event-B models by considering the consistency of models and the dependency of properties and elements.

We aimed to improve maintainability and reusability of Event-B models by refactoring the refinement structure of them. We proposed a method to restructure the refinements of Event-B models according to refactoring criterion in terms of the vocabulary of a new model. Our main method finds necessary variables and predicates from the original models and helps to manually find complementary predicates to make the new model consistent with original models. This helps users to construct an abstraction of an existing model that focuses on certain aspects of the original model. In case studies, we used our method to split up refinements in large-scale Event-B models and succeeded in constructing small and consistent models. Moreover, we succeeded in extracting predicates of an existing model that are about reusable aspects to construct a new model.

We also tackled an advanced problem of refinement planning before construction of models, which is very essential among mixed activities to make effective and efficient use of Event-B. The challenge here was finding valid refinement plans from limited information from documents of the target system, before rigorous modeling and verification in Event-B. In order to address the challenge, we defined general rationales to determine whether a refinement structure is meaningful and it follows common refinement strategies. The rationales enable us to explicitly argue desirability of a plan. A planner tool for the problem of refinement planning has also been presented on the basis of the fundamentals. In addition, we discussed the feasibility of finding information of possible abstractions from informal analysis on the target problem, by showing a preliminary method and a case study using the method. That case study and discussions on the planning method have demonstrated how the method works effectively and helps tackle the essential difficulties, directly or potentially.

Therefore, we conclude that our proposed methods support developers in planning and refactoring the refinement structure of Event-B models.

Our primary future work will be thorough user experiments on eliciting abstraction information, planning, and constructing models.

## 10.2 Foresight

### 10.2.1 Systematization of Elicitation Method

We are interested in making our information elicitation method more systematic by using metadata on problem diagrams and by integrating the method into the foundation of Problem Frames. Currently, our information elicitation method uses only notation of Problem Frames. We also want to use the argumentation method of Problem Frames to elicit information on abstraction.

### 10.2.2 Thorough Experiments Including User Studies

As we argued in Section 8.4, one of our primary future works will be to perform thorough experiments, in particular for methods related to early phases of development, such as elicitation of information on abstraction and planning refinement structure. We would like to conduct user experiments for the whole process of eliciting information, planning, modeling, and verification.

### 10.2.3 Comparison of Multiple Specifications Constructed with Refinement Restructuring

Now that our method for refinement refactoring enables systematic restructuring, restructuring refinement structure of a single model by following various refinement structures and comparing the results will be an interesting future work.

We anticipate that the findings from this experiment will also be insightful from the point of view of refinement planning.

### 10.2.4 Various Applications of Refinement Refactoring

We expect that our refactoring method will have various potential applications.

#### Porting Specifications from Another Formal Method to Event-B

There are various formal specification methods that have state-based notation similar to Event-B's, such as VDM. Porting specifications from such other methods to Event-B is expected to be useful because it will enable developers to leverage the refinement mechanism of Event-B to do verification.

Because specifications for such methods usually do not consider multiple abstractions, developers need to introduce abstraction layers of target systems. To this end, developers can use our planning method and refactoring method to systematically introduce abstraction layers in a comprehensible manner.

#### Decomposing Refinement for Specific Purposes

Extracting special aspects of the Event-B model to analyze it with other methods will be interesting. For example, there have been various proposals to integrate Event-B with model checkers, including a probabilistic one [57, 79]. Extracting aspects that should be investigated with such model checkers will help developers to focus on essential parts of model and shrink state space.

#### Prepare for Arbitrary Restructuring towards Flexible Use

Although our current approach for restructuring employs a "slice-and-complement" strategy, it is also possible to compute all possible ways of restructuring and then flexibly restructure. This approach will be effective to facilitate use of Event-B models in the models@run.time approach [14], which aims to leverage models of target systems for monitoring environment, analyzing monitored data, and changing behavior at runtime. We consider that using Event-B models with many possibilities of multiple abstraction layers is useful in various ways, such as reducing costs for monitoring and reasoning.

# References

[1] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, 2005.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, 2010.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, 2010.

[4] Jean-Raymond Abrial, Dominique Cansell, and Guy Laffitte. "Higher-Order" Mathematics in B. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors, *ZB 2002:Formal Specification and Development in Z and B: 2nd International Conference of B and Z Users Grenoble, France, January 23–25, 2002 Proceedings*, pages 370–393. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[5] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[6] Advance Project. Advanced Design and Verification Environment for Cyber-physical System Engineering. `http://www.advance-ict.eu/`. [Online; accessed 06-December-2016].

[7] Information Promotion Agency. Report on Application of Formal Methods to Real-World Information Systems (in Japanese). `http://www.ipa.go.jp/sec/softwareengineering/reports/20120420.html`. [Online; accessed 10-February-2017].

[8] Aws Albarghouthi and Kenneth L. McMillan. Beautiful Interpolants. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 313–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] R. J. R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. *Distributed Computing*, 3(2):73–87, 1989.

[10] Ralph-Johan Back and Joakim Wright. *Refinement Calculus: a Systematic Introduction.* Springer Science & Business Media, 2012.

[11] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal*

*Specification and Development in Z and B: 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005. Proceedings*, pages 334–354. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[12] Richard Banach and Michael Butler. Cruise Control in Hybrid Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theoretical Aspects of Computing – ICTAC 2013: 10th International Colloquium, Shanghai, China, September 4-6, 2013. Proceedings*, pages 76–93. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A Successful Application of B in a Large Project. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*, pages 369–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[14] G. Blair, N. Bencomo, and R. B. France. Models@run.time. *Computer*, 42(10):22–27, Oct 2009.

[15] Frédéric Boniol and Virginie Wiels. The Landing Gear System Case Study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study: Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings*, pages 1–18. Springer International Publishing, 2014.

[16] J. W. Bryans, J. S. Fitzgerald, A. Romanovsky, and A. Roth. Patterns for Modelling Time and Consistency in Business Information Systems. In *2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 105–114, March 2010.

[17] Jeremy Bryans and Wei Wei. Formal Analysis of BPMN Models Using Event-B. *Formal Methods for Industrial Critical Systems*, pages 33–49, 2010.

[18] Michael Butler. Decomposition Structures for Event-B. In Michael Leuschel and Heike Wehrheim, editors, *IFM 2009*, volume 5423 of *LNCS*, pages 20–38. Springer, Heidelberg, 2009.

[19] Michael Butler and Issam Maamria. Practical Theory Extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, pages 67–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] Michael J. Butler. Stepwise Refinement of Communicating Systems. *Sci. Comput. Program.*, 27(2):139–173, September 1996.

[21] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings*, pages 359–364. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[22] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E.Allen Emerson and AravindaPrasad Sistla, editors, *CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, Heidelberg, 2000.

[23] ClearSy. Atelier B: the Industrial Tool to Efficiently Deploy the B Method. `http://www.atelierb.eu/`. [Online; accessed 06-December-2016].

[24] Peter Coad and Edward Yourdon. *Object-oriented Design.* Yourdon Press computing series. Yourdon, 1991.

[25] Alexandre Correa, Cláudia Werner, and Márcio Barros. An Empirical Study of the Impact of OCL Smells and Refactorings on the Understandability of OCL Specifications. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS 2007*, volume 4735 of *LNCS*, pages 76–90. Springer, Heidelberg, 2007.

[26] William Craig. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[27] R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. GRAIL/KAOS: an Environment for Goal-driven Requirements Engineering. In *Proceedings of the 19th international conference on Software engineering*, pages 612–613. ACM, 1997.

[28] Renzo Degiovanni, Dalal Alrajeh, Nazareno Aguirre, and Sebastian Uchitel. Automated Goal Operationalisation Based on Interpolation and SAT Solving. In *Proceedings of the 36th International Conference on Software Engineering*, pages 129–139, New York, 2014. ACM.

[29] Andreas Demuth, Roberto E Lopez-Herrejon, and Alexander Egyed. Supporting the Co-evolution of Metamodels and Constraints through Incremental Constraint Management. In *Model-Driven Engineering Languages and Systems*, pages 287–303. Springer, 2013.

[30] Deploy Project. Deploy Project. `http://www.deploy-project.eu/`. [Online; accessed 06-December-2016].

[31] Edsger W. Dijkstra. The Humble Programmer. *Commun. ACM*, 15(10):859–866, October 1972.

[32] Rodin User Documentation. Refactoring Framework. `http://wiki.event-b.org/index.php/Refactoring_Framework`. [Online; accessed 10-February-2017].

[33] Event-B.org. Event-B.org. `http://www.event-b.org/`. [Online; accessed 06-December-2016].

[34] J. Fitzgerald, P.G. Larsen, P. Mukherjee, N. Plat, and M. Verhoef. *Validated Designs for Object-oriented Systems.* Springer London, 2005.

[35] Dependable Software Forum. Dependable Software Forum (in Japanese). `http://www.nttdata.com/jp/ja/news/release/2010/112400.html`. [Online; accessed 10-February-2017].

[36] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Pearson Education India, 1999.

[37] Rohit Gheyi and Paulo Borba. Refactoring Alloy Specifications. *Electronic Notes in Theoretical Computer Science*, 95:227–243, 2004.

[38] Ali Gondal, Michael Poppleton, and Michael Butler. Composing Event-B Specifications - Case-Study Experience. In Sven Apel and Ethan Jackson, editors, *Software Composition: 10th International Conference, SC 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, pages 100–115. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[39] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A Reference Model for Requirements and Specifications. *Software, IEEE*, 17(3):37–43, 2000.

[40] Jon G Hall, Lucia Rapanotti, and Michael Jackson. Problem Oriented Software Engineering. Technical Report 2010/03, Department of Computing, The Open University, January 2010.

[41] S. Hallerstede, M. Jastram, and L. Ladenberger. A Method and Tool for Tracing Requirements into Specifications. *in Electronic Communications of the EASST*, 2012.

[42] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-Animation for Event-B — Towards a Method of Validation. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z: Second International Conference, ABZ 2010, Orford, QC, Canada, February 22-25, 2010. Proceedings*, pages 287–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[43] Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[44] Ian Hayes and Bill Flinn. *Specification Case Studies.* Prentice-Hall International London, 1987.

[45] Thai Son Hoang and Jean-Raymond Abrial. Reasoning about Liveness Properties in Event-B. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering: 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, pages 456–471. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[46] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. Event-B Patterns and Their Tool Support. *Software & Systems Modeling*, 12(2):229–244, 2013.

[47] C. A. R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[48] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

[49] John E. Hutchinson. Model Theory via Set Theory. *Israel Journal of Mathematics*, 24(3):286–304, 1976.

[50] Fuyuki Ishikawa, Alexander Romanovsky, and Elena Troubitsyna. Event-B Day. National Institute of Informatics Tokyo, Japan. Technical report, School of Computing Science, University of Newcastle upon Tyne, 2016.

[51] M. Jackson. *Problem Frames: Analysing and Structuring Software Development Problems.* Addison-Wesley, 2001.

[52] Michael Jackson. The Role of Formalism in Method. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems Toulouse, France, September 20–24, 1999 Proceedings, Volume I*, pages 56–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[53] Michael Jastram. *The ProR Approach: Traceability of Requirements and System Descriptions.* PhD thesis, Heinrich Heine University Düsseldorf, 2012.

[54] Cliff B Jones. *Systematic Software Development Using VDM*, volume 2. Citeseer, 1986.

[55] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouet. A First Attempt to Combine SysML Requirements Diagrams and B. *Innovations in Systems and Software Engineering*, 6(1-2):47–54, 2010.

[56] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[57] Michael Leuschel and Michael Butler. ProB: A Model Checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, pages 855–874. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[58] Shaoying Liu. *Formal Engineering for Industrial Software Development: Using the SOFL Method.* Springer Science & Business Media, 2013.

[59] F. Loesch, R. Gmehlich, K. Grau, C. Jones, and M. Mazzara. Report on Pilot Deployment in Automotive Sector. *A Deliverable of Deploy Project*, 2010.

[60] Slaviša Marković and Thomas Baar. Refactoring OCL Annotated UML Class Diagrams. In Lionel Briand and Clay Williams, editors, *MoDELS 2005*, volume 3713 of *LNCS*, pages 280–294. Springer, Heidelberg, 2005.

[61] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification. In *Proceedings of the 37th International Conference on Software Engineering*, pages 722–732, New York, 2015. ACM.

[62] A. Matoussi, F. Gervais, and R. Laleau. A Goal-Based Approach to Guide the Design of an Abstract Event-B Specification. In *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 139–148. IEEE, 2011.

[63] Tim McComb and Graeme Smith. A Minimal Set of Refactoring Rules for Object-Z. In Gilles Barthe and FrankS. de Boer, editors, *FMOODS 2008*, volume 5051 of *LNCS*, pages 170–184. Springer, Heidelberg, 2008.

[64] C Métayer. AnimB Homepage. `http://animb.org/`. [Online; accessed 10-February-2017].

[65] Jayadev Misra. A Logic for Concurrent Programming. *University of Texas at Austin*, 1994.

[66] L. Rapanotti, J. G. Hall, and Z. Li. Deriving Specifications from Requirements through Problem Reduction. *IEE Proceedings - Software*, 153(5):183–198, October 2006.

[67] Antoine Requet. BART: A Tool for Automatic Refinement. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z: First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, pages 345–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[68] Alexander Romanovsky and Martyn Thomas. *Industrial Deployment of System Engineering Methods.* Springer Publishing Company, Incorporated, 2013.

[69] A. Russo, B. Nuseibeh, and J. Kramer. Restructuring Requirements Specifications for Managing Inconsistency and Change: A Case Study. In *Requirements Engineering, 1998. Proceedings. 1998 Third International Conference on*, pages 51–60, 1998.

[70] Mar Yah Said, Michael Butler, and Colin Snook. Language and Tool Support for Class and State Machine Refinement in UML-B. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 579–595. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[71] Naoto Sato and Fuyuki Ishikawa. Separation of Considerations in Event-B Refinement toward Industrial Use. *Formal Methods in Software Engineering Education and Training 2015*, 2015.

[72] Robert Seater, Daniel Jackson, and Rohit Gheyi. Requirement Progression in Problem Frames: Deriving Specifications from Requirements. *Requirements Engineering*, 12(2):77–102, 2007.

[73] Renato Silva and Michael Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering: 11th International Conference on Formal Engineering Methods ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, pages 466–484. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[74] J Michael Spivey and JR Abrial. *The Z Notation.* Prentice Hall Hemel Hempstead, 1992.

[75] Stephen Wright. A Formally Constructed Instruction Set Architecture Definition of the XCore Microprocessor. `http://deploy-eprints.ecs.soton.ac.uk/346/`. [Online; accessed 09-February-2017].

[76] Susan Stepney, Fiona Polack, and Ian Toyn. Refactoring in Maintenance and Development of Z Specifications and Proofs. *ENTCS*, 70(3):50 – 69, 2002.

[77] Wen Su, Jean-Raymond Abrial, and Huibiao Zhu. Formalizing Hybrid Systems with Event-B and the Rodin Platform. *Science of Computer Programming*, 94, Part 2:164 – 202, 2014.

[78] Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, and Timo Latvala. The Formal Derivation of Mode Logic for Autonomous Satellite Flight Formation. In *SAFECOMP 2015*, volume 9337 of *LNCS*, pages 29–43. Springer, Heidelberg, 2015.

[79] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Integrating Stochastic Reasoning into Event-B Development. *Formal Aspects of Computing*, 27(1):53–77, 2015.

[80] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[81] Kriangkrai Traichaiyaporn and Toshiaki Aoki. Refinement Tree and Its Patterns: A Graphical Approach for Event-B Modeling. In Cyrille Artho and Peter Csaba Ölveczky, editors, *Formal Techniques for Safety-Critical Systems: Second International Workshop, FTSCS 2013, Queenstown, New Zealand, October 29–30, 2013. Revised Selected Papers*, pages 246–261. Springer International Publishing, Cham, 2014.

[82] Guido Wachsmuth. Metamodel Adaptation and Model Co-adaptation. In *ECOOP 2007–Object-Oriented Programming*, pages 600–624. Springer, 2007.

[83] Iain Johnston Whiteside. *Refactoring Proofs*. PhD thesis, The University of Edinburgh, 2013.

[84] Niklaus Wirth. Program Development by Stepwise Refinement. *Commun. ACM*, 14(4):221–227, April 1971.

[85] Hamed Yaghoubi Shahir, Roozbeh Farahbod, and Uwe Glässer. Refactoring Abstract State Machine Models. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *ABZ 2012*, volume 7316 of *LNCS*, pages 345–348. Springer, Heidelberg, 2012.

[86] Sanaz Yeganefard, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a Guideline by Formal Modelling of Cruise Control System in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, pages 182–191. NASA, April 2010.