Abstraction and Search in Verification by State Exploration
状態探査による検証における抽象化と探索

Koichi Takahashi
高橋 孝一

A thesis submitted in partial fulfillment of
the requirements for the degree of Doctor of Philosophy
in Information Science at the University of Tokyo

January 2002

Abstract

The importance of software verification is still growing due to the increase of safety-critical systems including those for supporting network infrastructures. As a verification technology, the method called model checking, which is based on automatic state exploration, is widely used in the field of hardware verification and is expected to be usable also for verifying software. However, software has large state spaces compared with hardware in general, and in many cases model checking cannot be directly applied to verifying it. Therefore, in order for model checking to be successful in software verification, techniques for reducing state spaces are indispensable. One of the most important techniques to reduce state spaces is abstraction, which has been successfully applied to model checking of hardware. The purpose of this thesis is to investigate abstract model checking, i.e., model checking enhanced with abstraction techniques, for software verification.

In Chapter 3 of the thesis, in order to examine how abstraction works in model checking, we verify algorithms for concurrent garbage collection (GC) as a case study. Since the correctness of concurrent GC algorithms is hard to prove manually, automatic tools such as abstract model checking are effective for verifying those algorithms. Through the case study, two important issues arise concerning abstract model checking. One is the need for a theoretical foundation that guarantees that properties of an original algorithm are preserved by abstraction. Another is how to find appropriate abstraction for specific problems.

In Chapter 4, we give a theoretical foundation for abstract model checking. By applying the refinement theory of program semantics, we make the relation between model checking and abstraction theoretically clear.

In the case study of verifying concurrent GC algorithms, we have to abstract states of the heap. The abstraction of the heap is not trivial because it is a complex link structure which admits infinite possibilities of abstraction. In Chapter 5, we develop a general abstraction technique for link structures. Using regular expressions, we generalize the abstraction used in the abstract model checking of concurrent GC algorithms. It is a technique for dealing

with the second issue raised in Chapter 3.

Model checking not only automatically verifies that given software satisfies given specifications, but is also expected to be used for automatic synthesis of software which satisfies given specifications. In Chapter 6, for automatic algorithm synthesis, we propose a method for using model checking in algorithm exploration. As case studies, we try to discover some new algorithms for concurrent garbage collection and mutual exclusion. In Chapter 7, in order to increase the efficiency of algorithm exploration, we apply a symbolic model checking technique on the space of algorithms. By using this method, we can search through the algorithm space by a single execution of model checking. Moreover, to further reduce the search space, we try to apply an abstraction technique to the method, i.e., we combine symbolic exploration and abstraction. As a case study of the method, we search for algorithms for mutual exclusion.

Finally, we describe research that aims to formally prove the correctness of the entire process of abstract model checking. In particular, abstraction used in abstract model checking must satisfy some conditions to guarantee the correctness of the entire process of abstract model checking. Ideally, these conditions should be formally proven. In Chapter 8, we give a formal proof of the correctness of the abstraction used in the abstract model checking of concurrent GC algorithms. The formal proof was developed on a proof assistant system called HOL. In connection with formal proofs, we also describe research on the environment of proof assistants in Chapter 9. We first propose "proving as editing paradigm" as a guideline for user interfaces of proof assistant systems. This guideline was actually employed in the formal proof in Chapter 8. We also describe a graphical user interface for the commutative diagrams that often appear in the field of program semantics.

概要

ネットワークのインフラストラクチャを含むセイフティ・クリティカルなシステムが増えるに従って，ソフトウェア検証の重要性は今後ますます大きくなって行くと考えられる．状態の自動的な探索を基礎とするモデル検査法は，検証技術の一つとして，ハードウェア検証の分野で広く利用されており，ソフトウェアの検証に対してもその利用が期待されている．しかし，ソフトウェアはハードウェアに比べて状態空間が巨大であり，ソフトウェア検証に対してモデル検査法を直接には適用できない場合が多い．従って，ソフトウェアのモデル検査を成功させるためには，状態空間を小さくする技術が不可欠である．状態空間を小さくする最も重要な方法の一つとして抽象化があり，ハードウェアのモデル検査に対しては成功している．本論文は，ソフトウェア検証を行うために，抽象化の技術を付加したモデル検査法，すなわち，抽象モデル検査法について探求することを目標としている．

本論文の第 3 章では，モデル検査法における抽象化の有効性を調べるために，一つのケーススタディとして，並行ごみ集めのアルゴリズムの検証を行う．並行ごみ集めアルゴリズムの正しさは簡単に証明できるものではなく，抽象モデル検査のような自動検証が有効な手段となる．このケーススタディを通じて，抽象モデル検査における二つの重要な課題が明らかになる．一つは，抽象化によって元のアルゴリズムの性質が失われないことを保証する理論的基礎が必要なことである．もう一つは，問題に応じて適切な抽象化をどのように構成するかという課題である．

第 4 章では，抽象モデル検査の理論的基礎を与える．プログラム意味論の分野で研究されていた詳細化の理論を適用することによって，モデル検査と抽象化の関係を理論的に解明する。

並行ごみ集めアルゴリズムを検証するケーススタディでは，ヒープの状態を抽象化する必要がある．ヒープは複雑なリンク構造を成し抽象化の可能性も無限にあるため，ヒープの抽象化をどのように定義すべきかは自明ではない．第 5 章では，リンク構造を抽象化するための一般的な技術を開発する．正則表現を用いることにより，並行ごみ集めアルゴリズムの抽象モデル検査に用いた抽象化を一般化する．この技術は，第 3 章で与えた二番目の課題の一つの答えになっている．

モデル検査法は，与えられたソフトウェアが仕様を満たしているかどう

かを自動検証するだけでなく，与えられた仕様を満たすソフトウェアを自動合成する問題にも適用可能なことが期待される。第 6 章では，アルゴリズムを自動合成することを目標に，モデル検査を用いてアルゴリズムの探索を行う方法を提案する．そのケーススタディとして，並行ごみ集めのアルゴリズムと排他制御のアルゴリズムの発見を試みる．第 7 章では，アルゴリズムの探索の効率を上げるために，シンボリックなモデル検査法をアルゴリズムの空間に適用する．この方法を用いることにより，モデル検査の一回の実行でアルゴリズム空間を探索することが可能である．さらに探索空間を小さくするため，この方法に抽象化技術を適用する．すなわち，シンボリックな探索と抽象化を組み合わせる．この方法のケーススタディとして，相互排除のアルゴリズムの探索を行う．

最後に、抽象モデル検査法の全プロセスの正しさを形式的に証明することを目指す研究について述べる．特に，抽象モデル検査で用いられる抽象化は，抽象モデル検査全体の正しさを保証するために，いくつかの条件を満たしていなければならない．理想的には，これらの条件は形式的に証明されるべきである．第 8 章では，並行ごみ集めの抽象モデル検査で用いられた抽象化の正しさの形式的証明を与える。形式的証明は，HOL と呼ばれる証明支援系の上で開発された．形式的証明と関連して，第 9 章では，証明支援系のための環境に関する研究についても述べる．まず，証明支援系のユーザ・インタフェースのためのガイドライトして，“proving as editing paradigm” と呼ぶ考えを提案する．このガイドラインは，第 8 章の形式的証明に際して実際に活用されたものである．また，プログラム意味論の分野で頻出する可換図をそのまま扱うことのできるグラフィカル・ユーザ・インタフェースについても述べる．

Acknowledgments

First of all, I would like to express my gratitude to Masami Hagiya. He was my supervisor when I studied in the University of Tokyo from April 1997 to March 1998. He introduced me to the field of abstract model checking. Without his invaluable support, insightful suggestions, and continual encouragement to now I would have never written this thesis.

I also thank to Akinori Yonezawa, Shinichi Morishita, Tetsuo Tamai, Kei Hiraki, and Kokichi Futatsugi for their willingness to be members of the thesis committee.

I would like to deeply thank Yoshiki Kinoshita for his suggestions, comments, and great support in National Institute of Advanced Industrial Science and Technology (AIST), to which I have belonged.

I am grateful to Mitsuharu Yamamoto, Ichiro Ogata, Isao Kojima and John Power for their helpful comments. I am also grateful to all the members of Hagiya Laboratory and my colleagues of AIST.

Finally, I sincerely thank my parents.

Contents

Chapter 1

Introduction

Verification of computer systems such as software, algorithms, and communication protocols, is becoming more and more important due to the increase of computer systems that support the infrastructures of everyday life. Among them, rigorous verification techniques have been applied only to a very small number of safety-critical systems so far. However, commercial software now causes huge economic damage because of widespread personal computers and networks. This situation raises the importance of the correctness of software, and its underlying algorithms and protocols. Software is usually checked by tests, but, checking by tests is incomplete in general. For a complete check, formal verification is required.

In this chapter, we first describe the background and motivations of the research described in this thesis. We then outline the thesis by introducing its chapters and summarizing its contributions.

1.1 Background and Motivations

There are two main approaches to formal verification. One approach is theorem proving, which constructs a formal proof from axioms and inference rules. A merit of this approach is that any properties of any systems can be proven in principle. However, fully automatic construction of formal proofs is impossible in general. Various kinds of heuristics are needed for theorem proving. Consequently, verification by theorem proving is very expensive. To reduce the cost, a number of proof assistant systems have been developed. However, expert knowledge is still required to use such proof assistant systems. Furthermore, theorem proving takes a long time even if an expert uses a proof assistant system. Therefore, theorem proving is not practical when a large amount of software have to be verified.

Another approach is model checking, which verifies a given property of a given finite system by exhaustively exploring its possible states. Model checking is a fully automatic verification technique. Needless to say, automatic verification by computers is desirable for practical software development, because no expert knowledge is needed, and the power of computers is increasing every day.

Model checking has already been used in hardware verification, because hardware has a small state space compared with software. Although model checking is also expected to be used in software verification, software often has very large or infinite state. This state space explosion is a major problem in model checking of softwares. Recently many techniques for model checking have been developed to solve the problem, such as binary decision diagrams, partial order reduction, and the tableau method. Thanks to these methods, model checking has now become practical for hardware verification. Nevertheless, it is still not applicable to software verification in general.

As mentioned above, the obstacle to model checking of software is state space explosion. Software often have infinite state, or finite but large state. To use model checking, an infinite (or huge) system has to be abstracted to a rather small finite one. Abstraction is a critical technique for model checking of software. Model checking of the system reduced by abstraction is called abstract model checking.

Beyond verification, automatic synthesis of software, algorithms or protocols by using model checking is also an important research subject. Verification techniques would be widespread if they could not only verify existing systems but also synthesize new ones according to given specifications. The basic procedure for automatic synthesis by model checking, employed in this thesis, is to generate a space of candidates of software, then automatically check them by model checking, and finally find software satisfying the given specifications. An important point of this procedure is how to explore the space of candidates. It is also possible to apply abstraction to this exploration.

Although model checking is a useful verification technique, theorem proving is often needed for guaranteeing the correctness of the entire process of verification. For example, the correctness of the abstraction used in abstract model checking should be formally verified. It is only possible by a formal proof from axioms and inference rules. In this sense, the combination of model checking and theorem proving is required.

1.2 Overview

According to the motivation discussed in the previous section, this thesis investigates abstract model checking, i.e., model checking enhanced with abstraction techniques, towards software verification. The subjects of the thesis are classified into the following three groups.

- Abstract model checking for formal verification.
- Synthesis by model checking and abstraction.
- Verification of the entire process of abstract model checking.

In the following, we outline the thesis according to the three.

In Chapter 2, we briefly sketch the foundation of model checking and refer to the related work on the three subjects. We also compare the thesis with the related work.

In Chapters 3–5, we describe research on the first subject: abstract model checking for formal verification.

In Chapter 3, we do a case study of abstract model checking of a complex example in software verification. We choose algorithms for concurrent garbage collection (GC) as an example. It is known that "security holes" of network software come from the improper assignment of memory such as buffers or stacks. Automatic memory allocation may be the key to solve these problems, but it is still unclear whether memory reclamation algorithms, i.e., garbage collectors, are safe or not. Especially, the correctness of "concurrent" GC algorithms is hard to prove manually, because the human being is far from perfect in exhaustive search of all branching cases. Hence we believe that abstract model checking itself is useful as an automatic, computer-aided verification tool for software of this kind.

Ordinary finite model checking restricts the size of the heap, which is a component of the model of concurrent GC. Therefore, to verify concurrent GC with any size of heap by model checking, the heap should be abstracted. We construct an abstract heap whose size is independent of the size of the original concrete heap and is small and finite. We abstract the concurrent GC algorithms by using the abstract heap, and verify the safety of the abstract system. The safety of the concurrent GC algorithms is that any free cells are not reachable from the register (root) at any time. The concurrent GC algorithms we verify are the so-called "on-the-fly" GC and "snapshot" GC. We use the same abstraction to verify both of them. Moreover, the variants of GC algorithms, which are discovered in Chapter 6, are verified by the same abstraction. We also verify the liveness of the concurrent GC algorithms by

abstract model checking. The liveness is that any garbage cells will eventually become free.

Two important issues arise concerning abstract model checking through the case study. One is the need for a theoretical foundation that guarantees that properties of an original algorithm are preserved by abstraction. Another is how to find appropriate abstraction for specific problems. In the following two chapters, we describe research on these issues.

In Chapter 4, we make theoretical analysis of abstract model checking. Since abstract model checking only verifies a property of the abstract system, it is not obvious whether the original concrete system also satisfies the same property. To prove that the former implies the latter, abstract model checking has to be analyzed theoretically.

Theoretical treatment of refinement of terminating programs has been done. Hoare-He-Sandars theory, which is an extension of Hoare logic, is an example. We employ Hoare-He-Sandars theory for the analysis of abstract model checking because refinement is similar to abstraction. Since model checking is a verification technique for reactive systems which often do not terminate, Hoare-He-Sandars theory cannot be applied to such systems directly. So we introduce the new notion "cumulatives" to analyze the safety of non-terminating systems, and prove the correctness of abstract model checking with respect to safety. We then describe the required conditions for the abstract model checking of safety. We finally show that the case study in Chapter 3 is an instance of our theory. Therefore, the safety of the original GC algorithm can be verified by abstract model checking.

Many abstraction methods of simple data structures such as integers have been proposed and commonly used. For example, integer data can be abstracted to positive/negative, even/odd, or prime/composite. On the other hand, there are only few abstraction methods for complex data structures such as heaps, and proposing new abstraction methods for such data structures is an important research subject.

The heap of concurrent GC, abstracted in Chapter 3, is a link structure. By generalizing the method in Chapter 3, we propose a general abstraction method for link structures in Chapter 5. We use regular expressions as attributes of a cell in a link structure, and make an abstract cell by calculating the attributes of each cell by regarding a link structure as an automaton. We then construct an abstract state by collecting abstract cells. We show that the abstraction in Chapter 3 is an instance of this abstraction of link structures.

In Chapters 6 and 7, we describe research on the second subject: synthesis by model checking and abstraction.

As mentioned in the previous section, verification techniques would be more practical if they could also synthesize new systems according to given specifications. Automatic synthesis of software, algorithms and protocols is discussed in Chapter 6. We do two case studies of automatic algorithm synthesis by using model checking. The basic procedure is simple. First, we generate a space of candidates of algorithms, then we check them by model checking, and finally we find algorithms satisfying the specifications. We examine this procedure through case studies.

The first one is on the synthesis of concurrent GC algorithms, and the second on the synthesis of mutual exclusion algorithms. The reasons why we select these algorithms for case studies are that their correctness is not obvious, so model checking is effective, and it is an interesting question whether there exist new algorithms (for concurrent GC or mutual exclusion) in addition to the existing known ones. In the case study of concurrent GC, we first select some operations of the mutator and the collector. We then define the algorithm space as the set of sequences of truth values. Each truth value in a sequence indicates permission/prohibition of the corresponding operation of the mutator or the collector. For each sequence of truth values, we check whether the corresponding algorithm satisfies the safety property by model checking. In the case study, we found several bit patterns that express possibly safe GC algorithms. The resulting algorithms we found are the on-the-fly, the snapshot, and their three variants. In this case study, we use a small model that has only three cells in the heap for model checking. The correctness of the variants has been verified by abstract model checking.

In the case study of mutual exclusion, we try to discover variants of Dekker's algorithm. We first design a small language of pseudo-codes. The algorithm space is then defined as the set of programs with a fixed length in the language. We explore this algorithm space by model checking. In the case study, we found that Dekker's algorithm was the shortest one in this language. We also discovered a new program, which was shorter than Dekker's, under an assumption on the behaviors of the processors. Note that in these case studies, each algorithm is individually checked by model checking.

In Chapter 7, we apply the symbolic model checking technique to the exploration of an algorithm space. In symbolic model checking, binary decision diagrams (BDDs) are used to represent and manipulate sets of states rather than individual states. By representing an algorithm space using BDDs, we can explore the space by a single run of model checking.

As a case study, we try to discover mutual exclusion algorithms by this method. We first parameterize some parts of Dekker's algorithm, and define the algorithm space by the values of the parameters. The representation of

an algorithm by parameters is suitable for BDDs. We then apply symbolic model checking to the algorithm space. In the case study, we found some values of parameters that satisfy the specifications. However, the algorithms corresponding to the parameters were essentially equivalent to the original Dekker's algorithm. We also constructed another algorithm space which contained both Dekker's and Peterson's algorithms. Unfortunately, we could not discover any essentially new algorithms.

As for the latter algorithm space, we also apply abstraction to the method. We first construct an approximate BDD by collapsing some sub-diagrams which are close to one another in Hamming distance. We then use this approximation as the abstraction in the exploration of mutual exclusion algorithms. This means that the algorithm space is reduced by the approximation. In order to ensure the correctness of the reduction of the algorithm space, we prove that algorithms discovered in the abstract space always satisfy the required specifications.

In Chapters 8 and 9, we describe research on the last subject: verification of the entire process of abstract model checking.

According to Chapter 4, abstraction used in abstract model checking requires some conditions. These conditions have to be formally proved if the correctness of the entire process of verification is to be rigorously guaranteed. In Chapter 8, we formally prove the validity of the abstraction used in Chapter 3 along with the results of Chapter 4. We use the Higher Order Logic (HOL) proof assistant system for constructing the formal proof. We first formalize the concurrent GC algorithm and its abstraction on HOL. Because HOL supports ample data structures and inductive definitions, the formalization is straightforward. The required conditions are also formalized in HOL, and their proofs are constructed. The proofs are long, but almost straightforward. If we proved the correctness of the concurrent GC algorithm only with a theorem prover, we would have to identify some invariants and represent them as logical formulas. The combination of abstract model checking and theorem proving makes the entire verification much easier.

As mentioned in the previous section, theorem proving is also important as an approach to formal verification. In Chapter 4, it is used to guarantee the correctness of abstraction in abstract model checking. Although proof assistant systems like HOL have powerful automatic theorem proving functions, formal proofs should be basically constructed by hand. This means that user interfaces of proof assistant systems are important for the theorem proving approach to be practical.

In Chapter 9, we propose two user interfaces of proof assistants. One interface we propose is a textual interface for HOL. The "Computing as Editing Paradigm", which is a design principle for user interfaces, advocates

that computation should be executed by solving constraints on the text. We apply this paradigm to user interfaces of proof assistants. The paradigm is also called the "Proving as Editing Paradigm". According to the paradigm, we developed the user interface "Boomborg-HOL" to edit tactics of HOL using the Emacs. Under this interface, some intermediate subgoals can be embedded in a tactic, and the undo operation is achieved by undoing the text. The proofs in Chapter 8 were actually constructed under the Boomborg-HOL interface. Another interface is a graphical one based on commutative diagrams. Under this interface, the goal diagram is proven by pasting some diagrams in the assumption. We call the interface proof-by-pasting.

Finally, we summarize the thesis and state future work in Chapter 10.

1.3 Contribution

As a whole, the research described in this thesis contributes to expand the possibility of formal verification of software. Note that we not only applied verification techniques to existing algorithms but also pioneered an exploration technique of new algorithms.

Our first contribution is that we gave a background theory of abstract model checking. We applied the refinement theory of program semantics to abstract model checking, and gave sufficient conditions that allow abstract model checking to work.

The second contribution is that we developed a general abstraction technique of link structures using regular expressions. In software verification, it is often necessary to abstract complex data like link structures. Abstraction of link structures is not obvious, and we used regular expressions as attributes of abstract link structures. We took up several concurrent garbage collection algorithms as case studies, and did abstract model checking using abstraction of link structures. In finite model checking of them, we have to bound the size of the heap. By abstract model checking, we can verify them for any size of the heap.

Since model checking is an automatic verification technique, it should also be useful for automatic algorithm exploration. We discovered new algorithms in some case studies. This is the third contribution. Our case studies are on concurrent garbage collection and mutual exclusion algorithms.

For more effective algorithm exploration, abstraction seems to be necessary. In this thesis, we applied a symbolic method to algorithm spaces. This is the fourth contribution. We symbolically expressed the space of algorithms and verified all algorithms in the space by a single execution of model checking. As a case study, we did symbolic exploration of mutual exclusion

algorithms. We also tried dynamic abstraction during symbolic exploration. This is the first step to apply abstraction techniques to algorithm exploration.

Finally, we constructed formal proofs of the correctness of abstraction using a proof assistant system. Since user interfaces of proof assistants were poor, we proposed the "Proving as Editing" paradigm as a guideline of user interfaces for proof assistants, and formally proved the abstraction of the concurrent GC algorithm under the interface. As a result, we obtained the formal proof of the entire algorithm. This is the fifth contribution.

Chapter 2

Related Work

Needless to say, the central verification technique in this thesis is model checking. As mentioned in the previous chapter, model checking is a well-established technique for hardware verification. We outline the foundations of model checking in Section 2.1. Several important issues of model checking are also stated.

To make model checking feasible for software verification, how to reduce state spaces is the most serious issue, and a number of studies have been done to deal with the issue. We briefly summarize them in Section 2.2.

In the second subject of the thesis, we also use model checking for program synthesis. There are not many studies that have been done in this direction. In Section 2.3, we look over such studies.

In the third subject, we use theorem provers for constructing formal proofs that guarantee the correctness of abstract model checking. Since the research on theorem proving has a long history, we survey only the studies directly related to this thesis in Section 2.4.

2.1 Foundation of Model Checking

In this section, we survey the foundation of model checking [18, 45, 48, 60, 61, 79] that this thesis is based on.

Strictly speaking,"model checking" in this thesis should be called "temporal model checking". This is because temporal properties of systems are verified by "model checking". In a general procedure of model checking, the system to be checked is firstly translated into a *model*. The model of the system should reflect its temporal properties. In the next subsection, we state models of systems for model checking. Secondly, the temporal property is expressed as a *logical formula*. In the subsequent subsection, we survey

some temporal logics that are used for representing temporal properties of models. We give a semantics of them. Lastly, the model is checked whether it satisfies the formula by exhaustively exploration. In the third subsection, we survey model checking algorithms. In this thesis, safety properties and liveness properties are main properties of systems to be checked. We also describe a simple model checking algorithm for safety.

2.1.1 Model

The system to be verified have to be translated into a model. To verify a temporal property of the system afterward, the translated model have to reflect its temporal properties. The most typical such model is a state transition system. A snapshot of the system corresponds to a state. A transition between states represents a change of the system by time elapse. State transition systems are formally defined as follows.

Definition 1 *A state transition system is a tuple $\langle S, \rightarrow, i \rangle$, where S is a set of states, $\rightarrow \subset S \times S$ is a set of transitions, and $i \in S$ is the initial state.*

If states $s_1, s_2 \in S$ are in the transition relation $\rightarrow$, we write $s_1 \rightarrow s_2$. s_1 is the source and s_2 is the destination of the transition $s_1 \rightarrow s_2$.

Because the system may behave nondeterministically, a state can make transitions to multiple destinations. Our state transition system is often called a nondeterministic state transition system. The state space S may contain infinite states.

To describe properties of the system, atomic propositions each state holds are needed. A model that is a state transition system with such information is called as a Kripke structure. The definition of a Kripke structure is as follows.

Definition 2 *A Kripke structure is a tuple $\langle T, P, L \rangle$, where $T = \langle S, \rightarrow, i \rangle$ is a state transition system, P is a set of atomic propositions, and L is a labeling function from S to 2^P.*

To examine properties of infinite runs in the system, a Kripke structure may be restricted to non-terminating, i.e., every state has at least one destination.

In the models we mentioned above, the time is not continuous. There are some models which has continuous time. A timed automaton is a typical such model.

Definition 3 *A timed automaton is a 5-tuple $\langle S, i, X, I, T \rangle$, where S is a set of locations, $i \in S$ is the starting location, X is a set of clocks, $I : S \rightarrow C(X)$ is a mapping from locations to clock constraints, $T \subset S \times C(X) \times 2^X \times S$ is a*

set of transitions. Each transition have clock constraints and a set of clocks that are reset when the transition is executed.

When we regard a timed automaton as a state transition system, a state is a tuple of location and values of clocks. So the number of states is infinite.

2.1.2 Logic

Temporal properties of a state transition system or a Kripke structure are represented by a formal language. Such a language called as temporal logic. There are many kinds of temporal logics. We survey a few major temporal logics.

To shorten the explanation, we first survey a powerful logic called CTL* [27].

Definition 4 *Formulas of CTL* are state formulas or path formulas. State formulas are defined as follows.*

- *p is a state formula if p is atomic formula.*
- *$\neg f$, $f \wedge g$,$f \vee g$ are state formulas if f and g are state formulas.*
- *$\mathbf{A}f$, $\mathbf{E}f$ are state formulas if f is a path formula.*

Path formulas are defined as follows.

- *f is a path formula if f is a state formula.*
- *$\mathbf{X}f$,$\mathbf{G}f$,$\mathbf{F}f$,$f\,\mathbf{U}\,g$ are path formulas if f and g are path formulas.*

We define the semantics of CTL* with respect to a Kripke structure. The semantics of a state formula is defined as a state of a Kripke structure, and the semantics of a path formula is defined as a path in a Kripke structure. A path π is an infinite sequence of states $\pi_0, \pi_1, \ldots$ such that $\pi_k \rightarrow \pi_{k+1}$ for any k. We use π^k to denote the sub-path $\pi_k, \pi_{k+1}, \ldots$. The semantics is defined as follows.

Definition 5 *Let $\mathcal{M}$ be a Kripke structure, and $s \in S$ be a state. We define a judgment whether a CTL* state formula f holds at a state s in $\mathcal{M}$. We denote this judgment by*

$$\mathcal{M}, s \models f.$$

Similarly, we define a judgment whether a CTL* path formula f holds on a path π in $\mathcal{M}$.

$$\mathcal{M}, \pi \models f.$$

They are defined by structural induction on formula.

- $\mathcal{M}, s \models p \stackrel{\text{def}}{\Leftrightarrow} p \in L(s)$.
- $\mathcal{M}, s \models \neg f \stackrel{\text{def}}{\Leftrightarrow} \mathcal{M}, s \not\models f$.
- $\mathcal{M}, s \models f \wedge g \stackrel{\text{def}}{\Leftrightarrow} \mathcal{M}, s \models f$ *and* $\mathcal{M}, s \models g$.
- $\mathcal{M}, s \models f \vee g \stackrel{\text{def}}{\Leftrightarrow} \mathcal{M}, s \models f$ *or* $\mathcal{M}, s \models g$.
- $\mathcal{M}, s \models \mathbf{A}f \stackrel{\text{def}}{\Leftrightarrow}$ *for all path* π, $\pi_0 = s$ *implies* $\mathcal{M}, \pi \models f$.
- $\mathcal{M}, s \models \mathbf{E}f \stackrel{\text{def}}{\Leftrightarrow}$ *there is a path* π *such that* $\pi_0 = s$ *and* $\mathcal{M}, \pi \models f$.
- $\mathcal{M}, \pi \models f \stackrel{\text{def}}{\Leftrightarrow} \mathcal{M}, \pi_0 \models f$ *when* f *is a state formula.*
- $\mathcal{M}, \pi \models \mathbf{X}f \stackrel{\text{def}}{\Leftrightarrow} \mathcal{M}, \pi^1 \models f$.
- $\mathcal{M}, \pi \models \mathbf{G}f \stackrel{\text{def}}{\Leftrightarrow}$ *for all* k, $\mathcal{M}, \pi^k \models f$.
- $\mathcal{M}, \pi \models \mathbf{F}f \stackrel{\text{def}}{\Leftrightarrow}$ *there exists* k *such that* $\mathcal{M}, \pi^k \models f$.
- $\mathcal{M}, \pi \models f \,\mathbf{U}\, g \stackrel{\text{def}}{\Leftrightarrow}$ *there exists* k *such that* $\mathcal{M}, \pi^k \models g$ *and* $\mathcal{M}, \pi^j \models f$ *for all* $j < k$.

We say that a CTL* state formula f holds on a model $\mathcal{M}$ if $\mathcal{M}, i \models f$.

There are some frequently used sub-logics of CTL*. Computation tree logic (CTL) [6] is a typical branching-time temporal logic. Formulas of CTL are obtained by restricting path formulas of CTL*. Here is the precise definition of CTL formulas.

Definition 6 *Formulas of CTL are defined as follows.*

- p *is a formula if* p *is an atomic formula.*
- $\neg f$, $f \wedge g$, $f \vee g$ *are formulas if* f *and* g *are formulas.*
- $\mathbf{AX}f$, $\mathbf{EX}f$, $\mathbf{AG}f$, $\mathbf{EG}f$, $\mathbf{AF}f$, $\mathbf{EF}f$, $\mathbf{A}[f \,\mathbf{U}\, g]$, $\mathbf{E}[f \,\mathbf{U}\, g]$ *are formulas if* f *and* g *are formulas.*

Another typical sub-logic is the Linear-time temporal logic (LTL) [74]. A formulas of LTL describes a temporal property of a execution sequence of a system.

Definition 7 *LTL formulas are inductively defined as follows.*

- p *is formula if* p *is atomic formula.*

- $\neg f$, $f \wedge g$, $f \vee g$ *are formulas if* f *and* g *are formulas.*
- $\mathbf{X}f$, $\mathbf{G}f$, $\mathbf{F}f$ *are formulas if* f *is formula.*
- $f\,\mathbf{U}\,g$ *is formula if* f *and* g *is formula.*

Historically, CTL and LTL were developed first, then Emerson and Halpern invented CTL*.

Another sub-logic of CTL* (CTL) is ACTL* (ACTL respectively) in which only universal path quantifiers **A** are allowed and the negation operator is allowed only on atomic formula.

The most important properties of systems are "safety" and "liveness". A safety property is roughly described as "a system never fall into any bad states". In CTL*, a safety property is represented by $AG\phi$. Therefore, safety properties are represented in sub-logics we mentioned above. A liveness property is roughly described as "a system will eventually reach some desirable states". Liveness properties are properties of an execution of a system. Therefore, LTL is suitable for representing liveness properties. In LTL, a liveness property is generally represented by $GF\phi$. A simple liveness property can be represented in CTL, such as $AGAF\phi$, but a liveness property with fairness conditions can not be represented in CTL. A fairness condition is roughly described as "some desirable states happen infinitely often on every computation path". A fairness condition also has a $GF\psi$ form. A liveness property with a fairness condition is represented by $GF\psi \Rightarrow GF\phi$. It can not be represented in CTL.

2.1.3 Algorithm

For any CTL* formula f and any finite Kripke structure $\mathcal{M}$, there exist some decision procedures whether $\mathcal{M} \models f$. Since a naive algorithm is very slow, improvement of algorithms is an important issue for model checking. In this subsection, we survey several algorithms of model checking.

The labeling algorithm was developed for model checking of CTL formulas. In this algorithm, sub-formulas of a given CTL formula are used as labels. These labels are inductively labeled to states of a Kripke structure. If a state is labeled by a sub-formula, this sub-formula holds at the state. Therefore, when this labeling procedure finished, if the initial state is labeled by the given full formula, the Kripke structure holds this formula. Otherwise, it does not hold.

The model checking using ordered binary decision diagrams (OBDDs) is called the symbolic model checking. When the states of a Kripke structure are represented by boolean values, every set of states can be represented

```
U = empty-set;
NFS(i);
terminate with success;

procedure NFS(s)
        if s is not safe then abort with error ;
        add s to U;
        for all successors s' of s do
                if s' is not in U then NFS(s');
end procedure
```

Figure 2.1: Depth First Search algorithm for a safety property

by a boolean function. The labeling algorithm also can be represented by boolean operations of boolean functions. OBDDs are efficient data structures to represent boolean functions and to calculate binary operation of them. Therefore, symbolic model checking is very efficient.

For LTL, the tableau method was developed. A state transition system which is called tableau is constructed from a given LTL formula. A given Kripke structure and the tableau are synchronously composed. By investigating this composed system, we can automatically check whether the given formula holds on the given Kripke structure. By using tableau method, we can verify a Kripke structure on the fly.

There are algorithms using automata. A Kripke structure can be transformed to an automaton. Model checking of automaton is also developed. The Büchi automaton plays an important role. The efficient double depth first search is used in this model checking.

A safety property of a state transition system can be verified by simple state exploration. Figure 2.1 is an algorithm using the depth first search to verify a safety.

2.2 Reducing State Space

Although model checking algorithms were improved, the state space explosion is still serious problem. Especially a model of software easily has huge or infinite states. Hence reducing the state space is necessary for model checking of software. In this section, we survey existing state space reduction techniques. They are classified into two types. The first type of the reduction is dependent on the property of the structure of the state transition system. The order of transitions may be independent of the property of the system.

By using these independency, the state transition system can be reduced. We regard this type of reduction as an *abstraction of the control part* of a system. The second type is dependent on the representation of a state. In the model of software, the state may be represented by the values of variables. The values are some data. By abstracting these data, the model can be reduced. We call this technique as a *data abstraction*.

2.2.1 Abstraction of Control Part

The number of states of a concurrent system is exponential to that of processes. That causes state explosion. But the execution order between processes is independent of some properties of the entire system. In such a case, exchanged transitions can be reduced. The partial order reduction [32, 33, 72] is such a reduction method. This is implemented in SPIN [46] model checker. In the partial order reduction, we consider an *ample* subset of the transitions. We can get a reduced transition system whose transitions are ample. If an ample subset holds some conditions, the reduced system preserve temporal properties of the original system. By the partial order reduction, every formula of LTL_{-X}, which is LTL but the next operator is not allowed, is preserved. An ample set can be effectively calculated in some kinds of concurrent systems. The partial order reduction is effective in a such case.

Another reduction technique of control part uses a symmetry of the state transition system [19, 28]. A symmetry may appear in concurrent systems. By the symmetry, several states are equivalent in a sense. The quotient system by this equivalence is bisimular to the original system. Therefore, every CTL^* formula is preserved in the quotient system.

Reductions of the control part of a concurrent system effectively work. But when the representation of a state of system contains a value of a variable, as in software, the space of possible values can not be reduced by abstractions of the control part. Therefore, abstractions of the control part is not enough for reducing the state space of software.

2.2.2 Data Abstraction

In order to reduce the state space of a model of software, abstraction of data part is necessary. The data abstraction is a main subject of this thesis.

The state of the model of software can be represented by the values of variables. The values in software may contain integer or real, so the number of states may be very large or infinite. By using abstract data, we can construct an abstract model which is small enough to make model checking feasible. This technique is researched by several researchers [17, 23, 24, 58, 82, 83].

The fundamental idea of this technique is the same as the abstract interpretation [22]. The abstract interpretation is proposed in the field of data-flow analysis. In the abstract interpretation, there is a Galois connection between the concrete data and the abstract data. In data-flow analysis, the semantics of a program and a Galois connection derive a abstract semantics of the program. There are some differences between data-flow analysis and abstract model checking. In data-flow analysis, the abstract domain is determined from the properties to be examined. Because the user knows what abstract semantics he requires. But in abstract model checking, the choice of abstract domain is not determined in advance. If abstract domain is too small, abstract model checker reports a failure even if the original system satisfy the property. If abstract domain is too large, abstract model checker does not report any results in a realistic time. The user may have to find an appropriate abstract domain by a trial-and-error method.

The abstract model checking in this thesis is a kind of this abstraction, because we want to reduce the state spaces of software. Such abstract model checking is investigated by many works [17, 23, 57]. Clarke, Grumberg and Long [17] showed a condition that abstract model preserve ACTL* formulas. We also show a condition that abstract model preserves the safety. Their conditions are almost equivalent to ours. The detailed difference is discussed in Chapter 4.

What kinds of data should be abstracted, and how the data should be abstracted? That is a practical problem in software abstract model checking. In [17], some examples are explained: integers are abstracted to "*mod* n" numbers, integers are abstracted by their logarithm representations, any data are abstracted to a symbol. How about complex structures? The abstraction by a symbol is too abstract in many cases. There is a few methods for them. For queue data structure, Boigelot [10, 11, 12] developed queue decision diagrams for finite representation of (infinite) queue. In this thesis, we introduce an abstraction method of link structures which is one of the most complex data structures in software. For link structure data, size independent properties of link structures that programs operate on have been investigated under the name of shape-analysis [81]. But their analysis mention only shapes of link structure such as list, tree, or cycle. So shape-analysis can not work on the analysis of the heap appeared in garbage collection. In our abstraction, the contents of cells of link structure can be handled. We show the safety of garbage collection by our abstraction technique in this thesis.

2.3 Synthesis of Models

The automatic synthesis of model is another important subject. There are some kinds of approaches.

Deriving a model from a given temporal specification is one approach. Many temporal logics have finite model property, i.e., there always exists finite model that satisfies a given formula. The derivation of the finite model is based on tableau construction. In some research [16, 26, 75], a skeleton of each process is calculated from a temporal specification assuming several processes run asynchronously.

Another approach of automatic synthesis is automatic filtering of the candidates by model checking. There are some researches using similar approaches. Superoptimization [64] is an automatic discovery technique of machine code. Perrig and Song [73] recently used Song's protocol verifier, Athena [87], to try to discover symmetric-key and asymmetric-key mutual authentication protocols that satisfy the agreement property. The details of these two automatic discovery techniques are stated in Chapter 6.

Discovery of appropriate parameter values of a parameterized model can be regarded as a kind of program synthesis. There are some works using this approach for real-time model such as timed automata [2, 47].

2.4 Formal Verification

Formal verification is important even if abstract model checking can work. Because the validity of the abstraction have to be proved formally. So theorem provers and their interfaces are important. We briefly survey them in this section. The relation between theorem provers and model checker is stated. We overview interfaces for theorem proving because we propose new interfaces in this thesis.

2.4.1 Theorem Prover

Many theorem provers are developed. ACL2 [52] is a semi automatic theorem prover of quantifier free first order logic. HOL [35] is a kind of LCF style theorem prover of a higher order logic. In HOL, the consistency is kept by hiding the axioms and inference rules with abstract data types in the meta language, and the proof can be constructed by backward reasoning. Isabelle [71] is a meta logic which can describe logics to prove. Isabelle/HOLCF, Isabelle/FOL, and Isabelle/ZF are instances. Coq [3] is a prover of calculus of inductive constructions. PVS [70] is a theorem prover

that actively takes in many decision procedures. PVS takes a model checker as a decision procedure. The relation of theorem provers and model checker is surveyed in the next subsection.

2.4.2 Combining Model Checker and Theorem Prover

Model checking and theorem proving have complementary nature. Model checkers are automatic tools, but they have a limitation on the scale of the problems. On the other hand, theorem provers need tremendous human interaction, but they can handle large or infinite systems. To complement the other's weakness, integration of model checking and theorem proving have been studied in several contexts [1, 50, 63, 78, 84]. These works combine two kinds of verification in several ways: using a model checker as a decision procedure inside a theorem prover, decomposing a large proposition into smaller ones with a theorem prover so that a model checker can handle them, proving abstractions correct with a theorem prover and verifying the abstracted system with a model checker, or doing local reasoning with a theorem prover and deriving global consequences with a model checker.

Concerning combination of model checking and theorem proving, there are some works other than the integration of both verifiers: proving the correctness of a model checker with a theorem prover. Some textbooks and tutorials adopt a simple model checker as a target of theorem proving: a CTL model checker is verified with Isabelle/HOL [69], and a μ-calculus model checker is verified with ACL2 [62]. Sprenger [88] verified a model checker of μ-calculus using Winskel's local model checking algorithm with Coq. Verma [98] formalized a BDD and a symbolic model checker of μ-calculus with Coq, and obtained a program of the model checker, which is proved to be correct, using extraction. Implementation of a model checker in LEGO [100] is not a verification of a model checker, however, it generates a proof term so that the prover can verify the correctness of the result. Proving the correctness of optimized algorithms used in model checkers is a more tough challenge. Chou and Peled [15] verified partial-order reduction, which is one of the optimizations in model checkers, with HOL.

2.4.3 User Interfaces of Theorem Prover

The interface is important even if the expert uses a theorem prover. Integrated user interfaces are developed for many theorem provers [7, 9, 37, 41, 89, 96].

Merriam and Harrison [65] pointed out many problems of proving with graphical user interfaces. Therefore, character user interfaces are also impor-

tant. Editors used in many integrated interfaces are structural editor. We propose an interface Boomborg-HOL in Emacs that is a structure-free editor in Chapter 9. Detailed discussion will be held in that chapter.

A useful graphical interface is proof-by-pointing [8]. By using proof-by-pointing, a proof of propositional logic is constructed by pointing a subterm of a formula directly. We propose proof-by-pasting interface in Chapter 9. By using proof-by-pasting, diagrams are directly manipulated.

Chapter 3

Abstract Model Checking of Concurrent Garbage Collection

In this chapter, we do a case study as a touchstone for abstract model checking of software. As a practical and non-trivial example, we verify algorithms for concurrent garbage collection (GC) by using finite model checking and abstract model checking.

The concurrent GC algorithms are complex and their correctness is not at all obvious. On the other hand, concurrent GC is becoming more and more important not only due to the wide use of parallel machines but also because of the need for collecting objects under distributed environments.

In general, theorem proving of the correctness of concurrent GC is a hard work. We survey the research on formal verification of concurrent GC algorithms in the next section.

In Section 3.2, we introduce a modified model of the concurrent GC algorithms. We state the properties of the concurrent GC to verify in Section 3.3. Finite model checking works when the size of the heap is small as mentioned in Section 3.4, and for any size of heap, we have to abstract the heap. We explain the abstraction of the heap and abstract model checking in Section 3.5. To define abstract transitions, we introduce a rather tricky procedure called "filter". In Section 3.6, we discuss the problems raised through this case study.

(The contents of this chapter are published in [39].)

3.1 Related work

There have been spent substantial efforts on formally verifying the correctness of garbage collection algorithms [34, 42, 43, 49, 80] using theorem proving

systems.

Russinoff's proof [80] in the Boyer-Moore prover is the first formal proof. He proved the safety and the liveness. Havelund proved the safety in PVS [42, 43]. He used refinement techniques. Jackson proved the safety and the liveness in PVS [49]. He made the linear temporal logic framework in PVS. These formal proofs did not employ model checking.

Havelund verified a finite state version in the model checker Murphi in [42].

The study presented in this chapter differs from the existing ones in that it aims to verify concurrent garbage collectors using abstract model checking. The merits of this approach are as follows.

- Most parts of verification are fully automatic.
- After appropriate abstractions are defined, abstract model checking with that abstraction can verify not only one algorithm but also its many variations as far as the abstractions can be applied.

3.2 The Model

We deal with two major algorithms for concurrent garbage collection. One is the so-called *on-the-fly garbage collector* formulated by Dijkstra *et al.* [25]. The other is the *snapshot garbage collector* by Yuasa [101, 102], modified to fit our model.

3.2.1 Cells and Registers

The heap for cells is formally defined as a function (or an array) from cell indices to cells. We do not specify the set of cell indices at the moment. We use `C` to denote the heap, and `C[i]` to denote the `i`-th cell.

```
C : heap = cell_index -> cell
```

A cell consists of its color and its fields that hold pointers to other cells. A pointer to a cell is a cell index or the null pointer `nil`. In this study, for the sake of simplicity, a cell has only one field. The set of cells, denoted by `cell`, is defined as follows.

```
cell = color * (cell_index + {nil})
color = {free, white, gray, black}
```

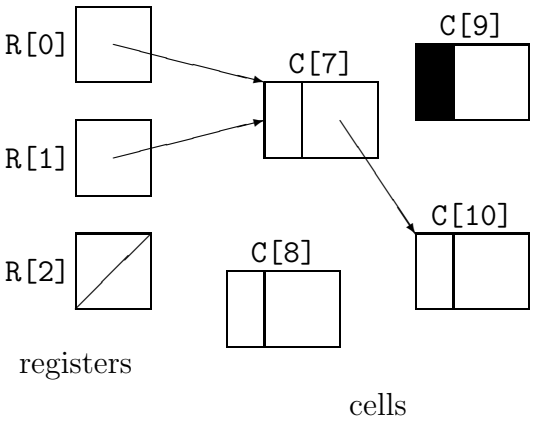


Figure 3.1: registers and cells

In the following, the pointer stored in the unique field of the `i`-th cell is denoted by `C[i].f`.

In this model, a list of free cells is not maintained, but a free cell has the color `free`. By this simplification, a cell has one of the four colors.

Registers are used by the mutator for manipulating cells. They are also used by the collector as the root for marking cells. In order to make the model closer to real implementations, we introduced a set of registers. In the following, `R` denotes a function (or an array) from register indices to registers, and `R[i]` the `i`-th register. Each register holds a pointer to a cell.

```
R : register
register = register_index -> (cell_index + {nil})
```

A cell is called *directly reachable* if its index is stored in some register. A cell is called *reachable* if it is accessible through a chain of pointers from some register.

3.2.2 Mutator

The mutator has operations in Table 3.1. Each operation modifies the value of a register or the field of a cell. Some operations also change the color of a cell as specified in the comments in Table 3.1. The comments in the second column are for the on-the-fly garbage collector, while those in the third are for the snapshot garbage collector. The expressions, `R[i]`, `C[R[i]].f` and `C[R[j]].f`, in the comments denote the values stored in a register or a cell field before the operation is executed, i.e., before the assignment.

	on-the-fly	snapshot
`R[i] := nil`		
`C[R[i]].f := nil`		If `C[C[R[i]].f]` is white, make it gray.
`R[i] := j`	(Only when `C[j]` is free.) Make `C[j]` gray.	(Only when `C[j]` is free, and the collector is not in `shade`.) Make `C[j]` black.
`R[i] := R[j]`		
`R[i] := C[R[j]].f`	If `C[C[R[j]].f]` is white, make it gray.	
`C[R[i]].f := R[j]`		If `C[C[R[i]].f]` is white, make it gray.

Table 3.1: mutator operations

These operations are indivisible in that each causes a single state transition in the entire system.

3.2.3 Collector

The collector is shared by the on-the-fly garbage collector and the snapshot garbage collector.

The collector has four states. In order to distinguish the state of the collector from that of the entire system, the collector state is called the *collector step.* The collector has four steps: `shade`, `mark`, `append` and `unmark`. The operations allowed in each step are summarized in Table 3.2.

Notice that the entire shading operation is indivisible. It makes all the white and directly reachable cells gray at once. The snapshot garbage collector does not allow allocation of a new cell during the `shade` step. This means that allocation is prohibited at the instance after unmarking finishes and before shading begins.

3.3 Properties

The properties we should prove is:

(safety) In each state (that is reachable from the initial state), there is no free cell that is reachable from a register.

(liveness) Cells that are not reachable eventually become free.

`shade`	Make each directly reachable cell gray, if it is white. Then, go to the `mark` step.
`mark`	Choose a gray cell and make it black. If the cell refers to a white cell, make it gray.
`mark`	If there is no gray cell, go to the `append` step.
`append`	Choose a white cell and make it free.
`append`	If there is no white cell, go to the `unmark` step.
`unmark`	Choose a black or gray cell and make it white.
`unmark`	If there is no black or gray cell, go to the `shade` step.

Table 3.2: collector operations

3.4 Finite Model Checking

As Havelund [42] and Bruns [13] have done previously, we first performed finite model checking on our model of concurrent garbage collection. We set `register_index` and `cell_index` as follows.

```
register_index = {1, 2, 3}
cell_index = {1, 2, 3}
```

This results in a finite model with three registers and three cells. Since a pointer can be represented by two bits, 20 bits are sufficient to implement a state of the model. This allows a simple bit-state encoding.

We verified the safety of the on-the-fly and snapshot garbage collectors by a hand-crafted C program, and found that the number of reachable states from the initial state was

- 257,730 for the on-the-fly garbage collector, and
- 227,285 for the snapshot garbage collector.

In the initial state, all registers hold the `nil` pointer and all cells are free. This program for finite model checking was later extended to search for variations of the algorithms. See Chapter 6.

3.5 Abstract Model Checking

In this section, we introduce the abstractions we employed in our abstract model checking.

3.5.1 Abstract Cells

We first define abstractions of cells. An *abstract cell* is a data structure consisting of some attributes about concrete cells. In this study, we gradually increased the number of attributes of an abstract cell as we verified more and more properties of the garbage collectors. The first set of attributes is the following, which is enough for verifying the safety of the on-the-fly garbage collector:

(A1) the color of a cell,

(A2) the flag expressing whether a cell is directly reachable from a register,

(A3) the flag expressing whether a cell is (not necessarily directly) reachable from a register, and

(A4) the color of the cell that a cell refers to, or `nil`.

The first attribute is a color. The second and third are boolean values. The last attribute is either a color or the `nil` pointer. If the field of a cell holds the index of some cell, this attribute is the color of the cell having the index. Otherwise, it is `nil`.

The reader can now skip the rest of this subsection and proceed to the next one, where abstractions of heaps and entire system states are defined. In the following, we explain the attributes that are required for verifying the other properties of the garbage collectors.

The next attribute we introduced was for proving the liveness of the garbage collectors. It is

(A5) the age of a garbage cell.

When a cell becomes garbage, i.e., when a cell becomes unreachable (but has not been made free yet), it is given the age zero. As the cell survives and remains garbage after the collector changes its step, the age of the cell is increased by one.

By introducing this attribute, the set of possible abstract cells becomes infinite. This implies that model checking on the abstract system might not terminate because the set of abstract states is infinite. This situation is exactly the same as that of Müller and Nipkow's verification of alternating bit protocol [68]. If the garbage collector satisfies the liveness property, ages of cells must have some upper bound, and state exploration from the initial state must terminate. Notice that the liveness verified by this method is weaker than the ordinary one [13, 49]. We show the liveness under the assumption that the collector eventually changes its step.

The last attribute of an abstract cell is the following:

(A6) the flag whether a white cell is reachable from a gray cell through a chain of white cells.

When a cell satisfies the above attribute, it is called *reachable from gray,* or **rfg** for short. This is the most artificial attribute of an abstract cell. It reveals a limitation of our approach. See the discussion in the last section. This attribute is required for verifying the safety of the snapshot garbage collector.

3.5.2 Abstract Heaps and Abstract States

An *abstract heap* is simply a set of abstract cells. An abstract heap is an abstraction of a concrete heap, if for each concrete cell in the concrete heap, there exists an abstract cell in the abstract heap such that the concrete cell satisfies all the attributes of the abstract cell.

An *abstract state* of the entire system is a pair of a collector step and an abstract heap. Note that the values of the registers are not explicitly represented in an abstract state. They are implicit in the attributes of abstract cells.

State transitions within a same collector step does not depend on the *non-existence* of cells satisfying certain attributes. They only depend on the *existence* of some cells. Therefore, at the abstract level, they are monotone with respect to abstract heaps (where heaps are ordered by the set inclusion). Moreover, as we see in the next subsection, abstract transitions within a same collector step does not decrease abstract heaps. Therefore, we can merge the heaps of all abstract states having the same collector step. This means that we should prepare only one abstract heap for each collector step. In this sense, our abstract model checking is more like a data-flow analysis with an abstract heap assigned to each node of a control flow graph (Figure 3.2).

3.5.3 Abstract Transitions

State transitions in the concrete system should be abstracted in such a way that if a concrete transition changes state s to state s' and t is an abstraction of s, then there exists an abstract transition that changes t to some t', where t' is an abstraction of s'.

It is relatively easy to abstract the mutator's transitions and the collector's transitions within a same collector step. For example, consider the mutator operation `R[i]:=nil` of the on-the-fly garbage collector. This operation is abstracted by the following operation on an abstract heap:

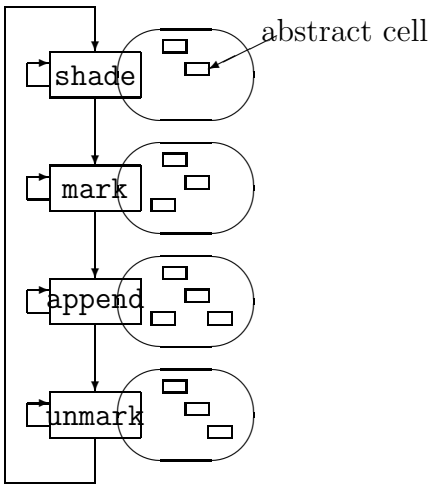


Figure 3.2: abstract states

> For each directly reachable abstract cell, add a new cell with the same attributes as those of the cell except that it is not directly reachable.
>
> For each reachable abstract cell, add a new cell with same attributes as those of the cell except that it is not reachable (and hence is not directly reachable).

Abstractions of the collector's transitions that change the collector step are not trivial. For abstracting those transitions, we introduce procedures we call filters. A *filter* is a procedure that deletes inconsistent abstract cells from an abstract heap. In this study, we define one filter for each transition that changes the collector step. We further introduce two general filters that can be combined with those filters.

The filters that are specific to some step change are listed in Table 3.3. The concrete transition from `mark` to `append`, for example, requires the condition that there is no gray cell. Therefore, its abstraction deletes gray cells from the abstract heap of the `mark` step before adding its cells to the abstract heap of the `append` step. Similarly, it deletes the cells that refer to gray cells. It also deletes cells that are **rfg** (reachable from gray).

These filters are natural abstractions of step changes. But the abstract system that adopts just these filters reaches an error state, i.e., a reachable free abstract cell is generated. The scenario is as follows. We suppose that the following two concrete cells exist in the heap of the `shade` step. One is gray and directly reachable from a register, and refers to the other cell,

`shade` → `mark`	Delete white and directly reachable cells.
`mark` → `append`	Delete gray cells, cells that refer to gray cells, and cells that are **rfg**.
`append` → `unmark`	Delete white cells and cells that refer to white cells.
`unmark` → `shade`	Delete gray cells, black cells, **rfg** cells, and cells that refer to gray or black cells.

Table 3.3: filters specific to step changes

which is white and refers to nothing. The corresponding abstract cells are $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$ and $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ (values of the tuple correspond to the attributes (A1)–(A4), respectively). Filter "`shade` → `mark`" deletes neither of them, and the abstract collector in the `mark` step generates some cells from them, but it does not delete any cell. Remember that abstract transitions other than filters do not delete abstract cells. Filter "`mark` → `append`" then deletes $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$ and some other cells, but it does not delete $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$, because this abstract cell has no gray attributes. The abstract collector of the `append` step finally generates $\langle \text{free}, \text{false}, \text{true}, \text{nil} \rangle$ from $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$. This is an unsafe cell.

In this scenario, however, the concrete cell corresponding to the abstract cell $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ should have changed its attributes during the `mark` step, and should no longer correspond to the abstract cell.

Fortunately, this inconsistency can be detected only by examining the abstract heap. The abstract cell $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ is reachable but is not directly reachable. This means that there should be some other reachable cells that refer to a white cell. In the above scenario, this is not the case because filter "`mark` → `append`" has deleted the abstract cell $\langle \text{gray}, \text{true}, \text{true}, \text{white} \rangle$.

For deleting inconsistent cells such as $\langle \text{white}, \text{false}, \text{true}, \text{nil} \rangle$ in the above scenario, we introduce two general filters. One is a filter concerning reachability. An abstract cell in an abstract heap is called *truely reachable,* if it satisfies one of the following conditions. (Notice that this is an inductive definition.)

- It is directly reachable.
- There exists another truely reachable abstract cell in the abstract heap whose color coincides with the attribute of the cell expressing *the color of the cell it refers to.*

By this filter, an abstract cell whose attribute for reachability is true is deleted

if it is not truely reachable. (Note that this notion of true reachability is considered as an approximation of the inductive notion of real reachability defined on concrete cells.)

By using this filter, the previous scenario is revised as follows. The set of abstract cells obtained by filter "`mark` → `append`" contains ⟨white, false, true, nil⟩, but it is not truely reachable in this set, because ⟨gray, true, true, white⟩ has been deleted by filter "`mark` → `append`", so it is deleted by the filter concerning reachability. More precisely, the set contains neither ⟨black, true, true, white⟩ nor ⟨black, false, true, white⟩ because the collector in the `mark` step can not generate them in case of the on-the-fly algorithm.

When we add the attribute (A6), we face another bad scenario. Let us consider an abstract heap of the `shade` step which contains an abstract cell ⟨white, false, true, nil, false⟩ (the fifth value indicates **rfg**). The collector makes directly reachable white cells gray. So a white reachable cell may become **rfg**. Therefore the abstract collector generates ⟨white, false, true, nil, true⟩. Filter "`shade` → `mark`" does not delete them. Filter "`mark` → `append`" deletes only the latter ⟨white, false, true, nil, true⟩ and leaves the former ⟨white, false, true, nil, false⟩. So a free reachable abstract cell is finally generated in the `append` step. The filter concerning reachability can not correct this scenario in case of the snapshot algorithm. So we need one more filter.

Another filter is concerning the reachability from gray cells. Remember that a white cell is **rfg** if it is reachable from a gray cell through a chain of white cells. We can easily prove the following lemma on the concrete system.

Lemma 1 *If all reachable cells are gray or white and all directly reachable cells are gray, then white reachable cells are all* **rfg**.

This filter first checks if the condition of the lemma holds in the given abstract heap. If it holds, then white reachable abstract cells that are not **rfg** are deleted.

The second bad scenario is revised as follows. The set of abstract cells filtered by "`shade` → `mark`" satisfies the condition of the lemma in case of the snapshot. Therefore the abstract cell ⟨white, false, true, nil, false⟩ is deleted by this filter.

3.5.4 Results

The abstract model checker was written in Python [97], and run on Pentium II (300MHz). See Table 3.4 for the results. The same checker was used to verify both algorithms. Therefore, an abstract cell has all the attributes described in Section 3.5.1.

	execution time (sec)	number of iterations	number of abstract cells				maximum age
			`shade`	`mark`	`append`	`unmark`	
on the fly	2.794	6	40	69	47	77	7
on the fly (variation)	2.150	5	58	69	35	73	7
snapshot	7.341	7	53	85	40	91	7
snapshot (variation)	6.003	6	53	85	52	95	7

Table 3.4: results

The checker iterates the processing of all collector steps. The *number of iterations* in Table 3.4 means the number of iterations of loop until the abstract heap of every collector step is saturated.

3.5.5 Variants

In chapter 6, we find out variant algorithms. By using the same abstraction, we could check them. The result is shown in Table 3.4. We can do abstract model checking by one uniform abstraction for several algorithms.

3.6 Discussion

In this chapter, we did abstract model checking of the concurrent GC algorithms. Through this case study, we examined that abstract model checking could work for such a complex algorithm. But some problems also arose.

One problem is on the validity of abstract model checking. The safety of the original model may fail even if the abstract model satisfies the safety. In Section 3.5, we discussed this problem, but the discussions there were rather informal. We should theoretically analyze the problem, and in the next chapter, we will make such theoretical analysis.

Another (more serious) problem is on the construction method of abstract cells and abstract transitions. We used attributes to define abstract cells. The four attributes were enough for the abstract model checking of the on-the-fly algorithm. But the fifth attribute was needed for the snapshot algorithm. When we want to verify another algorithm, we may have to find other attributes. Therefore, a smart representation of attributes is required. We also defined some filters to construct the abstract transitions. We may have to consider other filters when other attributes are introduced.

Therefore, a systematic construction of abstract transitions are needed. This problem is dealt with in Chapter 5.

Chapter 4

Validity of Abstract Model Checking

One problem of abstract model checking is that it does not necessarily hold that the safety property of an abstract system *implies* that of the concrete system.

For example, in Chapter 3, we verified safety properties of concurrent garbage collection algorithms using the technique of abstract model checking. There, the abstract states and transitions were defined, and then the safety property of the abstract system was verified by means of model checking. The validity of the implication mentioned in the previous paragraph, however, was not proved: namely, there was no proof that the safety property of the abstract system implies that of the concrete one.

In this chapter, we show that a safety property of an abstract system implies the same safety property of a concrete system, based on the theory of refinement. To that end, we use the formulation of Kinoshita and Power [53], where they developed a theory of refinement for while commands with nondeterministic choice. Since their theory is for imperative programs and treats only input-output relations, safety properties are out of their scope. So, we introduce the notion of 'semantic cumulative,' which informally is the collection of all states that have appeared *during* the execution of the program so far. Using that notion, one can state that a program never falls into an unsafe state, and show the safety property.

(The contents of this chapter are published in [54].)

4.1 Related work

Data abstractions of model checking have been studied in [17, 23, 24, 82, 83]. The validity of them also studied, but there is no research that uses program semantics. Clarke, Grumberg and Lang [17] showed a condition that the abstract model preserves safety. That condition is almost equivalent to our resulting condition. Only difference is that the condition with respect to abstract and concrete transitions. Their condition is represented by full transition relations. Our condition is represented by atomic transition relations. Though both condition is semantically equivalent, our condition is more convenient to prove.

Havelund and Shankar [43] formally proved a safety of a concurrent garbage collection program by refinement. They state the safety property by a specification (an abstract program), and repeatedly refine it, until the concrete program is obtained. Their setting does not have the notion of validity of abstract model checking because the safety of states are not considered. Therefore, nontrivial invariants have to be discovered at each refinement step in their formal proof.

4.2 Sequential μ-calculus

We take a variant of the μ-calculus as the calculus which models imperative programming languages. Given a set of atoms A and a set of variables X, a term or program P of μ-calculus is defined by the following BNF:

$$P ::= A \mid \texttt{skip} \mid \texttt{abort} \mid P; P' \mid P + P \mid \bigvee\{P_i | i \in \omega\}$$

where ω is the set of all natural numbers.

";" is a concatenation of program, "+" is a nondeterministic choice of program, and "$\bigvee$" allows us to talk about least fixed points which is necessary to model while commands. To formulate these meanings, a preorder $\leq$ between terms is defined to be the preorder generated by the following rules.

- $\texttt{skip}; P \leq P$, $P \leq \texttt{skip}; P$, $P; \texttt{skip} \leq P$, $P \leq P; \texttt{skip}$.
- $\texttt{abort} \leq P$.
- $(P; P'); P'' \leq P; (P'; P'')$, $P; (P'; P'') \leq (P; P'); P''$.
- $P' \leq P'' \Rightarrow P; P' \leq P; P''$. $P' \leq P'' \Rightarrow P'; P \leq P''; P$.
- $P \leq P + P'$, $P' \leq P + P'$.

- $P \leq P'' \wedge P' \leq P'' \Rightarrow P + P' \leq P''$.
- $P; (P' + P'') \leq (P; P') + (P; P'')$, $(P; P') + (P; P'') \leq P; (P' + P'')$.
- $(P' + P''); P \leq (P'; P) + (P''; P)$, $(P'; P) + (P''; P) \leq (P' + P''); P$.
- $(\forall i \in \omega) P_i \leq \bigvee \{ P_i \mid i \in \omega \}$.
- $((\forall i \in \omega) P_i \leq P) \Rightarrow \bigvee \{ P_i \mid i \in \omega \} \leq P$.
- $P; \bigvee \{ P_i \mid i \in \omega \} \leq \bigvee \{ P; P_i \mid i \in \omega \}$, $\bigvee \{ P; P_i \mid i \in \omega \} \leq P; \bigvee \{ P_i \mid i \in \omega \}$. $(\bigvee \{ P_i \mid i \in \omega \}); P \leq \bigvee \{ P_i; P \mid i \in \omega \}$, $\bigvee \{ P_i; P \mid i \in \omega \} \leq (\bigvee \{ P_i \mid i \in \omega \}); P$.

As usual, the preorder $\leq$ induces a partial order, which we also write $\leq$, on the quotient set $P/\equiv$, where the equivalence relation $\equiv$ is defined by $x \equiv y \iff x \leq y \wedge y \leq x$. In fact, $\equiv$ plays a role in the μ-calculus which corresponds to the $\beta\eta$-equivalence in the λ calculus.

If a term B has a term $\neg B$ such that

- $\texttt{skip} \leq B + \neg B$ and
- the greatest lower bound of B and $\neg B$ is $\texttt{abort}$,

we call B a *condition term*. For a condition term B, the `if` and `while` statements can be encoded as follows.

- $\texttt{if}\ B\ \texttt{then}\ P\ \texttt{else}\ P' = (B; P) + (\neg B; P')$.
- $\texttt{while}\ B\ \texttt{do}\ P = \bigvee \{ F^n(\texttt{abort}) \mid n \in \omega \}$, where

$$F(X) := \texttt{if}\ B\ \texttt{then}\ (P; X)\ \texttt{else}\ \texttt{skip}.$$

4.3 Interpretation

We review the interpretation of the calculus introduced in the previous section. The contents of this section are found in [53]. Let Σ be a set; we say its elements are *states*. We write $\mathcal{R}(\Sigma)$ for the set of all binary relations on Σ. A map $[\![-]\!]\colon A \to \mathcal{R}(\Sigma)$ taking an atom a to a binary relation on Σ is defined to be an *interpretation* of the calculus with atoms in A. For each $a \in A$ we write $[\![a]\!]$ for its image under the interpretation $[\![-]\!]$. The interpretation $[\![-]\!]$ is extended inductively to the whole language of the calculus as follows.

- $[\![\texttt{skip}]\!] \stackrel{\text{def}}{=} \Delta$,

- $[\![\texttt{abort}]\!] \stackrel{\text{def}}{=} \emptyset$,
- $[\![P; P']\!] \stackrel{\text{def}}{=} [\![P]\!]; [\![P']\!]$,
- $[\![\bigvee P_i]\!] \stackrel{\text{def}}{=} \bigcup [\![P_i]\!]$

where

- $\Delta \stackrel{\text{def}}{=} \{\, (x, x) \mid x \in \Sigma \,\}$,
- $\emptyset$ is the empty set, so $(\forall x, y \in \Sigma)(x, y) \notin \emptyset$,
- $(R_1; R_2)$ is the usual composition of relations R_1 and R_2, so $(x, y) \in (R_1; R_2) \stackrel{\text{def}}{\Leftrightarrow} (\exists z)(x, z) \in R_1 \wedge (z, y) \in R_2$,
- $\bigcup P_i$ is the union of a family P_i of sets, so $(x, y) \in \bigcup P_i \stackrel{\text{def}}{\Leftrightarrow} (\exists j)(x, y) \in P_j$.

Because $\bigvee$ has countably many arguments, these steps must be applied κ times for any countable ordinal κ, not only finitely many times as in the usual inductive definition, where all operators have only finite number of arguments.

The mapping $[\![-]\!]$ is monotone in the sense that

$$P \leq P' \Longrightarrow [\![P]\!] \subseteq [\![P']\!].$$

If an execution of a program P starts at the state x, we say "P *can* terminate at the state y," iff

$$(x, y) \in [\![P]\!].$$

So, we call $[\![P]\!]$ the input-output relation of P.

In the introduction, we spoke of abstract programs and concrete programs, but we consider that they really are abstract interpretations and concrete interpretations of the same program. So, we have two interpretations: an abstract interpretation $[\![-]\!]_A$ and a concrete interpretation $[\![-]\!]_C$. For simplicity, we assume the set of states Σ is the disjoint sum of the set of abstract states Σ_A and that of concrete states Σ_C, so $\Sigma = \Sigma_A + \Sigma_C$. Both interpretations $[\![-]\!]_A$ and $[\![-]\!]_C$ have by definition the same codomain $\mathcal{R}(\Sigma)$, but $[\![-]\!]_A$ factors through the injection $\iota_A \colon \Sigma_A \to \Sigma$ and $[\![-]\!]_C$ factors through the injection $\iota_C \colon \Sigma_C \to \Sigma$.

Now, let α be a relation between Σ_A and Σ_C; modulo injections, we regard α to be a binary relation on Σ, so an element of $\mathcal{R}(\Sigma)$.

Definition 8 *An interpretation* $[\![-]\!]_A$ *is an* abstraction *of an interpretation* $[\![-]\!]_C$ *with respect to an relation* α*, if and only if*

$$(\alpha; [\![a]\!]_C) \subseteq ([\![a]\!]_A; \alpha)$$

holds for every atom $a \in A$*.*

$$\begin{array}{ccc} \Sigma & \xrightarrow{[\![a]\!]_A} & \Sigma \\ {\scriptstyle\alpha}\downarrow & \bigcup & \downarrow{\scriptstyle\alpha} \\ \Sigma & \xrightarrow{[\![a]\!]_C} & \Sigma \end{array}$$

These diagrams are extended to programs.

Lemma 2 *If an interpretation* $[\![-]\!]_A$ *is an abstraction of an interpretation* $[\![-]\!]_C$ *with respect to an abstract relation* α*, then*

$$(\alpha; [\![P]\!]_C) \subseteq ([\![P]\!]_A; \alpha)$$

holds for every program P*.*

$$\begin{array}{ccc} \Sigma & \xrightarrow{[\![P]\!]_A} & \Sigma \\ {\scriptstyle\alpha}\downarrow & \bigcup & \downarrow{\scriptstyle\alpha} \\ \Sigma & \xrightarrow{[\![P]\!]_C} & \Sigma \end{array}$$

This extension can uniquely be done by the theorem of Kinoshita-Power [53]. But the uniqueness is not necessary in our discussion, we give an elementary proof of this extension.

If the program is an atom or `skip` or `abort`, the statement is obvious. When the program is a concatenation of two programs, the next proves.

$$\begin{array}{rcl} \alpha; [\![P; P']\!]_C & = & \alpha; ([\![P]\!]_C; [\![P']\!]_C) \\ & = & (\alpha; [\![P]\!]_C); [\![P']\!]_C \\ & \subseteq & ([\![P]\!]_A; \alpha); [\![P']\!]_C \\ & = & [\![P]\!]_A; (\alpha; [\![P']\!]_C) \\ & \subseteq & [\![P]\!]_A; ([\![P']\!]_C; \alpha) \\ & = & [\![P; P']\!]_A; \alpha. \end{array}$$

When the program is a least upper bound, the next proves.

$$\begin{aligned}
\alpha; [\![\bigvee P_i]\!]_C &= \alpha; (\bigcup [\![P_i]\!]_C) \\
&= \bigcup (\alpha; [\![P_i]\!]_C) \\
&\subseteq \bigcup ([\![P_i]\!]_A; \alpha) \\
&= (\bigcup [\![P_i]\!]_A); \alpha \\
&= [\![\bigvee P_i]\!]_A; \alpha.
\end{aligned}$$

The lemma has been proved.

4.4 Safety

In this section, we define safety properties of programs and the 'cumulative' of programs. We show the equivalence of a safety property of program and a formula by using cumulative.

A predicate ϕ which resolve a state is safe or not is given.

Definition 9 *The safety predicate $\Phi(P, \sigma)$ of a program P from a state σ is inductively defined as follows.*

- $\Phi(a, \sigma) \stackrel{\text{def}}{=} \phi(\sigma) \wedge (\forall \sigma')(\sigma, \sigma') \in [\![a]\!] \Rightarrow \phi(\sigma')$,
- $\Phi(\texttt{skip}, \sigma) \stackrel{\text{def}}{=} \phi(\sigma)$,
- $\Phi(\texttt{abort}, \sigma) \stackrel{\text{def}}{=} \phi(\sigma)$,
- $\Phi(P; P', \sigma) \stackrel{\text{def}}{=} \Phi(P, \sigma) \wedge (\forall \sigma')(\sigma, \sigma') \in [\![P]\!] \Rightarrow \Phi(P', \sigma')$,
- $\Phi(\bigvee P_i, \sigma) \stackrel{\text{def}}{=} (\forall i)\Phi(P_i, \sigma)$.

The safety predicate $\Phi(P, \sigma)$ returns true if any execution of the program P from the state σ never fall in unsafe states.

We define the 'cumulative' to describe the safety property by terms of binary relations.

Definition 10 *The cumulative $(\!|-|\!)$ with respect to an interpretation $[\![-]\!]$ is inductively defined as follows.*

- $(\!|a|\!) \stackrel{\text{def}}{=} [\![a]\!]$ *($a \in A$)*,
- $(\!|\texttt{skip}|\!) \stackrel{\text{def}}{=} \Delta$,

- $(\!|\texttt{abort}|\!) \stackrel{\text{def}}{=} \emptyset$,
- $(\!|P; P'|\!) \stackrel{\text{def}}{=} (\!|P|\!) \cup ([\![P]\!]; (\!|P'|\!))$,
- $(\!| \bigvee P_i |\!) \stackrel{\text{def}}{=} \bigcup (\!|P_i|\!)$.

By the definition, a program P can reach a state a' from a state a if and only if $(a, a') \in (\!|P|\!)$. For example, reachable states during the execution of $P; P'$ are the sum of reachable states during the execution of P and reachable states during the execution of P' after P had finished. The cumulative of loop programs may not be the empty relation:

$$(\!|\texttt{while TRUE do } x := x + 1|\!) = \\ [\![x := x + 1]\!] \cup [\![x := x + 2]\!] \cup \ldots.$$

A set of initial states I is given as a subset of Σ. A set of safe states $S = \{\sigma \mid \phi(\sigma)\}$ is also a subset of Σ. We define them as binary relations:

- $i \stackrel{\text{def}}{=} \Sigma \times I$,
- $s \stackrel{\text{def}}{=} \Sigma \times S$.

The safety predicate of programs is represented by a formula of binary relations including cumulatives.

Theorem 1 $(\forall P)((\forall \sigma \in I)\Phi(P, \sigma) \Leftrightarrow (i \subseteq s \wedge i; (\!|P|\!) \subseteq s))$

Proof is easy. we omit.

4.5 Validity of Abstract Model Checking

In this section, we describe and prove a theorem of a validity of abstract model checking. The cumulative $(\!|-|\!)$, initial states i and safe states s are indexed with A for the abstract one, and indexed with C for the concrete one. For the program P, the safety property of the abstract program and the concrete program can be represented by

- AS: $i_A \subseteq s_A \wedge i_A; (\!|P|\!)_A \subseteq s_A$,
- CS: $i_C \subseteq s_C \wedge i_C; (\!|P|\!)_C \subseteq s_C$.

By using this, a theorem which a safety property of a abstract program implies a safety property of a concrete program is formulated as follows.

Theorem 2 *Suppose an interpretation $[\![-]\!]_A$ is an abstraction of an interpretation $[\![-]\!]_C$ with respect to an abstract relation α. If the condition (N) for initial states and safe states holds,*

AS implies CS.

The condition (N) is as follows.

Definition 11 *$i_C, i_A, s_C, s_A, \alpha$ satisfies the condition (N) if and only if*

- *$i_C \subseteq i_A; \alpha$ and*
- *$s_A; \alpha \subseteq s_C$.*

These formulae are same to the followings.

- *$y \in I_C \Rightarrow \exists x \in I_A, (x, y) \in \alpha$ and*
- *$x \in S_A, (x, y) \in \alpha \Rightarrow y \in S_C$.*

For designers of abstract states, the condition (N) requests that abstract initial states cover all concrete initial states and every abstract safe state has never related to any unsafe concrete state. We think this condition is natural. For example, suppose there exists a function f from concrete states to abstract states and the abstract relation is defined as $(x, y) \in \alpha$ if and only if $x = f(y)$. In this case, the abstract initial states $I_A = \{f(y) \mid y \in I_C\}$ and the safe states S_a such that $x \in S_A \Leftrightarrow x \notin \{f(y) \mid y \notin S_C\}$ are satisfies the condition (N).

We use the next lemma to prove the theorem.

Lemma 3 *Suppose an interpretation $[\![-]\!]_A$ is an abstraction of an interpretation $[\![-]\!]_C$ with respect to a abstract relation α.*

$$\begin{array}{ccc} \Sigma & \xrightarrow{(\!|P|\!)_A} & \Sigma \\ {\scriptstyle\alpha}\downarrow & \bigcup & \downarrow{\scriptstyle\alpha} \\ \Sigma & \xrightarrow{(\!|P|\!)_C} & \Sigma \end{array}$$

$$(\alpha; (\!|P|\!)_C) \subseteq ((\!|P|\!)_A; \alpha)$$

holds for every program P.

Proof is similar to the proof of extension of interpretation. Only problem is the case of the concatenation of programs. But

$$\begin{aligned}
\alpha;(\!|P;P'|\!)_C &= \alpha;((\!|P|\!)_C \cup ([\![P]\!];(\!|P'|\!)_C)) \\
&= (\alpha;(\!|P|\!)_C) \cup (\alpha;[\![P]\!]_C;(\!|P'|\!)_C) \\
&\subseteq ((\!|P|\!)_A;\alpha) \cup ([\![P]\!]_A;\alpha;(\!|P'|\!)_C) \\
&\subseteq ((\!|P|\!)_A;\alpha) \cup ([\![P]\!]_A;(\!|P'|\!)_A;\alpha) \\
&= ((\!|P|\!)_A \cup ([\![P]\!]_A;(\!|P'|\!)_A));\alpha \\
&= (\!|P;P'|\!)_C;\alpha
\end{aligned}$$

proves the lemma.

Proof of the theorem is very easy:

$$i_C \subseteq i_A;\alpha \subseteq s_A;\alpha \subseteq s_C.$$

$$\begin{aligned}
i_C;(\!|P|\!)_C &\subseteq i_A;\alpha;(\!|P|\!)_C \\
&\subseteq i_A;(\!|P|\!)_A;\alpha \\
&\subseteq s_A;\alpha \\
&\subseteq s_C.
\end{aligned}$$

4.6 Example

In this section, we show the validity of a verification of the safety property of the on-the-fly concurrent garbage collection program by using abstract model checking.

In the on-the-fly garbage collection program [25], the 'mutator' as user program and the 'collector' of garbages run concurrently. The program is written as follows.

```
program =
  while TRUE do
    (mutator | collector)
mutator =
  while TRUE do
    (Rnil + CRnil + Rnew + RR + RCR + CRR)
collector =
  while TRUE do
    (Shade; PostShade ;
     mark; PostMark;
```

```
     append; PostAppend;
     unmark; PostUnmark)
mark =
  while ExistGrayCell do
    MarkAGrayCell
append =
  while ExistWhiteCell do
    AppendAWhiteCell
unmark =
  while ExistGrayBlackCell do
    UnmarkAGrayBlackCell
```

The concurrent operator (|) is a macro which is expanded a program with nondeterministic choices and concatenations. Capitalized programs are atoms. The whole program is a term of μ-calculus.

We explain our simple model of three-colored garbage collection. There are some registers and some cells. Each register has a pointer to a cell (or nil). Each cell has a color which is either white, gray, black or free. Free cells have the color free in our model. And each cell has one pointer to a cell (or nil). The system has a variable 'collector step'. So the concrete state is described by the state of the registers, the cells and the collector step.

In order to define abstract states, we introduce abstract cells. An abstract cell has the following attributes:

- A color of the cell,
- A flag whether the cell is directly reachable from registers,
- A flag whether the cell is reachable from registers,
- A color of a cell to which the pointer point.

The abstract state is a pair of a set of abstract cells and the collector step. We can calculate a abstract state from a concrete state. We define the function f as this calculation. We define the abstract relation α such that $(x, y) \in \alpha \Leftrightarrow f(y) = x$.

The concrete transition of each atom is clear, so it defines the concrete interpretation. The abstract transition is not obvious. We have to define the abstract transition such that the abstract interpretation is an abstraction with respect to the abstract relation α. We could defined the abstract transitions which satisfies this condition in Chapter 3. Now, we get the abstract program and the concrete program in our sense.

The definition of the abstract initial states is $\{f(y) \mid y \in I_C\}$. The definition of the concrete safe state is "no free cell is reachable from registers". The definition of the abstract safe states is "there is no abstract cell such that the color is free and the reachability flag is up".

We check the condition (N). For the initial states, it is clear. For the safe states, suppose x is a safe abstract state and $(x, y) \in \alpha$. By the definition, $f(y) = x$. If y is not a safe concrete state, there exists a cell which has a color free and is reachable from registers. By considering the abstraction of this cell, x is not safe abstract state. It is a contradiction.

The safety property of the abstract program was verified by model checking. By the theorem of this chapter, the safety property of the concrete program also holds. An interesting result is the safety of concrete program is independent of the number of registers and cells.

4.7 Discussion

We formalized and proved the validity theorem of abstract model checking. The key idea is the new notion "cumulative". Although the cumulative is close to the "derivative" introduced by Hitchcock and Park[44], we could discuss safety properties of reactive programs thanks to the cumulative.

The only difference between the abstract and concrete programs is in their interpretations. The terms of sequential μ-calculus representing the two programs should be the same. By using theories of process algebra, different terms can be compared by (bi)simulation relations. To compare different terms in our calculus is a future research issue.

We considered the validity of abstract model checking for only safety properties. Model checking of safety properties is a kind of reachability analysis. We think that abstract model checking is valid for all ACTL* formulas, because model checking for ACTL* is also a kind of reachability analysis. The validity of abstract model checking for other logics should also be investigated.

Chapter 5

Abstraction of Link Structures by using Regular Expression

In Chapter 3, we abstracted the heap for concurrent GC. But the abstraction was rather ad hoc. A standard abstraction technique of link structures is required as discussed in Chapter 3. In this chapter, we introduce a general abstraction method of link structures. By this abstraction, we can remove filters introduced in Chapter 3.

A link structure consists of cells which have some pointers to cells and a character drawn from a fixed alphabet as a label. Directed graphs and trees are examples of link structures. Link structures are natural as models of memory systems or networks. The search space of model checking of such a system depends on the size of link structures, which must be limited for model checking to be possible.

We use the fact that a link structure can be considered as a nondeterministic automaton. Each cell in the link structure determines a language of the alphabet of labels. By selecting a finite set of regular expressions, we can abstract each cell of the link structure. An abstract cell has the same character as its corresponding concrete cells, and a list of boolean values, where each element of the list corresponds to a regular expression and denoting whether the language determined by the cell intersects the regular expression. The abstract link structure is defined as a set of abstract cells. The size of the abstract link structure does not depend on the size of the original link structure. We can verify the system without limitation of the size of its link structure by abstract model checking.

To do model checking of the abstract system, we must define valid transitions between abstract states. The transitions of the abstract system are valid if the unreachability of error states in the abstract system implies that of the original system. The validity of the transitions is not clear in some cases.

The main cause of this problem is that the set of abstract cells may contain unnecessary and harmful cells. We propose some techniques to remove such unnecessary abstract cells.

(The contents of this chapter are published in [92, 93].)

5.1 Related Work

Size independent properties of link structures that programs operate on have been investigated under the name of *shape-analysis* [81]. Shape-analysis reveals link structures of data such as lists, trees and circular lists. Since contents of cells are ignored, shape-analysis cannot analyze complicated programs such as garbage collection.

The queue structure is a special case of the link structure. Finite representation of the queue structure was studied by Boigelot [10, 11, 12]. Boigelot developed Queue-content Decision Diagrams (QDDs) which are finite-automaton based data structures. Unbounded FIFO queues can be represented by finite QDDs.

5.2 Abstract model checking

A *state transition system with error states* is represented as a tuple (S, T, I, E), where S is a set of states, $T \subseteq S \times S$ is a set of transitions between states, $I \subseteq S$ is a set of initial states, and $E \subseteq S$ is a set of error states. We say a system is safe if the system never reaches an error state beginning with an initial state. Model checking is a methodology for verifying the safety of a state transition system by exhaustive search.

We say another state transition system (S', T', I', E'), which is constructed from the original system (S, T, I, E) by abstraction, is *valid* with respect to (S, T, I, E), if the safety of (S', T', I', E') implies that of (S, T, I, E). To verify the safety of a system, we should only verify that of a valid abstract system. While the search space of a system is infinite or intractable, the search space of a valid abstract system may be finite or tractable. Abstract model checking uses this property. The key for abstract model checking is how to construct a valid abstract system.

5.3 Abstraction of Link Structure

In this section, we propose a method of abstracting link structures using regular expressions.

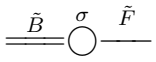
Let Σ be a finite alphabet. A *link structure* is a directed graph whose edges are called *links* and whose vertices are called *cells* and assigned a character of Σ as a label.

Let X be a link structure, and let c be a cell of X. The character of Σ assigned to c is denoted by $\sigma(c) \in \Sigma$. In the following, a cell of X is also called a concrete cell.

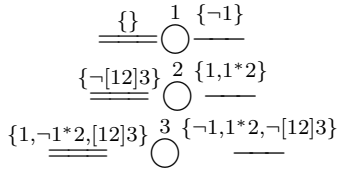
5.3.1 Abstract Cells

We abstract a link structure by a set of abstract cells. Let D be a finite set of regular expressions over *Sigma*. We introduce *negative* regular expressions by adding the prefix $\neg$ to regular expressions. We write $\neg D$ for negations of regular expressions of D, and define $\tilde{D} = D \cup \neg D$. Regular expressions without the prefix $\neg$ is called *positive.*

Definition 12 *An abstract cell (determined by D) is a triple $(\sigma, \tilde{F}, \tilde{B})$, where $\sigma \in \Sigma$, $\tilde{F} \subseteq \tilde{D}$ and $\tilde{B} \subseteq \tilde{D}$. The sets $\tilde{F}$ and $\tilde{B}$ of positive or negative regular expressions are called the forward and backward conditions, respectively. We also write $F = \tilde{F} \cap D$, $\neg F = \tilde{F} \cap \neg D$, $B = \tilde{B} \cap D$, $\neg B = \tilde{B} \cap \neg D$. In figures, abstract cells are drawn as follows.*



For example, if $\Sigma = \{1, 2, 3\}$ and $D = \{1, 1^*2, [12]3\}$, then $\neg D = \{\neg 1, \neg 1^*2, \neg [12]3\}$, where $[12]$ is an abbreviation of the regular expression $(1 + 2)$. Following are examples of abstract cells.



5.3.2 Abstraction Maps

In this section, we define the abstraction map from cells in the link structure X to abstract cells. Firstly we regard X as a nondeterministic automaton as follows:

- States of the automaton are cells in X.
- Transitions between states are links of X.

- The label of a transition is the label of the target cell of the link.
- Accepting states are all states.

A nondeterministic automaton is determined when the initial state is fixed. The language accepted by the automaton whose initial state is c is denoted by $L(X, c)$. The language accepted by the automaton whose initial state is c but whose transitions have the opposite direction is denoted by $L(X^{op}, c)$.

Definition 13 *Let D_F and D_B be subsets of D. The abstract cell $n(c) = (\sigma, \tilde{F}, \tilde{B})$ of the concrete cell c determined by D_F and D_B is defined as follows.*

- $\sigma = \sigma(c)$,
- *For each $d \in D_F$, $\tilde{F}$ has $\neg d$ when $d \cap L(X, c) = \emptyset$, and it has d, otherwise.*
- *For each $d \in D_B$, $\tilde{B}$ has $\neg d$ when $d \cap L(X^{op}, c) = \emptyset$, and it has d, otherwise.*

The abstract link structure (determined by D_F and D_B) is defined by

$$N(X) = \{n(c) \mid c \in X\}.$$

For example, let X consist of three cells c_1, c_2, c_3, and its alphabet be $\{1, 1, 2\}$. Let the links be c_1 to c_2 and c_2 to c_3. The abstract structure determined by $D_F = \{1^*2\}$ and $D_B = \{1\}$ is the set of the following abstract cells.

$$\begin{aligned} n(c_1) &= \overset{\{\neg 1\}}{=\!=\!=}\!\!\bigcirc\!\!\overset{1}{\overset{\{1^*2\}}{---}} \\ n(c_2) &= \overset{\{1\}}{=\!=\!=}\!\!\bigcirc\!\!\overset{1}{\overset{\{1^*2\}}{---}} \\ n(c_3) &= \overset{\{1\}}{=\!=\!=}\!\!\bigcirc\!\!\overset{2}{\overset{\{\neg 1^*2\}}{---}} \end{aligned}$$

5.3.3 Validity of Abstract Transitions

To construct abstract systems, we have to construct abstract transitions. In our abstraction, each abstract state is a set of abstract cells. So abstract transitions are between sets of abstract cells. We can state the validity of abstract transitions so that the abstract system becomes valid.

Definition 14 *A set of abstract transitions are valid, if for any concrete transition t from X to $t(X)$, and for any abstract state N such that $N \supseteq N(X)$, there exists an abstract transition t_A in the set such that*

$$t_A(N) \supseteq N(t(X)).$$

Lemma 4 *Let (S, T, I, E) be a state transition system whose states are link structures. Let (S', T', I', E') be an abstract system such that S' are sets of abstract cells, $I' = \{N(s) \mid s \in I\}$, and $E' = \{N | N \supseteq N(s)$ for some $s \in E\}$. If T' is valid, the abstract system is also valid.*

This lemma is easy to prove. Assume that (S, T, I, E) is not safe. Then there exists a sequence of transitions $s^0 \xrightarrow{t^1} s^1 \cdots \xrightarrow{t^n} s^n$ such that $s^0 \in I$ and $s^n \in E$. Let $s^0_A = N(s_0)$. We have $s^0_A \supseteq N(s^0)$. By the validity of T', there exists an abstract transition t^1_A such that $t^1_A(s^0_A) \supseteq N(s^1)$. Let $s^1_A = t^1_A(s^0_A)$. We can repeat this process. Consequently, there exists a sequence of abstract transitions $s^0_A \xrightarrow{t^1_A} s^1_A \cdots \xrightarrow{t^n_A} s^n_A$. We have $s^n_A \supseteq N(s_n)$. By the definition of E', $s^n_A \in E'$. So the abstract system (S', T', I', E') is not safe. Therefore, the validity of the abstract system is derived.

In general, good construction of valid transitions is not trivial. Naive construction of valid transitions tends to be too rough in the sense that the abstract system reaches error states while the concrete system is safe. In the following, we propose an approach to construct more accurate valid transitions from valid but rough transitions. One reason that transitions become too rough is because the target of an abstract transition may contain abstract cells that are inconsistent with other cells. Our approach is to calculate inconsistent abstract cells in the target and remove them.

5.3.4 Inconsistent Abstract Cells

In order to calculate inconsistent abstract cells, we first make a directed graph from a set of abstract cells. We put an edge between two abstract cells, taking negative conditions of abstract cells into account. From negative conditions, we can detect non-existence of a link. We put edges unless we detect non-existence.

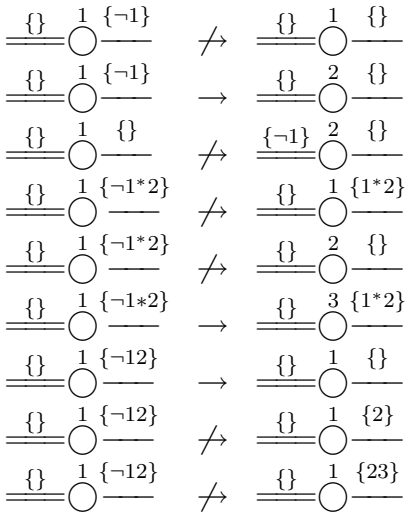
Definition 15 *Let N be a set of abstract cells and i be a natural number. A set of edges $E^i_N \subseteq N \times N$ is defined as follows. We define $(n_1, n_2) \notin E^i_N$ if and only if*

- $\exists \neg f_1 \in \neg F_1 \ . \ (\sigma_2 \in f_1 \vee \ \exists f_2 \in F_2 \ . \ (\sigma_2 f_2 \subseteq f_1 \vee 1 \leq \exists k \leq i \ . \ pre_k(\sigma_2 f_2) \subseteq f_1))$, *or*
- $\exists \neg b_2 \in \neg B_2 \ . \ (\sigma_1 \in b_2 \vee \ \exists b_1 \in B_1 \ . \ (\sigma_1 b_1 \subseteq b_2 \vee 1 \leq \exists k \leq i \ . \ pre_k(\sigma_1 b_1) \subseteq b_2))$,

where $n_1 = (\sigma_1, \tilde{F}_1, \tilde{B}_1), n_2 = (\sigma_2, \tilde{F}_2, \tilde{B}_2) \in N$. We have used the following notations: $\sigma_2 f_2$ and $\sigma_1 b_1$ denotes concatenations of regular expressions.

$pre_k(\sigma_2 f_2) = \{\ s|_k \mid s \in \sigma_2 f_2\}$, where $s|_k$ is a prefix of s with length k if the length of s is greater than k, and $s|_k$ is s, otherwise.

For example, E_N^i is as follows ($\Sigma = \{1, 2, 3\}, D = \{1, 1^*2, 12, 2, 23\}$).



Notice that the final example is for a case $i \geq 2$. There exists an edge when $i = 1$. Others are independent of i.

The value of i in E_N^i has no effect in the following discussions. So we assume an appropriate value of i has been selected, and omit i in the following. We simply write E_N for E_N^i.

Notice that the existence of an edge between two cells does not depend on other cells than them. The next lemma is an easy consequence of this fact.

Lemma 5 *Let N and N' be sets of abstract cells. If n_1 and n_2 belong to both of them, then $(n_1, n_2) \in E_N$ is equivalent to $(n_1, n_2) \in E_{N'}$. Especially, when N is a subset of N', (N, E_N) is a subgraph of $(N', E_{N'})$.*

Now we use positive conditions to calculate inconsistent abstract cells. We first regard the graph of abstract cells defined so far as an automaton.

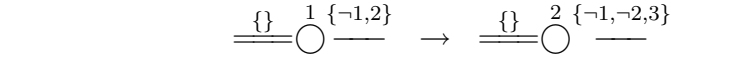
When a directed graph (N, E) of abstract cells and an abstract cell $n \in N$ are given, we define a *nondeterministic finite automaton* (N, E, n) as follows. The states are abstract cells in N and the initial state is n. A transition from n_1 to n_2 exists only when $(n_1, n_2) \in E$, and its label is the character assigned to n_2. The accepting states are all the abstract cells. The language accepted by (N, E, n) is written as $L(N, E, n)$. We write E^{op} for the edges whose directions are opposite to those of E.

Definition 16 *Let N be a set of abstract cells. An abstract cell $n = (\sigma, \tilde{F}, \tilde{B}) \in N$ is consistent in N if and only if*

- $\forall f \in F \ . \ f \cap L(N, E_N, n) \neq \emptyset$, *and*
- $\forall b \in B \ . \ b \cap L(N, E_N^{op}, n) \neq \emptyset$.

If n is not consistent in N, we say n is inconsistent in N.

For example, let N have just two abstract cells, $n_1 = (1, \{\neg 1, 2\}, \{\})$ and $n_2 = (2, \{\neg 1, \neg 2, 3\}, \{\})$. The directed graph (N, E_N) is as follows.



The languages accepted by them are $L(N, E_N, n_1) = \{2\}$ and $L(N, E_N, n_2) = \emptyset$. In this case, n_1 is consistent in N, but n_2 is inconsistent in N.

Definition 17 *Let $I(N) = \{n \in N \mid n \text{is inconsistent in } N\}$. We repeat $N := N - I(N)$ until $I(N) = \emptyset$, and define $R(N)$ as the resulting set N.*

The set $R(N)$ is uniquely determined from N, and every element of $R(N)$ is consistent in $R(N)$.

In the previous example, n_1 is consistent in N and n_2 is inconsistent. But, n_1 is inconsistent in $N - I(N) = \{n_1\}$. Therefore, $R(N) = \emptyset$.

Lemma 6 *If there exists a link from c_1 to c_2 in X, $(n(c_1), n(c_2)) \in E_{N(X)}$.*

Proof. Let $n(c_1) = (\sigma_1, \tilde{F_1}, \tilde{B_1})$ and $n(c_2) = (\sigma_2, \tilde{F_2}, \tilde{B_2})$. If $(n(c_1), n(c_2)) \notin E_{N(X)}$, we can assume $\exists \neg f_1 \in \neg F_1 \ . \ (\sigma_2 \in f_1 \vee \ \exists f_2 \in F_2 \ . \ (\sigma_2 f_2 \subseteq f_1 \vee 1 \leq \exists k \leq i \ . \ \mathrm{pre}_k(\sigma_2 f_2) \subseteq f_1))$ because of symmetry. If $\sigma_2 \in f_1$, $\sigma_2 \in L(X, c_1)$. This contradicts to $f_1 \cap L(X, c_1) = \emptyset$. If $\sigma_2 f_2 \subseteq f_1$, $f_2 \cap L(X, c_2) \neq \emptyset$. $f_1 \cap L(X, c_1)$ is not empty, because $\sigma_2(f_2 \cap L(X, c_2)) = \sigma_2 f_2 \cap \sigma_2 L(X, c_2) \subseteq f_1 \cap L(X, c_1)$. This contradicts to the construction of $n(c_1)$. If $\exists k \geq 1 \ . \ \mathrm{pre}_k(\sigma_2 f_2) \subseteq f_1$, there exists a sequence $s_2 \in L(X, c_2) \cap f_2$. Therefore $\sigma_2 s_2|_k \in f_1$ holds. But $\sigma_2 s_2|_k \in L(X, c_1)$ also holds. This contradicts to $f_1 \cap L(X, c_1) = \emptyset$.

Lemma 7 *All abstract cells in $N(X)$ are consistent in $N(X)$.*

Proof. We assume $n(c)$ is inconsistent in $N(X)$. We can then assume that there exists $f \in F$ such that $f \cap L(N(X), E_{N(X)}, n(c)) = \emptyset$ by symmetry. By the construction of $n(c)$, $f \cap L(X, c) \neq \emptyset$ holds. We select a sequence $\sigma_1 \ldots \sigma_k \in f \cap L(X, c)$. By the definition of $L(X, c)$, there exists a sequence of cells $cc_1 \ldots c_k$ in X such that $\sigma_i = \sigma(c_i)$. By the previous lemma, $n(c), n(c_1), \ldots, n(c_k)$ is a transition sequence in $(N(X), E_{N(X)}, n(c))$. Therefore, $\sigma_1 \ldots \sigma_k$ is acceptable. This contradicts to $f \cap L(N(X), E_{N(X)}, n(c)) = \emptyset$.

Now we can conclude the next theorem.

Theorem 3 *If a link structure X and a set of abstract cells N satisfy $N \supseteq N(X)$, then $N \supseteq R(N) \supseteq N(X)$ holds.*

We define a transition $t_A^R(N) = R(t_A(N))$ from a transition t_A. As a corollary, this new transition t_A^R is valid if t_A is valid, and t_A^R is more accurate than t_A.

We show an example of t_A^R. Let $\Sigma = \{1, 2\}$, $D_B = \emptyset$, and $D_F = \{2\}$. We consider a concrete transition t that deletes $\{c \in X \mid \sigma(c) = 2\}$ from X. The abstract transition t_A that naturally corresponds to t deletes $\{(\sigma, \tilde{F}, \tilde{B}) \mid \sigma = 2\}$ from a set of abstract cells. Now assume

$$N = \{\overset{\{\}}{=\!=}\!\overset{1}{\bigcirc}\!\overset{\{\neg 2\}}{-\!-}, \overset{\{\}}{=\!=}\!\overset{1}{\bigcirc}\!\overset{\{2\}}{-\!-}, \overset{\{\}}{=\!=}\!\overset{2}{\bigcirc}\!\overset{\{2\}}{-\!-}\}.$$

Then

$$t_A(N) = \{\overset{\{\}}{=\!=}\!\overset{1}{\bigcirc}\!\overset{\{\neg 2\}}{-\!-}, \overset{\{\}}{=\!=}\!\overset{1}{\bigcirc}\!\overset{\{2\}}{-\!-}\}.$$

The second abstract cell is inconsistent, so

$$t_A^R(N) = \{\overset{\{\}}{=\!=}\!\overset{1}{\bigcirc}\!\overset{\{\neg 2\}}{-\!-}\}.$$

Defining this abstract transition directly from t is troublesome.

We will see another example in which t_A is not safe but t_A^R is safe in Section 5.4.

5.3.5 Conservative Abstract Transitions

As elementary transitions which update a link structure, we consider the following kinds of transitions.

- change the character of a cell.
- delete a link between cells.
- insert a link between cells.

For these three kinds of transitions, we define the following conservative abstract transitions. Given a set N of abstract cells, each transition adds new abstract cells to N. The resulting set of abstract cells becomes the target of the transition.

The case a transition changes the character $\sigma(c)$ of the cell c to σ'. The corresponding conservative abstract transition adds the following abstract cells to N.

- $(\sigma_1, \tilde{F}_1 - \{f\} \cup \{\neg f\}, \tilde{B}_1)$, for each ancestor $(\sigma_1, \tilde{F}_1, \tilde{B}_1)$ of $n(c)$ in the graph (N, E_N), and each $f \in F_1$ which contains $\sigma(c)$.
- $(\sigma_1, \tilde{F}_1 - \{\neg f\} \cup \{f\}, \tilde{B}_1)$, for each ancestor $(\sigma_1, \tilde{F}_1, \tilde{B}_1)$ of $n(c)$ in the graph (N, E_N), and each $\neg f \in \neg F_1$ which contains σ'.
- $(\sigma_2, \tilde{F}_2, \tilde{B}_2 - \{b\} \cup \{\neg b\})$, for each descendant $(\sigma_2, \tilde{F}_2, \tilde{B}_2)$ of $n(c)$ in the graph (N, E_N), and each $b \in B_2$ which contains $\sigma(c)$.
- $(\sigma_2, \tilde{F}_2, \tilde{B}_2 - \{\neg b\} \cup \{b\})$, for each descendant $(\sigma_2, \tilde{F}_2, \tilde{B}_2)$ of $n(c)$ in the graph (N, E_N), and each $\neg b \in \neg B_2$ which contains σ'.

The case a transition deletes a link c to c'. The corresponding conservative abstract transition adds the following abstract cells to N.

- $(\sigma_1, \tilde{F}_1 - \{f\} \cup \{\neg f\}, \tilde{B}_1)$, for each ancestor $(\sigma_1, \tilde{F}_1, \tilde{B}_1)$ of $n(c)$ in the graph (N, E_N), and each $f \in F_1$ which contains $\sigma(c')$.
- $(\sigma_2, \tilde{F}_2, \tilde{B}_2 - \{b\} \cup \{\neg b\})$, for each descendant $(\sigma_2, \tilde{F}_2, \tilde{B}_2)$ of $n(c')$ in the graph (N, E_N), and each $b \in B_2$ which contains $\sigma(c)$.

The case a transition inserts a link c to c'. The corresponding conservative abstract transition adds the following abstract cells to N.

- $(\sigma_1, \tilde{F}_1 - \{\neg f\} \cup \{f\}, \tilde{B}_1)$, for each ancestor $(\sigma_1, \tilde{F}_1, \tilde{B}_1)$ of $n(c)$ in the graph (N, E_N), and each $\neg f \in \neg F_1$ which contains $\sigma(c')$.
- $(\sigma_2, \tilde{F}_2, \tilde{B}_2 - \{negb\} \cup \{b\})$, for each descendant $(\sigma_2, \tilde{F}_2, \tilde{B}_2)$ of $n(c')$ in the graph (N, E_N), and each $\neg b \in \neg B_2$ which contains $\sigma(c)$.

The validity of these abstract transitions are straightforward.

Actual transitions can be combinations of these elementary transitions. In case combinations are indivisible, we would have to make other abstract transitions.

These conservative abstract transitions never delete abstract cells. In general, abstract transitions which delete abstract cells are also needed for abstract model checking. In some cases, the validity of such transitions are guaranteed naturally. We call them natural transitions. From a natural transition t, we can make a more accurate abstract transition t_A^R. In the next section, we will see an example in which t_A^R is actually needed.

5.4 Example: Concurrent Garbage Collection

We verify the safety property of a concurrent garbage collection algorithm using this abstraction.

5.4.1 Abstract Heap

Now, we consider the on-the-fly garbage collection algorithm. We postpone the treatment of the snapshot algorithm until Section 5.4.3. In our model of garbage collection, the registers and heap form a link structure. We abstract this link structure using our method. We call a set of abstract cells an *abstract heap.*

To define an abstract heap, we need some preparations. Firstly, the alphabet consists of the following characters:

- r which means the cell is in register,
- w which means the color of cell in heap is white,
- g which means the color of cell in heap is gray,
- b which means the color of cell in heap is black, and
- f which means the color of cell in heap is free.

That is, we use $\Sigma = \{r, f, b, g, w\}$.

The most important decision in our abstraction method is the selection of regular expressions for abstraction. We use the color of the pointed cell for forward conditions: $D_F = \{f, b, g, w, r\}$. For backward conditions, we should have the property "the cell is reachable from registers", to express the safety. We should also have the property "the cell is directly reachable from registers" because the mutator manipulates directly reachable cells. These properties can be described as regular expressions: $D_B = \{r, [bgw]^*r\}$.

In our model, each cell has just one link. So an abstract cell can have at most one positive forward condition. The backward conditions of an abstract cell may not contain both r and $\neg[bgw]^*r$. In general, for two regular expressions f_1 and f_2, if $f_1 \subseteq f_2$, then abstract cells which contain f_1 and $\neg f_2$ in its forward (backward) conditions are inconsistent.

The abstract heap of figure 3.1 have the following abstract cells.

$$\{\neg r,\underline{\neg[bgw]^*r}\}\overset{r}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,w,\neg r\}$$
$$\{\neg r,\underline{\neg[bgw]^*r}\}\overset{r}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,\neg w,\neg r\}$$
$$\{r,\underline{[bgw]^*r}\}\overset{w}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,w,\neg r\}$$
$$\{\neg r,\underline{\neg[bgw]^*r}\}\overset{b}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,\neg w,\neg r\}$$
$$\{\neg r,\underline{[bgw]^*r}\}\overset{w}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,\neg w,\neg r\}$$
$$\{\neg r,\underline{\neg[bgw]^*r}\}\overset{w}{\bigcirc}\{\underline{\neg f},\neg b,\neg g,\neg w,\neg r\}$$

An entire abstract state is a pair of a collector step and an abstract heap. But, in our model checker, abstract heaps which have the same collector step are merged (see Figure 3.2).

5.4.2 Abstract Transitions

To do abstract model checking, we also have to define valid abstract transitions.

The abstract transitions of mutator transitions and collector transitions which do not change the collector step are defined as conservative transitions according to Section 5.3.5. For each transition, the abstract transition adds some abstract cells to the abstract heap.

Abstraction of the transitions which change the collector step is not trivial. We define filters for them. The filters delete abstract cells according to the condition with which the collector is allowed to change its step. They are listed in Table 3.3.

We then define a natural abstract transition corresponding to each change of the collector step. For example, the collector step changes `append` from `mark`, a natural transition adds to the abstract heap of `append` those abstract cells that are filtered from the abstract heap of `mark`. The validity of these abstract transitions is easy to verify. But the resulting abstract system reaches error states.

We use more accurate abstract transitions instead of the natural transitions defined above by removing inconsistent abstract cells as described in Section 5.3.4. The unreachability of error states of this abstract system is verified by model checking. We use 1 as the number i for the construction of E^i_N.

5.4.3 Snapshot Algorithm

In the case of the snapshot algorithm, we need another property: "the cell is reachable from a gray cell through white cells". This property can be expressed by the regular expression w^*g. By adding it to D_B, abstract model checking of the snapshot algorithm succeed.

As is shown in this example, "how to abstract" amounts to "what regular expressions to use". This greatly eases the design of abstraction.

5.5 Discussion

In this chapter, we proposed a new abstraction method for link structures using regular expressions. The size of abstract states is independent of the size of a link structure. We then discussed abstract model checking using this abstraction. As an example, the safety property of concurrent garbage collection algorithms was verified.

Although we restricted each cell to have at most one pointer for making the model of garbage collection simple, we can easily extend the model to allow multiple pointers.

One advantage of our abstraction is that selection of regular expressions determines abstraction. We can easily try various instances of abstraction by simply changing the set of regular expressions.

The abstract transitions were not defined systematically. We gave an informal construction principle of abstract transitions. The paper [17] gave a construction of abstract transitions under the condition that transitions of concrete data are represented by first order formulas. A construction of *abstract BDDs* is developed in the paper [20], where concrete transitions and abstract relations are both represented by BDDs. However, these constructions do not work in our case. Therefore, abstract transitions should be defined by the user, and the user has to prove that the abstract transitions satisfy the conditions given in Chapter 4 that guarantee the safety. In order to give a mechanical construction of abstract transitions, further research is needed.

Chapter 6

Discovery by Model Checking

In this chapter, we describe an approach to synthesizing programs or algorithms by model checking. Automatic verifiers employed in model checking are called model checkers. They explore the state space constructed from a given program and its specification expressed using temporal logic.

In order to synthesize a program using a model checker, a space of programs must first be defined to which the target program is expected to belong. There are many ways to define such a space. It can be defined by a set of parameters. A program belonging to the space is then defined as a tuple of the values of the parameters. Programs can also be written in a programming language. In this case, the space consists of programs expressed in the language. Since such a space is infinite in general, it is usually necessary to impose restrictions on the size of programs.

An automatic verifier, i.e., model checker, is then invoked on each program in the program space to try to verify the specification that the program should satisfy. The verifier determines, within a finite time, whether the specification is satisfied or not. Therefore, it is possible to search for a program that satisfies the specification by simply searching through the program space.

In this chapter, we take the approach to find algorithms for concurrent garbage collection (Section 6.2) and for mutual exclusion without semaphores (Section 6.3).

(The contents of this chapter are published in [40].)

6.1 Related work

There are successful works in which new algorithms are discovered using similar approach to ours. One is superoptimization which is automatic code generation technique. Another is Perrig and Song's recent work on finding

```
(x in d0)
add.l d0,d0
subx.l d1,d1
negx.l d0
addx.l d1,d1
(signum(x) in d1)
```

(a) Motorola 68000

```
(x in %o0)
addcc  %o0, %o0, %l0
subxcc %o0, %l0, %l0
addxcc %o0, %l0, %o0
(signum(x) in %o0)
```

(b) Sparc

Figure 6.1: Code generated by superoptimizers.

authentication protocols using a protocol verifier of authentication protocols. We briefly survey them in the following subsections.

6.1.1 Superoptimization

Superoptimization [64, 36, 30] is the most practical and successful work in this field. It is a technique used to synthesize the code generation table of a compiler. Given the specification of a code sequence, corresponding to some operation in an intermediate language, it searches for sequences of machine code that satisfy that specification.

Massalin initiated this technique, later made popular by Granlund and Kenner, who used it to synthesize a code generation table for the GCC compiler. The code in Figure 6.1(a) was found by Massalin's superoptimizer for Motorola's 68020. This is the code for the following function (written in C):

```
int signum(int x) {
    if (x>0) return 1;
    else if (x<0) return -1;
    else return 0;
}
```

Recently, Mizukami repeated Massalin's work for Sparc, and obtained the code in Figure 6.1(b) [67].

Superoptimization does not employ a real verifier; it only checks the correctness of the discovered code using some random numbers. This is sufficient in practice, since a human has the final decision as to whether to incorporate the code sequences obtained into the code generation table or not. In this final step, the correctness of the generated code can be manually checked.

$$
\begin{array}{lcl}
A \to B & : & \{N_A, A\}_{K_{AB}} \\
B \to A & : & \{N_A, N_B\}_{K_{AB}} \\
A \to B & : & N_B
\end{array}
\qquad
\begin{array}{lcl}
A \to B & : & N_A, A \\
B \to A & : & \{N_A, N_B, A\}_{K_{AB}} \\
A \to B & : & N_B
\end{array}
\qquad
\begin{array}{lcl}
A \to B & : & N_A, A \\
B \to A & : & \{N_A, N_B, B\}_{K_{AB}} \\
A \to B & : & N_B
\end{array}
$$

Figure 6.2: Protocols generated by the automatic protocol generator.

6.1.2 Authentication Protocols

Finding algorithms using verifiers is effective in a field where the algorithms are very short, but still require substantial effort to verify their correctness. Security protocols, such as those used for authentication, are good examples of such algorithms.

Perrig and Song [73] recently used Song's protocol verifier, Athena [87], to try to discover symmetric-key and asymmetric-key mutual authentication protocols that satisfy the agreement property. They also defined a metric function that measures the cost or overhead of protocols. This function was intended to be applied to correct protocols to select the most efficient. The function was also used to restrict the set of generated protocols.

After setting `UNIT_ELEMENT_COST=1` (cost to send a nonce or a principal name), `NEW_NONCE_COST=1` (cost to generate a new nonce), and `ASYM_ENCRYPTION_COST=3` (cost to encrypt a message with an asymmetric key), they succeeded in discovering the following protocol among those whose cost is less than or equal to 14:

$$
\begin{array}{lcl}
\texttt{Protocol}: \quad A \to B & : & \{N_A, A\}_{K_B} \\
B \to A & : & \{N_A, N_B, B\}_{K_A} \\
A \to B & : & N_B
\end{array}
$$

The protocol coincides with the Needham-Schroeder protocol, except for the last message. In the Needham-Schroeder protocol, the last message is $\{N_B\}_{K_B}$. This difference occurred because they did not include secrecy in the specification.

The symmetric mutual authentication protocols they found, shown in Figure 6.2, are more interesting. All these protocols are simpler than any known previously. The first protocol was found using a metric function whose `UNIT_ELEMENT_COST` is high, while the last two were found using a metric function whose `SYM_ENCRYPTION_COST` is high.

Perrig and Song demonstrate the applicability of their method based on these results. Their success is due mainly to their efforts to reduce the number of protocols by imposing reasonable restrictions, and to prune protocols by *cheap* tests before applying the protocol verifier.

6.2 Concurrent Garbage Collection

The first case study using model checking is finding algorithms for concurrent garbage collection. Such an algorithm is comprised of two processes: a collector that collects garbage cells, and a mutator that manipulates cell pointers and does some computations. Algorithms such as on-the-fly [25] and snapshot [101, 102] are well known for concurrent garbage collection. Although these two algorithms are based on completely different ideas, they can be modeled in a uniform framework. For example, both employ four cell colors: white, black, gray, and free. The color "free" means that a cell is free, i.e., allocatable.

In our framework, the mutator refers to some registers, each holding a pointer to a cell. The collector also refers to the registers when it marks the cells in use. Since the collector begins marking with the pointers held in the registers, a register is also called a *root.*

The mutators used in the on-the-fly and snapshot algorithms have different operations. In this case study, thirteen operations are defined that cover both algorithms. In the following description, `R[i]` denotes the contents of the `i`-th register, and `F[i]` denotes the contents of the field of the `i`-th cell. Indices of cells begin with `1`, and the index `0` denotes nil (the null pointer). The procedure, `white_to_gray(i)`, makes the white `i`-th cell gray, and `i` is not `0`.

(0) Allocate a new cell in the shade step.

(1) Allocate a new cell in the mark step.

(2) Allocate a new cell in the append step.

(3) Allocate a new cell in the unmark step.

(4) Make a newly allocated cell gray.

(5) Make a newly allocated cell black.

(6) `white_to_gray(F[R[i]]); F[R[i]] := 0;`

(7) `F[R[i]] := 0;`

(8) `R[i] := F[R[j]]; white_to_gray(R[i]);`

(9) `R[i] := F[R[j]];`

(10) `F[R[i]] := R[j]; white_to_gray(F[R[i]]);`

(11) `F[R[i]] := R[j];`

(12) `white_to_gray(F[R[i]]); F[R[i]] := R[j];`

Allocation of a cell is accomplished by a combination of one of (0), (1), (2) or (3), and one of (4) or (5). Each of (0), (1), (2) and (3) corresponds to a collector step.

An error state occurs when a cell that is reachable from a register (root) becomes free. If error states are not reachable from an initial state, an algorithm for concurrent garbage collection is said to be safe. Although liveness is also required for concurrent garbage collection, we search for algorithms only according to the safety property.

In this case study, we searched for algorithms with respect to a finite model. The model consists of only three cells and three registers. A state of the model can therefore be represented by 20 bits. The only initial state is that in which all the registers hold the `0` value, all the cells are free, and the collector is in the shade step.

We defined the algorithm space according to whether or not each of thirteen operations is allowed. The number of algorithms in the space is therefore 2^{13}. We applied finite model checking to each algorithm and computed the maximal algorithms that satisfy the safety property, i.e., do not reach an error state. Algorithms are maximal if they allow as many operations as possible. The verifier, i.e., model checker, was implemented in C.

The following maximal algorithms were identified:

- 0000111111111 no allocation
- 1111001111111 no allocation
- 0111111110111 a variant of on-the-fly
- 1111101110111 on-the-fly
- 0111111011001 snapshot
- 1111101011001 a variant of snapshot
- 1111111110100 a variant of on-the-fly
- 1111111111000 no field update

Each algorithm is represented by thirteen bits corresponding to the thirteen operations, where 0 indicates that the corresponding operation is inhibited, and 1 indicates that it is allowed. Among these algorithms, the first two do

not permit allocation of new cells. The last one does not permit the fields to be updated. Therefore, only the following five algorithms are meaningful for concurrent garbage collection:

- on-the-fly and its two variants,
- snapshot and its one variant.

There are no other (correct) algorithms that consist of the above operations. Before the experiment, it was expected that there might be algorithms that were different from on-the-fly and snapshot, but the discovered algorithms can all be classified into one of these two kinds. Liveness of concurrent garbage collection means that garbage cells (those cells that are not reachable from a register) are eventually collected (and made free). The algorithms identified above all satisfy the liveness property. Therefore, in this case study, the safety property was sufficient to permit discovery of the appropriate algorithms. Furthermore, the correctness of the discovered algorithms is verified independent of the number of cells or registers, using the technique of abstraction in Chapter 3.

6.3 Mutual Exclusion without Semaphores

This case study looks for variants of Dekker's algorithm for mutual exclusion among processes. In the previous study, the algorithm space was defined by a fixed set of operations. In other words, the space was comprised of tuples of thirteen boolean values. In this case study, we define the space of programs consisting of pseudo-instructions.

Dekker's algorithm realizes mutual execution among processes without semaphores. Figure 6.3 shows an instance of the algorithm for two processes. It can be generalized to an arbitrary number of processes, but we consider only two processes in this chapter. In the figure, `me` denotes the number of the process that is executing the code (`1` or `2`), and `you` denotes the number of the other process (`2` or `1`). The entry part realizes mutual execution before entering the critical section, and the finishing part is executed after the critical section. The idle part represents a process-dependent task.

The safety property of Dekker's algorithm is:

> Two processes do not simultaneously enter the critical section.

Liveness is:

> There does not exist an execution path (loop) that begins and ends with the same state, at which one process is in its entry part, and satisfies the following conditions.

```
for (;;) {
    // beginning of the entry part
    flags[me] = true;
    while (flags[you] == true) {
       if (turn != me) {
           flags[me] = false;
           while (turn != me) {}
           flags[me] = true;
       }
    } // end of the entry part

    // the critical section

    // beginning of the finishing part
    turn = you;
    flags[me] = false;
    // end of the finishing part

    // the idle part
}
```

Figure 6.3: Dekker's algorithm.

- The process stays in the entry part on the execution path, i.e., it does not enter the critical section.
- Both processes execute at least one instruction on the execution path.

In this case study, liveness was also checked during the search, using the *nested depth-first search.* This is a technique developed for SPIN, one of the more popular model checkers [46]. The verifier, as well as the search problem discussed later, was implemented in Java.

We represent Dekker's algorithm using pseudo-code consisting of pseudo-instructions. The pseudo-code may refer to three variables, each of which holds a boolean value.

- `FLAG1`: This variable corresponds to `flags[1]` in Figure 6.3.
- `FLAG2`: This variable corresponds to `flags[2]` in Figure 6.3.
- `TURN`: This variable corresponds to `turn` in Figure 6.3. `TURN=true` means `turn=1`, and `TURN=false` means `turn=2`,

The set of pseudo-instructions are:

- `SET` *variable*

- `CLEAR` *variable*
- `IF` *variable* {*instructions*}
- `IF_NOT` *variable* {*instructions*}
- `WHILE` *variable* {*instructions*}
- `WHILE_NOT` *variable* {*instructions*}

The entry part of process 1 is represented by the following pseudo-code.

```
SET FLAG1
WHILE FLAG2 {
    IF_NOT TURN {
        CLEAR FLAG1
        WHILE_NOT TURN {}
        SET FLAG1
    }
}
```

In this case study, we searched for variants of the entry part of Dekker's algorithm that satisfy both safety and liveness. The entry parts of the two processes were assumed to be symmetric in the following sense. If one refers to `FLAG1` (or `FLAG2`), then the other refers to `FLAG2` (or `FLAG1`). If one contains an instruction that refers to `TURN`, then the other contains the corresponding symmetric instruction, where `IF` and `IF_NOT`, `WHILE` and `WHILE_NOT`, and `SET` and `CLEAR` are symmetric to each other.

In the first experiment, we searched for a 5-instruction pseudo-code consisting of the following instructions. Note that the original entry part consists of 6 instructions.

- `WHILE FLAG2 ...`
- `IF FLAG2 ...`
- `WHILE_NOT TURN ...`
- `IF_NOT TURN ...`
- `SET FLAG1`
- `CLEAR FLAG1`

```
SET FLAG1
WHILE FLAG2 {
    WHILE_NOT TURN {
        CLEAR FLAG1
    }
    SET FLAG1
}
```

(a)

```
WHILE FLAG2 {}
SET FLAG1
IF_NOT TURN {
    WHILE FLAG2 {}
}
```

(b)

Figure 6.4: Generated pseudo-code.

We discovered the code shown in Figure 6.4(a). This code, however, is equivalent to the original code of Dekker's algorithm, in the sense that they have the same meaning as sequential programs. We also found more than ten variants similar to the above, all equivalent to the original code as sequential programs.

In the next experiment, we imposed the following restriction.

> If both processes are in their entry or finishing parts, they run with the same speed, i.e., they both execute one instruction in a single state transition of the entire system. If they read or write to the same variable simultaneously, it is nondeterministic which process wins (runs first). If both processes are not in their entry or finishing parts, one process is chosen nondeterministically and allowed to execute one instruction.

Under this restriction, we discovered the code in Figure 6.4(b), which consists of 4 instructions. This is the only 4-instruction code that we discovered. We also searched for a 3-instruction code, but failed.

The correctness of the discovered code is not obvious. We believe that finding algorithms by verifiers is effective in situations where complex conditions are imposed that are difficult for humans to grasp.

6.4 Discussion

We found out new algorithms by using model checking. The issue of how to decrease the size of the search space used to find programs is the most crucial issue. For example, Perrig and Song pruned protocols before applying the protocol verifier. More research should be performed to obtain general principles for efficient search for programs.

Chapter 7

Searching for Synchronization Algorithms using BDDs

In Chapter 6, we tried to discover new algorithms that satisfy a given specification, by first defining a space of algorithms, and then checking each algorithm in the space against the specification, using a verifier, i.e., model checker. By this approach, we discovered new variants of the existing algorithms for concurrent garbage collection, and a new algorithm for mutual exclusion under some restrictions on parallel execution.

Needless to say, the most serious problem of this approach is search space explosion. The space of algorithms explodes if the size of enumerated algorithms is not carefully bounded. It is therefore vital for the approach to efficiently traverse the space of algorithms, employing various kinds of search heuristics.

In this chapter, we investigate the possibility of simultaneously checking all algorithms in the algorithm space by a single execution of the verifier. Algorithms in the space are symbolically represented by a template containing parameters. Each instantiation of the parameters in the template corresponds to a specific algorithm. By symbolically verifying the template, we obtain constraints on the parameters for the template to be correct.

Symbolic verification make it possible to share computations for verification among different algorithms, because computations with some parameters uninstantiated are shared by algorithms that correspond to instantiations of those parameters.

By symbolically representing a template of algorithms, it is also possible to apply approximation or abstraction [22] on the template. Before or during verification, algorithms can be approximated by another having similar but smaller representation. If abstraction could be applied on symbolic representation, it would greatly reduce the search space.

In this chapter, we employ BDDs (binary decision diagrams) for symbolic representation and verification [14, 18]. As a case study, we take the problem of searching for synchronization algorithms for mutual exclusion without using semaphores, which was also taken in Chapter 6.

The template of algorithms is represented by a sequence of pseudo-instructions including boolean parameters. The template has the original version of Peterson's or Dekker's algorithm as an instance. The predicate defining the initial state, that of error states, and the state transition relation are all represented by BDDs.

Finally, we make an experiment in which we collapsed BDDs with a small Hamming distance. This is a first step towards using approximation and abstraction in the approach.

(The contents of this chapter are published in [90])

7.1 Mutual Exclusion

The target of these case studies is to synthesize synchronization algorithms for mutual exclusion without using semaphores.

Peterson and Dekker's algorithms realize mutual exclusion among processes without semaphores. Figure 7.1 shows an instance of Peterson's algorithm for two processes. An instance of Dekker's algorithm is shown in Figure 6.3.

In the figures, `me` denotes the number of the process that is executing the code (`1` or `2`), and `you` denotes the number of the other process (`2` or `1`). The entry part realizes mutual execution before entering the critical section, and the finishing part is executed after the critical section. The idle part represents a process-dependent task.

The safety property of mutual exclusion algorithm is:

> Two processes do not simultaneously enter the critical section.

Liveness is:

> There does not exist an execution path (loop) that begins and ends with the same state, at which one process is in its entry part, and satisfies the following conditions.
>
> - The process stays in the entry part on the execution path, i.e., it does not enter the critical section.
> - Both processes execute at least one instruction on the execution path.

```
for (;;) {
    // beginning of the entry part
    flags[me] = true;
    turn = you;
    while (flags[you] == true) {
        if (turn != you) break;
    } // end of the entry part

    // the critical section

    // beginning of the finishing part
    flags[me] = false;
    // end of the finishing part

    // the idle part
}
```

Figure 7.1: Peterson's algorithm.

The results of searching for variants of these algorithms are described in the next two sections.

7.2 Search for Variants of Peterson's Algorithm

In this section, we describe the first case study. We make a template which has Peterson's algorithm as an instance, and check the safety and liveness of the template using BDDs.

7.2.1 Pseudo-code

We represent Peterson's algorithm using pseudo-code consisting of pseudo-instructions. The pseudo-code may refer to three variables, each of which holds a boolean value.

- `FLAG1`: This variable corresponds to `flags[1]` in Figure 7.1.
- `FLAG2`: This variable corresponds to `flags[2]` in Figure 7.1.
- `FLAG0`: This variable corresponds to `turn` in Figure 7.1. `FLAG0=true` means `turn=2`, and `FLAG0=false` means `turn=1`,

	Process 1	Process 2
SET_CLEAR(b00),L	goto L	goto L
SET_CLEAR(b01),L	FLAG0 := b; goto L	FLAG0 := not(b); goto L
SET_CLEAR(b10),L	FLAG1 := b; goto L	FLAG2 := b; goto L
SET_CLEAR(b11),L	FLAG2 := b; goto L	FLAG1 := b; goto L

Figure 7.2: SET_CLEAR

Each instruction in the pseudo-code is in one of the following three forms.

SET_CLEAR(p),L
IF_WHILE(p),L_1,L_2
NOP,L_1,L_2

The operands L, L_1 and L_2 in the instructions denote addresses in the pseudo-code. In SET_CLEAR(p),L, the operand L should point to the next address in the pseudo-code.

The operators SET_CLEAR and IF_WHILE have a three-bit parameter, denoted by p. Each value of the parameter results in a pseudo-instruction as defined in Figures 7.2 and 7.3 for each process. In the figures, b denotes 0 or 1.

The instruction

NOP,L_1,L_2

jumps to either L_1 or L_2 nondeterministically.

The original version of Peterson's algorithm for mutual exclusion can be represented by the following pseudo-code.

```
0:  SET_CLEAR(110),1
1:  SET_CLEAR(101),2
2:  IF_WHILE(111),3,4
3:  IF_WHILE(001),4,2
4:  NOP,5,5
5:  SET_CLEAR(010),6
6:  NOP,6,0
```

The first column of the pseudo-code denotes the address of each instruction. The fourth instruction, 4: NOP,5,5, represents the critical section, and the

	Process 1	Process 2
IF_WHILE(b00),L_1,L_2	goto L_1	goto L_1
IF_WHILE(b01),L_1,L_2	IF FLAG0=b then goto L_1 else goto L_2	IF FLAG0=not(b) then goto L_1 else goto L_2
IF_WHILE(b10),L_1,L_2	IF FLAG1=b then goto L_1 else goto L_2	IF FLAG2=b then goto L_1 else goto L_2
IF_WHILE(b11),L_1,L_2	IF FLAG2=b then goto L_1 else goto L_2	IF FLAG1=b then goto L_1 else goto L_2

Figure 7.3: IF_WHILE

```
0:  SET_CLEAR(p0),1
1:  SET_CLEAR(p1),2
2:  IF_WHILE(p2),3,4
3:  IF_WHILE(p3),4,2
4:  NOP,5,5
5:  SET_CLEAR(p4),6
6:  NOP,6,0
```

Figure 7.4: Template 1

sixth instruction, 6: NOP,6,0, the part that is specific to each process. Each process is allowed to loop around the sixth instruction.

We then parameterize five instructions in Peterson's algorithm as in Figure 7.4. The safety and liveness of this parameterized code were verified as described in the next section.

7.2.2 Verification

The initial state of the state transition system is defined as a state that satisfies the following condition.

$$\texttt{PC1} = \texttt{PC2} = 0$$
$$\texttt{FLAG0} = \texttt{FLAG1} = \texttt{FLAG2} = 0$$

In the above condition, PC1 and PC2 denote the program counter of Process 1 and Process 2, respectively. Let $I(x)$ denote the predicate expressing the

condition, where x ranges over states. $I(x)$ holds if and only if x is the initial state.

The safety of the state transition system is defined as unreachability of an error state that satisfies the following condition.

$$\texttt{PC1} = \texttt{PC2} = 4$$

In an error state, both processes enter their critical section simultaneously. The system is safe unless it reaches an error state from the initial state. Let $E(x)$ denote the predicate expressing the condition.

Let $T(x, y)$ mean that there is a one-step transition from state x to state y, and $T^*(x, y)$ denote the reflexive and transitive closure of $T(x, y)$. The safety of the system is then expressed by the following formula.

$$\neg\exists xy.\ I(x) \wedge T^*(x, y) \wedge E(y)$$

We can describe the liveness for Process 1 of the system as non-existence of an infinite path on which

$$0 \leq \texttt{PC1} \leq 3$$

is always satisfied though both processes are infinitely executed. The condition, $0 \leq \texttt{PC1} \leq 3$, means that Process 1 is trying to enter its critical section. $S(x)$ denote the predicate expressing the condition.

We verify the liveness as follows. Let $T_1(x, y)$ denote the one-step transition relation for Process 1. $T_1(x, y)$ holds if and only if state y is obtained by executing Process 1 for one step from x. Similarly, let $T_2(x, y)$ denote the one-step transition relation for Process 2. Note that $T(x, y)$ is equivalent to $T_1(x, y) \vee T_2(x, y)$. We then define the following three predicates.

$$\begin{aligned} T_1'(x, y) &= T_1(x, y) \wedge S(x) \\ T_2'(x, y) &= T_2(x, y) \wedge S(x) \\ T'(x, y) &= T(x, y) \wedge S(x) \end{aligned}$$

For any predicate $Z(x)$ on state x, and any binary relation $R(x, y)$ on states x and y, we define the relation, denoted by $Z \circ R$, as follows.

$$(Z \circ R)(x) = \exists y.\ Z(y) \wedge R(x, y)$$

$Z \circ R$ is also a predicate on states.

For verifying the liveness of the system, we compute the following limit of predicates.

$$\begin{aligned} &S \circ T_1' \circ T'^* \circ T_2' \circ T'^* \circ \\ &T_1' \circ T'^* \circ T_2' \circ T'^* \circ \\ &T_1' \circ T'^* \circ T_2' \circ T'^* \circ \cdots \end{aligned}$$

This limit always exists, because the sequence of predicates

$$
\begin{array}{l}
S \circ T'_1 \circ T'^* \\
S \circ T'_1 \circ T'^* \circ T'_2 \\
S \circ T'_1 \circ T'^* \circ T'_2 \circ T'^* \\
S \circ T'_1 \circ T'^* \circ T'_2 \circ T'^* \circ T'_1 \\
S \circ T'_1 \circ T'^* \circ T'_2 \circ T'^* \circ T'_1 \circ T'^* \\
\cdots\cdots
\end{array}
$$

is monotonically decreasing. For example, we can prove the second predicate is smaller than the first one as follows. $S \circ T'_1 \circ T'^* \circ T'_2 \subseteq S \circ T'_1 \circ T'^* \circ T' \subseteq S \circ T'_1 \circ T'^*$.

Let us denote this limit by S'. It expresses the beginning of an infinite path on which S always holds and both processes are infinitely executed, i.e., a state satisfies the limit if and only if there exists such an infinite path from the state. The liveness is then equivalent to unreachability from the initial state to a state satisfying the limit.

$$\neg\exists xy.\ I(x) \wedge T^*(x, y) \wedge S'(y)$$

The liveness for Process 2 can be symmetrically described as that for Process 1. The whole system satisfies the liveness if it holds for both processes.

Since there are 7 pseudo-instructions in the pseudo-code, the program counter of each process can be represented by three bits. Therefore, a state in the state transition system can be represented by nine boolean variables; three are used to represent the program counter of each process, and three for the three shared variables.

There are five parameters in the pseudo-code, so fifteen boolean variables are required to represent the parameters. Let p denote the vector of the fifteen boolean variables. All the predicates and relations introduced so far are considered indexed by p. For example, we should have written $T_p(x, y)$.

Predicates such as $I_p(x)$ and $S_p(x)$ contain 24 boolean variables, and relations such as $T_p(x, y)$ contain 33 boolean variables. All these predicates and relations are represented by OBDDs (ordered binary decision diagrams)

We employed the BDD package developed by David E. Long and distributed from CMU [59]. The verification program was written in C by hand. The essential part of the program is shown in Section 7.2.3.

Variables in predicates and relations are ordered as follows.

- variables in x first,
- variables in y (if any) next, and
- variables in p last.

We have tried the opposite order but never succeeded in verification.

By the above order, predicates such as $S_p(x)$ are decomposed into a collection of predicates on p as follows.

if ...x... **then** ...p...
else if ...x... **then** ...p...
else ...

The value of x is successively checked by a series of conditions, and for each condition on x, a predicate on p is applied.

Similarly, relations such as $T_p(x, y)$ are decomposed as follows.

if ...x...y... **then** ...p...
else if ...x...y... **then** ...p...
else ...

7.2.3 Program

In this section, we show the essential part of the verification program of Section 7.2.2. It employs the BDD package developed by David E. Long and distributed from CMU [59]. Functions beginning with "`bdd_`" are from the bdd package. They include, for example, the following functions.

- `bdd_not`: returns the negation of a BDD.
- `bdd_and`: returns the conjunction of BDDs.
- `bdd_or`: returns the disjunction of BDDs.

Their first argument is the bdd manager as defined in the bdd package [59] and is of the type `bdd_manager`. The second and third arguments are BDDs of the type `bdd`.

The function `bdd_exists` computes the existential quantification of a given BDD. Before calling `bdd_exists`, the array of variables to be quantified should be specified as follows.

```
bdd_temp_assoc(bddm, y, 0);
bdd_assoc(bddm, -1);
```

The function `bdd_rel_prod` computes the relational product of given BDDs. Semantically, it first computes the conjunction of the second and third arguments and then computes the existential quantification as `bdd_exists` does.

We also defined some auxiliary functions described below.

The C function `bdd_closure_relation` takes the following seven arguments and computes the closure of a given relation.

- `bdd_manager bddm`: the bdd manager.
- `int AND`: the flag specifying whether the closure is computed by conjunction (if it is 1) or disjunction (if it is 0).
- `bdd g`: the initial predicate.
- `bdd f`: the relation whose closure is computed. It contains variables in `x` and `y`.
- `bdd *x`: the array of the variables in `f` and `g`.
- `bdd *y`: the array of the variables in `f`.
- `bdd *z`: the array of temporary variables.

Let x denote the vector of variables in `x`, y the vector of variables in `y`. It is assumed that `g` is a predicate on x and `f` is a relation between x and y. If the value of the flag `AND` is 0, then the above function computes and returns the following predicate on y.

$$\exists x.\ \mathtt{g}(x) \wedge \mathtt{f}^*(x, y)$$

The return value of the function is of the type `bdd`.

Following are some more auxiliary functions.

- `bdd_rename`: renames variables in a BDD and returns the resulting BDD.
- `bdd_equal_int`: constrains an array of variables as a binary number. For example,

  ```
  bdd_equal_int(bddm, 3, pc11, 4)
  ```

  returns the bdd expressing the following condition.

  $$\mathtt{pc11[0]} = \mathtt{pc11[1]} = 0 \wedge$$
  $$\mathtt{pc11[2]} = 1$$

- `bdd_and3`: returns the conjunction of three BDDs.
- `bdd_or4`: returns the disjunction of four BDDs.

The function `smash` makes approximation as described in Section 7.4. It takes the bdd manager, the BDD to be collapsed, and the threshold value. The identifier `smash_threshold` must have been defined as a C macro.

Following is the main function of the verification program. In the initialization part, omitted from the following code, the transition relations are initialized as follows.

- `t`: the one-step transition relation.
- `t1`: the one-step transition relation for Process 1.
- `t2`: the one-step transition relation for Process 2.

States are represented by nine variables. In the main function, the arrays `x`, `y` and `z` store nine variables representing states. Since they are null terminated, their size is 10. `x` denotes the state before a transition and `y` the state after a transition. `z` consists of temporary variables. The pointers `pc01`, `pc02`, `pc11`, and `pc12` point to the program counters in `x` and `y`.

```
int main()
{
  bdd_manager bddm;
  bdd x[10];
  bdd *pc01 = &x[0];
  bdd *pc02 = &x[3];
  bdd *vs0 = &x[6];
  bdd y[10];
  bdd *pc11 = &y[0];
  bdd *pc12 = &y[3];
  bdd *vs1 = &y[6];
  bdd z[10], *p, *q;
  bdd t, t1, t2,
      initial, reachable, conflict;
  bdd starving, starving_t,
      starving_t1, starving_t2;
  bdd starvation, s, s0;
  bdd e;

  bddm = bdd_init();

  ...
  /* initialization */
  ...

  initial = bdd_equal_int(bddm, 9, y, 0);
```

```
  reachable = bdd_closure_relation(bddm, 0,
                                   initial,
                                   t, x, y, z);

#ifdef smash_reachable
  reachable = smash(bddm, reachable,
                    smash_reachable);
#endif

  conflict = bdd_and3(bddm, reachable,
                      bdd_equal_int(bddm, 3,
                                    pc11, 4),
                      bdd_equal_int(bddm, 3,
                                    pc12, 4));
  bdd_temp_assoc(bddm, y, 0);
  bdd_assoc(bddm, -1);
  conflict = bdd_exists(bddm, conflict);

  starving = bdd_or4(bddm,
                     bdd_equal_int(bddm, 3,
                                   pc01, 0),
                     bdd_equal_int(bddm, 3,
                                   pc01, 1),
                     bdd_equal_int(bddm, 3,
                                   pc01, 2),
                     bdd_equal_int(bddm, 3,
                                   pc01, 3));

  starving_t = bdd_and(bddm, starving, t);
  starving_t1 = bdd_and(bddm, starving, t1);
  starving_t2 = bdd_and(bddm, starving, t2);

  s = starving;
  do {
    printf("iteration for fairness\n");
    s0 = s;

    s = bdd_rename(bddm, s, x, y);
    bdd_temp_assoc(bddm, y, 0);
    bdd_assoc(bddm, -1);
    s = bdd_rel_prod(bddm, starving_t1, s);
    s = bdd_closure_relation(bddm, 0, s,
                             starving_t,
                             y, x, z);

    s = bdd_rename(bddm, s, x, y);
    bdd_temp_assoc(bddm, y, 0);
    bdd_assoc(bddm, -1);
    s = bdd_rel_prod(bddm, starving_t2, s);
```

```
    s = bdd_closure_relation(bddm, 0, s,
                             starving_t,
                             y, x, z);

  } while (s0 != s);

  s = bdd_rename(bddm, s, x, y);
  bdd_temp_assoc(bddm, y, 0);
  bdd_assoc(bddm, -1);
  starvation = bdd_rel_prod(bddm, reachable, s);

  e = bdd_and(bddm,
              bdd_not(bddm, conflict),
              bdd_not(bddm, starvation));
}
```

7.2.4 Result

We checked the safety and liveness of the system as described in Section 7.2.2. The experiments are made by Vectra VE 6/450 Series 8 (Pentium II 450MHz processor), from Hewlett-Packard.

- It took 0.853 seconds to check the original version of Peterson's algorithm. The result was, of course, true.
- It took 117.393 seconds to check the template of algorithms containing a fifteen-bit parameter.

Since the template contains fifteen boolean variables, checking the template amounts to checking 2^{15} instances of the template simultaneously. If we simply multiply 0.853 by 2^{15} and compare the result with 117.393, we found speed up of about 238 times.

The size of the BDD that represents the constraints on the parameters is 93. Only 16 instances of parameters satisfy the constraints. So, we found 15 variants of Peterson's algorithm. But they are essentially equivalent to Peterson's algorithm. In the variants, the interpretation of some shared variables is simply reversed.

7.3 Search for Variants of Peterson's or Dekker's Algorithm

In this section, we show the second case study. We apply the method described in the previous section to another template. Since Dekker's algo-

	Process 1	Process 2
SET_CLEAR($b0$),L	FLAG0 := b; goto L	FLAG0 := not(b); goto L
SET_CLEAR($b1$),L	FLAG1 := b; goto L	FLAG2 := b; goto L

Figure 7.5: Modified SET_CLEAR

rithm is very similar to Peterson's, we make a template which cover both algorithms.

7.3.1 Pseudo-code

We define another template which has both Dekker and Peterson's algorithms as instances. In order to define such a template with a small number of parameters, we modify instructions as follows.

SET_CLEAR(p),L
JUMP_IF_TURN(p),L_{00},L_{01},L_{10},L_{11}
JUMP_IF_FLAG_YOU(p),L_{00},L_{01},L_{10},L_{11}
NOP,L_1,L_2

The instruction NOP is the same as in the previous section.

The operator SET_CLEAR is slightly changed to reduce parameters. The parameter p of SET_CLEAR(p) has two-bit value. Each value of the parameter results in a pseudo-instruction as defined in Figure 7.5.

The operators JUMP_IF_TURN and JUMP_IF_FLAG_YOU have a three-bit parameter and four operands. The operands denote addresses in the pseudo-code. The intuitive meaning of the operands is that L_{00} points to the next address, L_{01} points to the address of the critical section, L_{10} points to the address of the beginning of the loop of the entry part, and L_{10} points to the current address. Each value of the parameter results in a pseudo-instruction as defined in Figures 7.6 and Figures 7.7.

We define a template as in Figure 7.8. Note that we fixed the values of some parameters to reduce the search space. In the template, each parameter has two-bit value. So the template contains sixteen boolean variables.

The template contains both algorithms as instances. Dekker's algorithm can be represented by the following pseudo-code.

```
0:  SET_CLEAR(11),1
1:  SET_CLEAR(11),2
```

	Process 1	Process 2
JUMP_IF_TURN($b00$),L_{00},L_{01},L_{10},L_{11}	IF FLAG0 = b THEN goto L_{00} ELSE goto L_{00}	IF FLAG0 = not(b) THEN goto L_{00} ELSE goto L_{00}
JUMP_IF_TURN($b01$),L_{00},L_{01},L_{10},L_{11}	IF FLAG0 = b THEN goto L_{01} ELSE goto L_{00}	IF FLAG0 = not(b) THEN goto L_{01} ELSE goto L_{00}
JUMP_IF_TURN($b10$),L_{00},L_{01},L_{10},L_{11}	IF FLAG0 = b THEN goto L_{10} ELSE goto L_{00}	IF FLAG0 = not(b) THEN goto L_{10} ELSE goto L_{00}
JUMP_IF_TURN($b11$),L_{00},L_{01},L_{10},L_{11}	IF FLAG0 = b THEN goto L_{11} ELSE goto L_{00}	IF FLAG0 = not(b) THEN goto L_{11} ELSE goto L_{00}

Figure 7.6: JUMP_IF_TURN

	Process 1	Process 2
JUMP_IF_FLAG_YOU($b00$),L_{00},L_{01},L_{10},L_{11}	IF FLAG2 = b THEN goto L_{00} ELSE goto L_{00}	IF FLAG1 = b THEN goto L_{00} ELSE goto L_{00}
JUMP_IF_FLAG_YOU($b01$),L_{00},L_{01},L_{10},L_{11}	IF FLAG2 = b THEN goto L_{01} ELSE goto L_{00}	IF FLAG1 = b THEN goto L_{01} ELSE goto L_{00}
JUMP_IF_FLAG_YOU($b10$),L_{00},L_{01},L_{10},L_{11}	IF FLAG2 = b THEN goto L_{10} ELSE goto L_{00}	IF FLAG1 = b THEN goto L_{10} ELSE goto L_{00}
JUMP_IF_FLAG_YOU($b11$),L_{00},L_{01},L_{10},L_{11}	IF FLAG2 = b THEN goto L_{11} ELSE goto L_{00}	IF FLAG1 = b THEN goto L_{11} ELSE goto L_{00}

Figure 7.7: JUMP_IF_FLAG_YOU

```
 0:  SET_CLEAR(11),1
 1:  SET_CLEAR(p1),2
 2:  JUMP_IF_FLAG_YOU(0p2),3,8,2,2
 3:  JUMP_IF_TURN(1p3),4,8,2,3
 4:  JUMP_IF_TURN(0p4),5,8,2,4
 5:  SET_CLEAR(p5),6
 6:  JUMP_IF_TURN(1p6),7,8,2,6
 7:  SET_CLEAR(p7),2
 8:  NOP,9,9
 9:  SET_CLEAR(p8),10
10:  SET_CLEAR(01),11
11:  NOP,0,11
```

Figure 7.8: Template 2

```
2:  JUMP_IF_FLAG_YOU(001),3,8,2,2
3:  JUMP_IF_TURN(100),4,8,2,3
4:  JUMP_IF_TURN(010),5,8,2,4
5:  SET_CLEAR(01),6
6:  JUMP_IF_TURN(111),7,8,2,6
7:  SET_CLEAR(11),2
8:  NOP,9,9
9:  SET_CLEAR(10),10
10:  SET_CLEAR(01),11
11:  NOP,0,11
```

The Peterson's algorithm can be represented by the following pseudo-code.

```
0:  SET_CLEAR(11),1
1:  SET_CLEAR(10),2
2:  JUMP_IF_FLAG_YOU(001),3,8,2,2
3:  JUMP_IF_TURN(110),4,8,2,3
4:  JUMP_IF_TURN(001),5,8,2,4
5:  SET_CLEAR(00),6
6:  JUMP_IF_TURN(100),7,8,2,6
7:  SET_CLEAR(00),2
8:  NOP,9,9
9:  SET_CLEAR(01),10
10:  SET_CLEAR(01),11
11:  NOP,0,11
```

7.3.2 Verification

Verification of the template goes almost in the same manner as in the previous section.

In this verification, the condition of an error state is

$$\texttt{PC1} = \texttt{PC2} = 8.$$

For the liveness, we modify the condition of the starvation loop of Process 1 as follows.

$$0 \leq \texttt{PC1} \leq 7$$

Since there are twelve pseudo-instructions in the pseudo-code, the representation of the program counter of each process requires four bits. Therefore, a state in the state transition system can be represented by eleven boolean variables. Sixteen boolean variables are required to represented the parameters.

7.3.3 Result

We checked the safety and liveness of the system on the same machine as in the previous section.

- It took 10.195 seconds to check the Peterson's algorithm. The result was, of course, true.
- It took 19.506 seconds to check the Dekker's algorithm. The result was, of course, true.
- It took 741.636 seconds to check the template of algorithms containing a sixteen-bit parameter.

We found speed up of about 900 times at least.

The size of the BDD that represents the constraints on the parameters is 62. There are about 400 solutions of the constraints. We found that just one solution represents the original Dekker's algorithm, and the remaining solutions essentially represent Peterson's algorithm.

7.4 Approximation

Remember that according to the variable ordering we adopted in BDDs, a predicate on states indexed by p is represented as a collection of sub-predicates on p as follows.

if $C_0(x)$ **then** $P_0(p)$
else if $C_1(x)$ **then** $P_1(p)$
else if $C_2(x)$ **then** $P_2(p)$
else ...

The sub-predicates, $P_0(p), P_1(p), P_1(p), \cdots$, occupy the branches of the entire predicate. We tried to reduce the size of such a predicate by collapsing some of the sub-predicates on p with a small Hamming distance. This is considered a first step towards using approximation in our approach.

The Hamming distance between two predicates $P_i(p)$ and $P_j(p)$ is the fraction of assignments to p that make $P_i(p)$ and $P_j(p)$ different. For example, the Hamming distance between $p_0 \wedge p_1$ and $p_0 \vee p_1$ is $1/2$, since they agree on the assignments $(p_0 = 0,\ p_1 = 0)$ and $(p_0 = 1,\ p_1 = 1)$, while they do not agree on the assignments $(p_0 = 0,\ p_1 = 1)$ and $(p_0 = 1,\ p_1 = 0)$.

Let θ be some threshold between 0 and 1. By collapsing sub-predicates $P_i(p)$ and $P_j(p)$, we mean to replace both $P_i(p)$ and $P_j(p)$ with $P_i(p) \vee P_j(p)$, provided that their Hamming distance is less than θ. After $P_i(p)$ and $P_j(p)$ are replaced with $P_i(p) \vee P_j(p)$, the size of the entire predicate is reduced because the BDD node representing $P_i(p) \vee P_j(p)$ is shared. We note that this collapsed predicate R' by replacing the disjunction is bigger than the original predicate R, i.e. $\forall x.\ R(x) \Rightarrow R'(x)$.

We made the following experiment. We first computed the reachability predicate $R_p(y)$ defined as follows.

$$\exists x.\ I_p(x) \wedge T_p^*(x, y)$$

We then collapsed $R_p(y)$ according to some threshold θ, and continued verification using the collapsed $R'_p(y)$.

The discovered algorithms by using $R'_p(y)$ always satisfy the safety and the liveness. The expression of safety is $\neg\exists y.\ R_p(y) \wedge E(y)$. Because the collapsed predicate is bigger, $\forall p.\ (\neg\exists y.\ R'_p(y) \wedge E(y)) \Rightarrow (\neg\exists y.\ R_p(y) \wedge E(y))$. If the algorithm with parameter p satisfies the safety under the collapsed $R'_p(y)$, it satisfies the safety. This discussion can be used for the liveness, because the expression of the liveness is $\neg\exists y.\ R_p(y) \wedge S'(y)$.

We successfully obtained constraints on the parameters. We summarize the results in Figure 7.9. In the figure, if the size of the final result is marked as '$-$', it means that verification failed, i.e., no instantiation of parameters could satisfy the safety and liveness.

Template	Threshold θ	Size of $R_p(y)$		Size of final result	
		w/ collapse	w/o collapse	w/ collapse	w/o collapse
Template 1	0.03	175	278	38	93
Template 1	0.05	145	278	–	93
Template 2	0.15	50	429	30	62
Template 2	0.20	39	429	–	62

Figure 7.9: Results of collapsing

7.5 Discussion

We checked the safety and liveness of the template using BDDs, and successfully obtained the constraints on the parameters. We compared the time required for verifying a single concrete algorithm — Peterson's algorithm (or Dekker's algorithm) — with that for checking the template, and gained speed-up of more than two hundred times.

The result of the second case study might suggest that Peterson's algorithm could be discovered as a variant of Dekker's algorithm. In Chapter 6, we searched for variants of Dekker's algorithm, but we could not find Peterson's. This was because we fixed the finishing part as that of the original version of Dekker's. Designing an algorithm space that contains interesting algorithms is not easy. In these case studies, we found only known algorithms.

Analysis of the resulting constraints obtained by verification was also a difficult task. We examined all solutions of the constraints one by one. It was not easy to recognize which solutions were essential.

Using abstract BDDs [20] is another approach to reduce the size of BDDs. It is similar to ours in the sense that abstract BDDs are obtained by merging BDD nodes whose abstract values coincide. Abstract values of BDD nodes are given in advance. In our case, since it is difficult to define such abstraction before the search, we dynamically collapse the algorithm space according to the Hamming distance of BDDs.

Chapter 8

Formal Proof of Abstract Model Checking of Concurrent Garbage Collection

Although model checking is a useful automatic verification method, the state explosion problem is a major drawback of the method. Abstract model checking is a solution to this problem. In abstract model checking, an abstract system is constructed from the original one, often called a concrete system, and the correctness of the abstract one is verified by ordinary exhaustive search. The abstraction relation defines how a state in the concrete system is abstracted to a state in the abstract one.

If the abstraction relation hold some conditions, then the safety of the abstract system implies that of the concrete one. We gave such conditions in our own framework of abstract model checking for safety of programs in Chapter 4.

In Chapter 3, we verified the safety of concurrent garbage collection algorithms by abstract model checking. But we did not formally prove that the abstract relation satisfies the conditions required to guarantee the safety of the concrete system from that of the abstract one. In this chapter, we formalize and prove those conditions using the proof assistant, HOL [35], to complete verification.

8.1 Preliminaries

In this section, we recall our framework of abstract model checking for the safety in Chapter 4.

We assume that a finite set of atomic commands is given. Each atomic

command is interpreted to make a transition between states. Formally, a transition is a binary relation between states.

A program is in one of the following forms: atomic command, sequential execution of programs, nondeterministic choice of programs, if-statement, and while-statement. Based on the interpretation of atomic commands, a program also makes a transition between states. So, the set of states and a program made of atomic commands defines a state transition system.

We assume two kinds of states, concrete states and abstract states, and give two interpretations to each atomic command. The concrete interpretation of an atomic command is a transition between concrete states, while the abstract interpretation defines a transition between abstract states. The set of concrete states is written C, and that of abstract states A. Formally, the concrete interpretation of an atomic command a is a binary relation on C, which is written $[\![a]\!]_C \subseteq C \times C$. Similarly, the abstract interpretation is written $[\![a]\!]_A \subseteq A \times A$.

The abstraction relation between an abstract state and a concrete state is given by $\alpha \subseteq A \times C$.

The safety of a state transition system means that the system never falls into a unsafe state from the initial states. The set of initial concrete states and that of initial abstract states are written $I_C \subseteq C$ and $I_A \subseteq A$, respectively. The set of safe concrete states and that of safe abstract states are written $S_C \subseteq C$ and $S_A \subseteq A$, respectively.

Our final goal is to show the safety of the concrete system. It can be verified by model checking of the concrete state transition system, which is, however, not feasible if the number of concrete states is too large or infinite. Because the number of abstract states is relatively smaller than that of concrete states, model checking of the abstract system may be feasible. If the safety of the abstract system guarantees that of the concrete one, model checking of the safety of the abstract one is sufficient for the final goal. The following theorem that guarantees this.

> If $\alpha, I_C, I_A, S_C, S_A$ satisfy
>
> - $y \in I_C \Rightarrow \exists x \in I_A, (x, y) \in \alpha$,
> - $x \in S_A, (x, y) \in \alpha \Rightarrow y \in S_C$,
>
> and $(\alpha; [\![a]\!]_C) \subseteq ([\![a]\!]_A; \alpha)$ holds for every atomic command a, then for any program, the safety of the abstract system implies that of the concrete one.

In this theorem, the operator ";" composes two relations. The last formula

can be rewritten as follows.

$$\exists y.((x,y) \in \alpha \ \wedge \ (y,z) \in [\![a]\!]_C) \Rightarrow \exists w.((x,w) \in [\![a]\!]_A \ \wedge \ (w,z) \in \alpha)$$

8.2 Formalization

We formalize the abstraction employed in Chapter 3 in HOL [35]. As mentioned in the previous section, we have to formalize the following data of the concurrent garbage collection algorithm.

- C : Concrete states of GC.
- I_C, S_C : Initial states and safe states of C.
- $[\![a]\!]_C$: Concrete transitions for all atoms.
- A : Abstract states of GC.
- I_A, S_A : Initial states and safe states of A.
- $[\![a]\!]_A$: Abstract transitions for all atoms.
- α : Abstract relation between A and C.

8.2.1 Concrete States

In Chapter 3, we introduced arrays of registers and cells in the concrete model of concurrent garbage collection algorithms. Each register contains a pointer, which is an index of a cell in the heap, or `nil`. Each cell contains a color and a pointer. The kinds of colors are *White*, *Gray*, *Black* and *Free*. The mutator and the collector run in parallel. The collector has four states. In order to distinguish the state of the collector from that of the entire system, the collector state is called the collector step. The collector has four steps: Shade, Mark, Append and Unmark. We formalize the color and the step as datatypes of HOL as follows.

```
Hol_datatype `color =  Free | White | Gray | Black`;
Hol_datatype `step = Shade | Mark | Append | Unmark`;
```

We use the `num` type, the type of natural numbers in HOL, for indices of the arrays. We use an `option` type to formalize the type of pointers. The `option` type of `num` is written `num option` in HOL. A term of the type `num option` is either `NONE`, or of the form `SOME n`. We regard `SOME n` as the index `n`, and `NONE` as `nil`.

The array of registers is a term of the following type.

```
num -> num option
```

It is the type of a function from `num` to `num option`. The array of cells is a term of the following type.

```
num -> color # num option
```

The operator `#` makes the direct product of two types. We use two variables `rb` and `hb` as the upper bounds of indices of registers and cells.

A state of the entire system is represented by values of the following five variables.

```
(rb:num) (hb:num)
(s:step) (r:num->num option) (h:num->color#num option)
```

The right side of ":" shows the type of each variable. Since the values of `rb` and `hb` never change, the latter three variables are essential.

8.2.2 Initial States and Safe States

At the initial state of the system, the collector step is *Shade*, the contents of all registers are `nil`, the contents of all cells are `nil`, and the colors of all cells are *Free*. We define the predicate `initial` to judge whether a state is initial or not.

```
val initial_DEF =
  new_definition("initial_DEF",
``!(rb:num) (hb:num)
   (s:step) (r:num->num option) (h:num->color#num option).
   initial rb hb s r h =
   (s=Shade) /\ (!i. r i = NONE) /\ (!k. h k = (Free,NONE))``);
```

In the proof assistant and also in this chapter, a term of HOL is surrounded part by "``". The symbol "`!`" denotes the universal quantifier, and "`/\`" means conjunction. The juxtaposition `r i` means the application of the function `r` to the argument `i`. The term `(Free,NONE)` denotes the pair of `Free` and `NONE`.

The system is safe if and only if no free cells are reachable from registers by tracing pointers. The reachability is inductively defined as follows.

- If a register contains an index, the cell with the index is reachable.
- If a reachable cell contains an index, the cell with the index is reachable.

We define the predicate `direct_reachable` to judge whether a cell with the index `k` is contained by some register.

```
new_definition("direct_reachable_DEF",
``!(rb:num) (hb:num) (r:num->num option) (h:num->color#num option)
   k.
   direct_reachable rb hb r h k=
   ?i.(i < rb) /\ (k < hb) /\ (r i = SOME k)``);
```

In this definition, "?" denotes the existential quantifier.

We use the inductive definition package of HOL to define the predicate `reachable`, which judges whether the cell with the index `k` is reachable.

```
val (REACHABLE_thm1,REACHABLE_thm2,REACHABLE_thm3) =
  IndDefLib.new_simple_inductive_definition [
  ``direct_reachable r h k
    ==>
    reachable r h k``,
  ``reachable (r:num->num option) (h:num->color#num option) k
    /\ IS_SOME (field (h k))
    ==>
    reachable r h (THE (field (h k)))``];
```

In this definition, the function `field` extracts the contents of a cell. The predicate `IS_SOME` is true when the argument is not `NONE`. The function `THE` returns `x` from `SOME x`.

The predicate `safe` to judge whether the system is safe is defined as follows.

```
val safe_DEF =
new_definition("safe_DEF",
``!(rb:num) (hb:num)
   (s:step) (r:num->num option) (h:num->color#num option).
   safe rb hb s r h =
   !k. (k < hb) ==> ~ (reachable rb hb r h k)``);
```

In this definition, "==>" means implication, and "~" negation.

8.2.3 Concrete Interpretation of Atomic Commands

The operations of the mutator and the collector are defined as atomic commands. The concrete interpretation of an atomic command is a relation

between states before and after its execution. For each atomic command, we define a predicate that takes two states as arguments.

For example, the mutator's operator Rnew allocates the cell with index `k` and assigns it to the register with index `i`. The predicate `Rnew` is defined as follows.

```
new_definition("Rnew_DEF",
``!(rb:num) (hb:num)
   (s:step) (r:num->num option) (h:num->color#num option)
   s' r' h' i k.
   Rnew rb hb s r h s' r' h' i k=
   ((i < rb) /\ (k < hb) /\ ~(s=Shade) /\ (color (h k)=Free)
    =>(s' = s) /\
      (!m. r' m=((m=i) => (SOME k) | r m)) /\
      (!n. h' n=((n=k) => (Gray,NONE) | h n))
    | (s'=s) /\ (r'=r) /\ (h'=h))``);
```

The operator takes `i` and `k` as arguments in addition to those of the two states. In this definition, the infix operator " `=> |` " represents the if-then-else expression in HOL.

The predicates for other operations are defined similarly. All definitions are straightforward.

8.2.4 Abstract States

Now, we mention the abstract states. An abstract state, as defined in Chapter 3, is a pair of a collector step and an abstract heap. An abstract heap is a set of abstract cells. An abstract cell is a tuple consisting of the following four components.

- the color of the corresponding concrete cell,
- `nil` if the concrete cell contains `nil`, or the color of the cell whose index is contained in the concrete cell,
- the boolean value which represents the direct reachability from registers, and
- the boolean value which represents the reachability.

Therefore, an abstract cell is represented by values of the following four variables.

```
(c:color) (f:color option) (d_r:bool) (reach:bool)
```

A set of abstract cells, i.e., an abstract heap, is represented by a characteristic function. The variable `ah` representing an abstract heap has the following type.

```
color->(color option)->bool->bool->bool
```

Therefore, an abstract state is represented by the following two variables.

```
(as:step) (ah:color->(color option)->bool->bool->bool)
```

8.2.5 Abstract Initial States and Abstract Safe States

The abstract initial states and the abstract safe states were defined in Chapter 3. We translate them as follows. We define two predicates, `abs_initial` and `abs_safe`, as follows. The former defines abstract initial states, and the latter safe states.

```
val abs_initial_DEF =
new_definition("abs_initial_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool).
   abs_initial as ah =
   (as=Shade) /\
   (!c f d_r r.
     ah c f d_r r = (c=Free) /\ (f=NONE) /\ (d_r = F) /\ (r = F))``);
val abs_safe_DEF =
new_definition("abs_safe_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool).
   abs_safe as ah =
   (!f d_r.
     ~ (ah Free f d_r T))``);
```

8.2.6 Abstract Interpretation of Atomic Commands

For each atomic command, we define the predicate representing its abstract interpretation. For example, the predicate `abs_Rnew`, which represents the abstract interpretation of Rnew, is defined as follows. The variables with dash-mark represent the abstract state after the operator is executed.

```
val abs_Rnew_DEF =
new_definition("abs_Rnew_DEF",
``!(as:step) (ah:color->color option->bool->bool->bool)
   as' ah'.
   abs_Rnew as ah as' ah' =
```

```
(as=as') /\
(!c f d_r r.
  ah' c f d_r r
  = (ah c f d_r r) \/
    (ah c f T   T /\ (d_r = F) /\ (r = F)) \/
    (ah c f T   T /\ (d_r = F) /\ (r = T)) \/
    (ah c f F   T /\ (d_r = F) /\ (r = F)) \/
    ((?f' d_r' r'.ah Free f' d_r' r')
     /\ (c = Gray) /\ (f = NONE) /\ (d_r = T) /\ (r = T)))``;
```

The abstract interpretations of some operations are rather complicated. So, the formal definitions of them are long. But that formalization is straightforward.

8.2.7 Abstract Relation

Finally, we have to define the abstraction relation in HOL. It defines how a concrete state is abstracted to an abstract one. The four attributes of an abstract cell can be calculated from a concrete state. We first define the predicate to judge whether an abstract cell, `c f d_r reach`, is derived from the concrete cell whose index is `k`, where the heap is `h` and the register array is `r`.

```
``!(rb:num) (hb:num) (r:num->num option) (h:num->color#num option) k
   (c:color) (f:color option) (d_r:bool) (reach:bool).
   abstract_cell rb hb r h k c f d_r reach=
   (k<hb) /\
   (c = color (h k)) /\
   (f= (IS_SOME (field (h k))
        => SOME (color (h (THE (field (h k)))))
        |  NONE))/\
   (d_r = direct_reachable rb hb r h k) /\
   (reach = reachable rb hb r h k)``;
```

In this definition, the function `color` denotes the projection that extracts a color from the contents of a cell.

The abstraction relation between a concrete state, `rb hb s r h`, and an abstract state, `as ah`, is now defined as follows.

```
val abstract_relation_DEF =
new_definition("abstract_relation",
``!(rb:num) (hb:num) (s:step)
```

```
 (r:num->num option) (h:num->color#num option)
 (as:step) (ah:color->color option->bool->bool->bool).
 abstract_relation rb hb s r h as ah =
 (s = as) /\
 (!d_r reach c f.
  (?k. abstract_cell rb hb r h k c f d_r reach)
  ==> ah c f d_r reach)‘‘);
```

8.3 Formal Proof

We have formalized the necessary data as above. The required conditions to guarantee the validity of abstract model checking are formulas of first order logic as mentioned in Section 8.1. Of course, formalization of first order formula is easy in HOL. Here are formalized conditions in HOL. The condition about initial states:

```
‘‘!(rb:num) (hb:num)
   (s:step) (r:num->num option) (h:num->color#num option)
   (as:step) (ah:color->color option->bool->bool->bool).
   initial rb hb s r h
   ==> ?as ah.(abs_initial as ah /\
               abstract_relation rb hb s r h as ah)‘‘
```

The formula representing the condition about safe states:

```
‘‘!(rb:num) (hb:num)
   (s:step) (r:num->num option) (h:num->color#num option)
   (as:step) (ah:color->color option->bool->bool->bool).
   safe rb hb s r h /\ abstract_relation rb hb s r h as ah
   ==> abs_safe as ah‘‘
```

The formula for atomic command Rnew:

```
‘‘!(rb:num) (hb:num) (s:step) (r:num->num option)
  (h:num->color#num option) s’ r’ h’
  (as:step) (ah:color->color option->bool->bool->bool)
  as’ ah’.
 ?s’’ r’’ h’’ i k.
  abstract_relation rb hb s’’ r’’ h’’ as ah /\
  Rnew rb hb s’’ r’’ h’’ s’ r’ h’ i k
 ==>
 ?as’’ ah’’.
```

```
abs_Rnew as ah as'' ah'' /\
abstract_relation rb hb s' r' h' as'' ah''''
```

The formulas for other atomic commands are given by replacing "Rnew" with the corresponding atomic command name.

We prove these formulas by backward reasoning. We use Boomborg-HOL interface (see Section 9.1). The formula of the condition of initial states is proven by the next short tactic.

```
ZAP_TAC our_ss [] THEN
EXISTS_TAC
``\c (f: color option) d_r r.
   (c=Free) /\ (f=NONE) /\
   (d_r = F) /\ (r = F)`` THEN
ZAP_TAC bool_ss [] THEN
ONCE_REWRITE_TAC[REACHABLE_thm3] THEN
ZAP_TAC our_ss []
```

The `ZAP_TAC` is a strong semi-automatic reasoning tactic in HOL. The tactics for other formulas are longer. But the construction of tactics was straightforward.

8.4 Discussion

We formalized the concrete model and the abstract model of the on-the-fly GC algorithm in HOL. The formalization was straightforward thanks to many datatypes of HOL and its inductive definition library. The construction of proofs was also straightforward.

Because we formally proved that the abstraction used in Chapter 3 satisfies the conditions that guarantee the validity of abstract model checking, the safety of the on-the-fly GC algorithm is now completely formally verified.

The concurrent GC algorithms have been formally verified in several studies [42, 49, 34, 80]. Those except [42] did not use model checking. They found some invariants by inspecting the concurrent GC algorithm, and then constructed the proof on theorem provers. Havelund [42] verified the algorithms by model checking only with a finite model.

Our verification was done by the following three steps.

1. An abstract model was designed.
2. The abstract model was verified by model checking.
3. The validity of our abstraction was formally proven.

We felt that these steps were light tasks in the following sense.

1. The design was simple.
2. Model checking was fully automatic.
3. Formal proving was straightforward.

As a result of our experience, we can conclude that abstract model checking is an effective verification technique.

Chapter 9

Towards Mechanical Proof

Abstraction used in abstract model checking must satisfy certain conditions to guarantee the correctness of the entire process of abstract model checking. Ideally, these conditions should be formally proven. In the previous chapter, we give a formal proof of the correctness of the abstraction used in the abstract model checking of concurrent GC algorithms. The formal proof was developed on a proof assistant system called HOL. In this chapter, in connection with formal proofs, we describe the research on the environment of proof assistants. We first propose "proving as editing paradigm" as a guideline for user interfaces of proof assistant systems. This guideline was actually employed in the formal proof in the previous chapter. We also describe a graphical user interface for the commutative diagrams that often appear in the field of program semantics.

(The contents of this chapter are published in [91, 56].)

9.1 Boomborg-HOL

We introduce an Emacs interface for writing HOL proof scripts in SML based on the Computing-as-Editing paradigm.

With a tactic-based proof assistant such as HOL [35] or Isabelle [71], the user (i.e., the writer of a proof script) interactively inputs tactics to decompose the top goal to subgoals. In this process of interactive theorem proving, the current goal of the proof assistant is always printed out when the user inputs a tactic (Figure 9.1). The history of interaction therefore becomes very long, but it is difficult to understand the proof with only the sequence of input tactics (Figure 9.2).

After finishing the entire proof of the top goal, many users of a tactic-based proof assistant then compose a large structured tactic from the se-

```
val it =
  ‘‘c + SUC x = SUC x‘‘

  ----------------------------
      ‘‘c = 0‘‘
      ‘‘c + x = x‘‘
   : goalstack
```

Figure 9.1: A printed current goal

```
g ‘(c=0) ==> !x . c + x = x‘;
e DISCH_TAC;
e INDUCT_TAC;
e (ASM_REWRITE_TAC [ADD_0]);
e (REWRITE_TAC [GSYM ADD_SUC]);
e (POP_ASSUM (fn th => REWRITE_TAC [th]));
save_thm("a_thm", top_thm());
```

Figure 9.2: A sequence of tactics

quence of interactively input tactics using tacticals such as `THEN` and `THENL` in HOL, which represent control structures (Figure 9.3). Such a structured tactic can prove the top goal at a single step. Some users also try to write a structured tactic as they prove the top goal. While editing a structured tactic, they successively send fragments of the tactic to the proof assistant by hand. Of course, they have to keep track of the relationship between the edited tactic and the status of the proof assistant. The reason to write structured tactics is to gain reusability of proof scripts, but the readability of a structured tactic is relatively low because no subgoals are shown.

In this section, we introduce an interface[1] for writing HOL proof scripts in SML based on the Computing-as-Editing paradigm (CAEP) [38]. In this approach, a proof script is considered as a document consisting of constraints, and checking a proof script means solving the constraints contained in the script. Moreover, those constraints can be solved as they are being edited. In this sense, the process of editing and solving constraints amounts to that of interactive theorem proving.

In contrast to other interfaces based on the CAEP (such as that for computer algebra [38]), the interface in this section does not require a new format for expressing constraints. Users can directly edit HOL tactics written

[1]The developed interface is available from the following URL. `http://nicosia.is.s.u-tokyo.ac.jp/boomborg`

in the syntax of SML. A proof script in SML is regarded as a constrained document, where each tactic in the document is solved as a constraint by the HOL prover.

With our interface, the user can write a proof script while interactively executing tactics. When a command called `check` is invoked under the text editor, the tactic just before the text cursor is locally executed and confirmed by the HOL prover. According to the result of this execution, the text of the proof script is changed; when the tactic has produced subgoals, an appropriate template for solving the subgoals is inserted.

Using our interface, the user can avoid the errors that arise when he or she composes a structured tactic from a successful sequence of tactics. In addition, the user can write a structured tactic in a very flexible manner. No specific order is imposed on which subgoal to solve first, and the user can switch the current subgoal simply by moving the text cursor. Since a tactic at the position of the text cursor is executed and confirmed locally, the user can check the slight change of a tactic immediately after he or she has modified the text of the tactic. This greatly helps the user to reuse existing proof scripts.

In our interface, to make local execution of a tactic efficient, subgoals (i.e., intermediate goals) can be explicitly embedded in a proof script, which also improves the readability of the proof script. It is not necessary to write such subgoals by hand, as they are usually inserted as a result of the local execution of a tactic.

The user can also employ proof-by-pointing [8]. At any position in a proof script, the user can pop up the current goal window (Figure 9.7), and point out a subterm in the current goal using a mouse. A tactic called a proof-by-pointing tactic is then generated and inserted into the proof script (Figure 9.8). Since the result of proof-by-pointing is expressed in the form of text, it is possible to undo, reuse or modify the result by deleting, copying or editing the corresponding text. In addition, the user can replay proof-by-pointing by specifying the proof-by-pointing tactic in a proof script.

We describe our interface, called `Boomborg-HOL`, in detail in Section 9.1.1. In Section 9.1.2, we explain how proof-by-pointing is performed in our interface. Section 9.1.3 is about the implementation of our interface. Section 9.1.4 summarizes the merits and demerits of our interface and gives an example of reusing a proof script.

9.1.1 Boomborg-HOL

In our interface (`Boomborg-HOL`) two functions, `CLAIM` and `BY`, are prepared to make subgoals (i.e. intermediate goals) in a proof script explicit. They are

```
val a_thm = store_thm("a_thm",
  ``(c=0) ==> !x . c + x = x``,
  DISCH_TAC THEN
  INDUCT_TAC THENL [
    ASM_REWRITE_TAC [ADD_0]
  ,
    REWRITE_TAC [GSYM ADD_SUC] THEN
    POP_ASSUM (fn th => REWRITE_TAC [th])
  ]);
```

Figure 9.3: A structural tactic

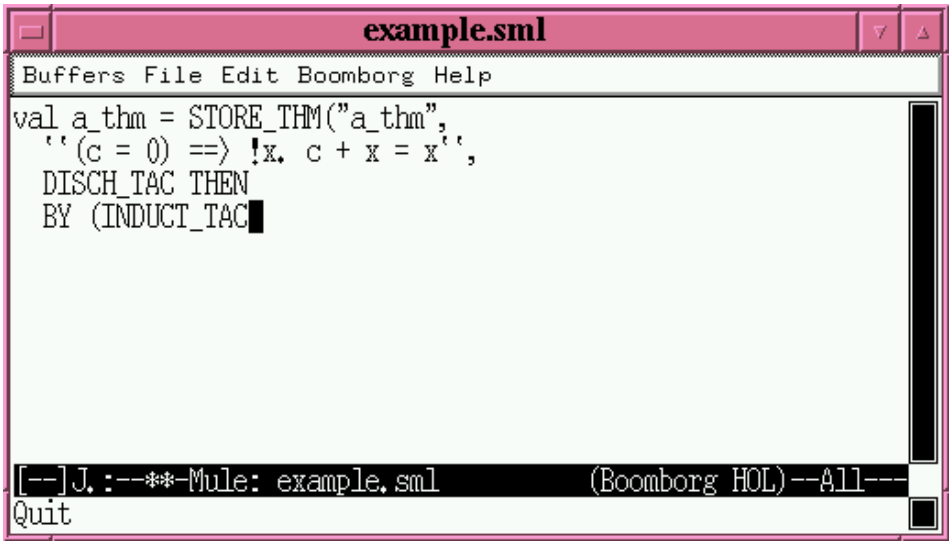


Figure 9.4: Before the `check` command

tacticals since they take a tactic as an argument. They are used as follows.

`CLAIM` "assumption part" "conclusion part" "tactic"

`BY` "tactic"

The assumption part and the conclusion part of a `CLAIM` specify the assumption and the conclusion of a goal as compared with the goal at its outer surrounding `CLAIM` or `BY` (or `STORE_THM`, a variant of `store_thm` in the HOL). The assumption part is a list whose elements are either of the form

`assume` A

or of the form

`forget` A.

The form `assume` A means that the assumption A is added to the goal at the outer `CLAIM` or `BY`. Since some assumptions may be deleted by a tactic, such as `POP_ASSUM`, we also support the form `forget` A. It means that the assumption A is deleted from the goal at the outer `CLAIM` or `BY`. The conclusion part is either of the form

`holds` C

or of the form

`thesis`.

The former simply specifies C as the conclusion of the goal at this `CLAIM`. The latter means that the conclusion of the goal at the outer `CLAIM` or `BY` has not been changed.

In our interface, the user edits a proof script that may contain `CLAIM`s and `BY`s in an Emacs buffer. While editing the script, the user can use the command `check`. It executes the tactic just before the text cursor and inserts each subgoal produced by the tactic using the tactical `CLAIM`.

After the `check` command is invoked in Figure 9.4, a `CLAIM` is inserted as in Figure 9.5. In Figure 9.5, the `CLAIM`

```
CLAIM [] (holds ``c + 0 = c``) (
```

means that there is no change in the assumption part of its goal as compared with the goal at the outer `BY`, while the conclusion is specified as ``` ``c + 0 = 0`` ```. Notice that the goal at the outer `BY` has the assumption ``` ``c = 0`` ```. This is not explicitly specified by the `CLAIM`. Using `BY`s, the user can avoid specifying evident changes of goals. If there was no `BY`, the `CLAIM`s in Figure 9.5 would be

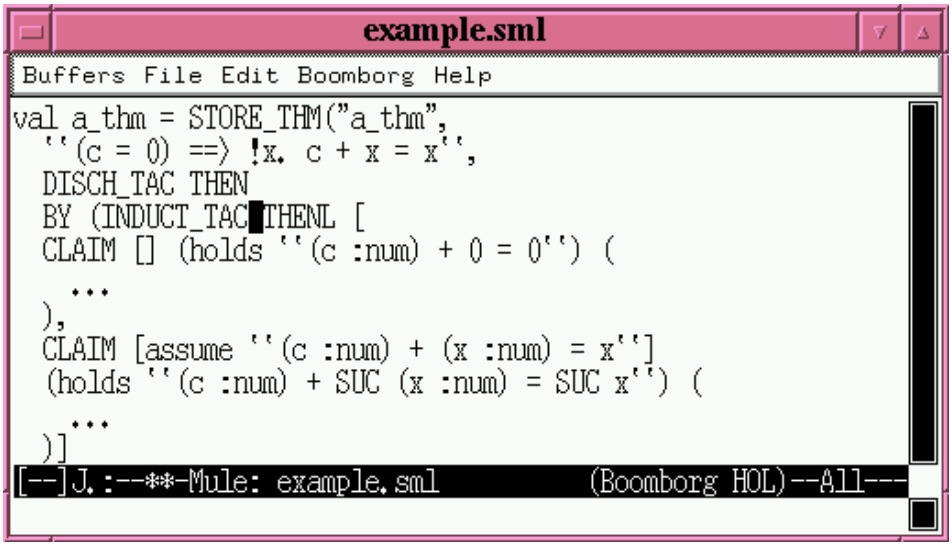


Figure 9.5: After the `check` command

```
CLAIM [assume ‘‘c = 0‘‘] (holds ‘‘c + 0 = c‘‘) (
```

and

```
CLAIM [assume ‘‘c + x = x‘‘,assume ‘‘c = 0‘‘]
(holds ‘‘c + SUC x = SUC x‘‘) (
```

In Figure 9.6, the `check` command shows the message `OK` at the message line of the editor because no subgoals remain. As in this example, the `check` command can be executed at any position in a proof script, even if the script is incomplete.

In our interface, the user does not directly interact with the HOL prover. The user just edits a proof script, and sends a fragment of the script to the HOL prover from time to time by invoking the `check` command. This process can be considered as constraint solving in the sense of the CAEP. Although the script may have many unsolved constraints, the user can solve any one without paying attention to the status of the HOL prover.

To support randomly solving constraints, we chose the state-less communication with the HOL for implementing our interface. Each time the `check` command is executed, the checked goal (the `CLAIM`ed goal) and the tactic that proves the goal are sent to the HOL prover.

Specification of subgoals by `CLAIM`s has two purposes. One is readability, as it is easier to understand the proof if the subgoals are explicitly embedded in the proof script. Another is to minimize the size of the code sent to the

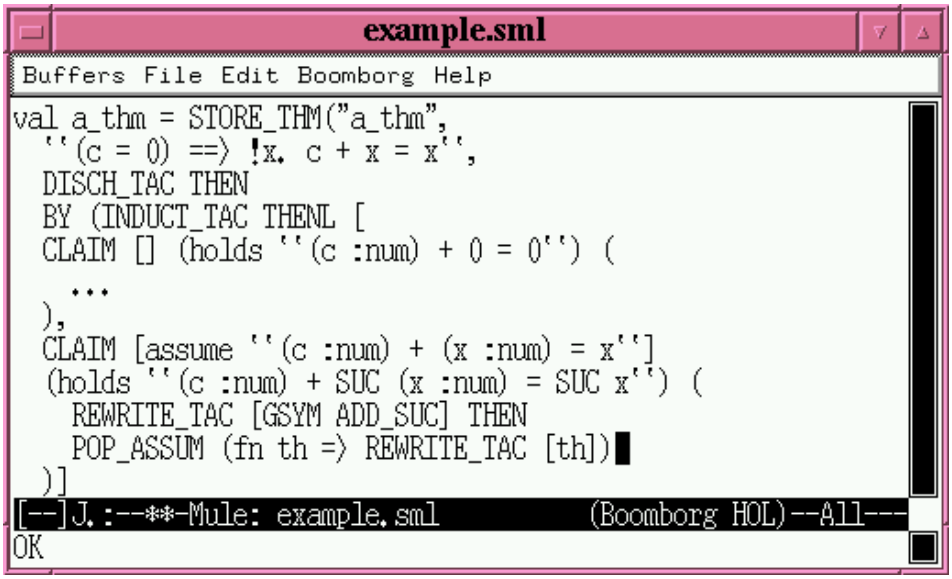


Figure 9.6: An example of using the `check` command

HOL prover. The `check` command assumes that the subgoal specified by an outer `CLAIM` is correct. This makes it only necessary to send the tactic between the text cursor and the outer `CLAIM`. Minimizing the size of the code has the following two consequences:

- The response of the HOL prover is improved, because it executes smaller tactics.
- It makes it possible to locally check a proof script. Even if a tactic is incomplete, one can check a part of the tactic if the code sent to the HOL is well-formed.

There are a few more commands supported by our interface.

- `check-claim-body`
- `check-claim-specs`

These two commands are available at `CLAIM`s.

The `check-claim-body` command checks whether the body tactic of the `CLAIM` proves the goal specified by the `CLAIM`. If the tactic can prove the goal, the message `OK` appears at the message line. Otherwise, an error is reported.

The `check-claim-specs` command recalculates the goal at the `CLAIM` by executing the tactic just before the `CLAIM`. If the obtained goal is as specified

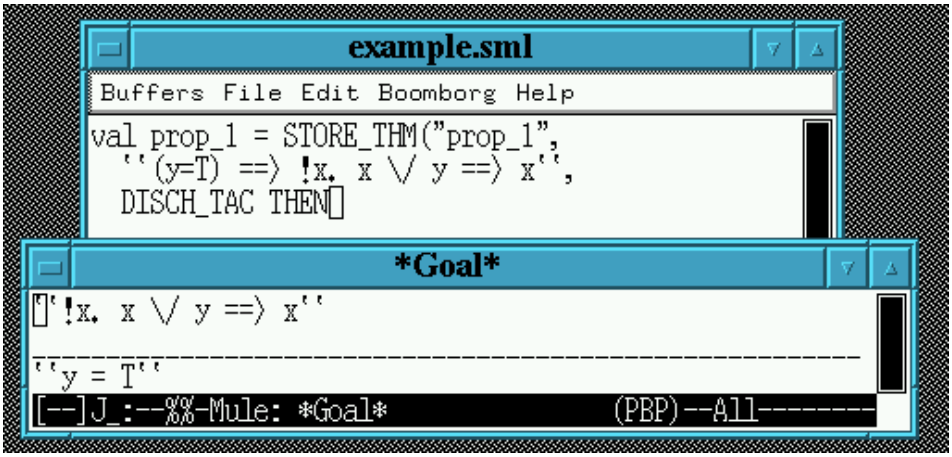


Figure 9.7: A window of the current goal

by the `CLAIM`, it merely shows the message `OK`. Otherwise, it replaces the assumption part and the conclusion part of the `CLAIM` so that they correctly specify the goal. This command is intended to be used for reusing a proof script. When the top goal of a proof script has been changed, it is possible to propagate the change through the script by repeatedly invoking this command. It is also considered as solving the constraint between a `CLAIM` and a tactic before it.

9.1.2 Proof-by-pointing

Proof-by-pointing [8] is a very useful interface for interactive theorem proving. Proof-by-pointing starts with pointing to a subterm in a goal. The goal is repeatedly decomposed by inference rules until the subterm appears at the top-level. For example, when the user points to the subterm `B` in the goal `?- A ==> B \/ C`, the implication and disjunction are decomposed and the user gets the new goal `A ?- B`. Many proof steps are executed by one pointing. Note that pointing is a graphical operation.

Our interface also supports proof-by-pointing. We first prepare a tactic to perform decomposition, called a proof-by-pointing tactic:

`PROOF_BY_POINTING` “subterm” “position” “instances”.

The “subterm” specifies a subterm in the conclusion of the current goal. The “position” argument is the index of the occurrence of the subterm; 0 means the first occurrence, 1 the second, etc. The “instances” argument is the list of instantiation of quantified variables. The call

```
val a_thm = STORE_THM("a_thm",
  ‘‘(y = T) ==> !x . x \/ y ==> x‘‘,
  DISCH_TAC THEN
  PROOF_BY_POINTING ‘‘x‘‘ 0 []
```

Figure 9.8: A proof-by-pointing tactic inserted

`PROOF_BY_POINTING` s n $[u_1, u_2, \cdots]$

means that the n-th subterm of s in the conclusion of the current goal is pointed to and quantified variables are instantiated by u_1, u_2, $\cdots$ in this order. For example, when `B` is pointed to in the goal `?- A ==> B \/ C`, the corresponding proof-by-pointing tactic is

```
PROOF_BY_POINTING ‘‘B :bool‘‘ 0 [].
```

In our interface, the user can pop up a window showing the current goal at any position in a proof script (Figure 9.7). In the window of the current goal, the conclusion appears at the top line, and the assumptions are represented below the underlines, as is HOL. The user can point to a subterm in the pop-up window using a mouse. A proof-by-pointing tactic is then generated and inserted at the given position. In Figure 9.7, if the user points to the first occurrence of `x`, then a proof-by-pointing tactic is generated and inserted as in Figure 9.8. The "position" is 0, meaning the first occurrence. The proof-by-pointing tactic is only inserted, and not executed. Our interface for proof-by-pointing is only for inputting a tactic; the `check` command is used to execute it. A similar approach to proof-by-pointing has been taken by Théry [96], and also by Bertot, Schreiber and Sequeira [9]. We discuss the differences between our approach and theirs in Section 9.1.5.

In the current implementation, the "instances" of an inserted tactic are always a null list; the user is expected to fill in this list as needed. Instead of a null list, the list of the names of binders could be inserted. In that case, the user will replace each binder with its corresponding term.

9.1.3 Implementation

Since our interface is implemented on Emacs and communicates with a HOL prover, its implementation consists of an Emacs Lisp program and an SML program for communication and manipulation of goals, including the definitions of `CLAIM` and `BY` as functions in SML.

Before describing the implementation of the `check` command, let us explain an Emacs Lisp function called `get-goal-at-point`. The `get-goal-at-point`

function calculates the goal at a CLAIM or BY, or at the beginning of a proof (i.e., at STORE_THM). If the text cursor is at the beginning of a proof, it returns the goal as specified by STORE_THM. If the text cursor is at a CLAIM, since the difference between the goal at the CLAIM and the goal at the outer CLAIM or BY is specified by the arguments of the CLAIM, it returns the goal at the CLAIM by calculating it with the difference; the goal at the outer CLAIM or BY is calculated by a recursive call of get-goal-at-point. If the text cursor is at a BY, get-goal-at-point essentially uses the HOL prover. It sends to the HOL prover the goal at the outer CLAIM or BY (obtained by a recursive call) and the tactic between the BY and the outer CLAIM or BY, and returns the result of the tactic.

Note that the tactic is sent to the HOL prover only when the cursor is at a BY. When the cursor is at a CLAIM, get-goal-at-point assumes that the arguments of the CLAIM are correct and does not send the tactic to the HOL prover.

The check command executes the tactic between the text cursor and the outer CLAIM or BY by sending it to the HOL prover with the goal at the outer CLAIM or BY (obtained by get-goal-at-point). It then inserts the subgoals returned by the tactic into the buffer with appropriate control structures using THEN or THENL.

Notice that the overall implementation is state-less in that the HOL prover does not keep any state for the communication with the interface.

To implement the proof-by-pointing interface, we need to know the position of each subterm in a goal that is shown in the pop up window. We modified the pretty printer of the HOL so that it records the position of each subterm as it prints out a given goal. (Unfortunately, we could not make use of the extensible pretty printer of the HOL for this purpose. We had to directly modify the source code of the pretty printer.)

In the HOL, every term surrounded by ‘‘ must be completely typed by itself after it is parsed; it cannot be polymorphic in the sense of Milner and Damas. So terms that appear in the arguments of a CLAIM must also be completely typed. The default HOL pretty printer has two modes. In one mode it never shows the types in a term, while in the other mode it prints every variable or constant with its type. If arguments of a CLAIM are printed in the former mode, they may cause errors when they are sent to the HOL prover because they may not be completely typed. In the latter mode, however, outputs become too noisy. In order to solve this problem, we modified the original HOL pretty printer and made a new one that shows types as little as possible. This pretty printer shows a type only on one occurrence of each variable or constant; and if the type of a variable or constant can be easily inferred, the type is not shown.

9.1.4 Experience

In this section, we describe some of the insights we have gained through the experience of writing HOL proof scripts using our interface.

With respect to writing a new proof script, our interface has the following obvious merits:

- It is extremely easy to undo and modify tactics.
- Modified tactics can be immediately checked.
- Subgoals can be proven in any order.
- Structural errors seldom occur.

When we write a new proof script, undoing tactics often occurs, as the HOL manual [86] says "Often (we are tempted to say *usually!*) one takes a wrong path in doing a proof." Although HOL provides the `backup` command for undoing, it can only undo a specified number of proof steps. In our interface, on the other hand, undoing tactics amounts to deleting the text of the tactics on the editor buffer. This is obviously intuitive and clear. There is no restriction on the number of steps to be undone. One can even use the `undo` command of Emacs for simply undoing the proof steps up to the most recent one.

For undoing tactics in an unstructured sequential proof script, more intelligent functions are needed (e.g., see [76]), by which the related parts are also removed and the remaining parts are modified appropriately. Because our interface manipulates a structural proof script directly, we did not feel such functions necessary.

Modifying erroneous tactics is also very common when writing a new proof script. Usually, modifying tactics is a combination of undoing old tactics and then writing new ones. In our interface, modifying tactics amounts to modifying the text of the tactics on the editor buffer. This is also intuitive and clear. In particular, the user can modify the tactics even if they were written at some very old step. The modified tactics can be immediately checked by the `check` command, because tactics are written and maintained only on the editor buffer.

It is also common to clean up once checked tactics. For example, one often wants to replace

```
REWRITE_TAC[a_thm] THEN ASM_REWRITE_TAC[]
```

with

```
ASM_REWRITE_TAC[a_thm].
```

After such modification, one can immediately check the modified part by `check`.

Since the ordinary interface of HOL holds the current goal, to which tactics are applied, the user has to use the `rotate` command when he or she wants to prove another goal. Our interface does not have the current goal. The user can write proofs of the subgoals in any order (see Figure 9.5 and 9.6). The ellipses `...` show the incomplete parts of the proof script. The user can prove those parts in any order.

When the `check` command is executed, control structures (`THEN` or `THENL`) and parentheses are automatically inserted. This simple structure-editor-like functions have almost eliminated the possibility of structural errors in our experience.

Our interface was also found very powerful for reusing an existing proof script. In particular, it is of a great help when a proof script is modified to prove a goal that is similar to the original one. Let us give a very simple example. Assume that the goal

```
‘‘!x. x + 0 = 0 + x‘‘
```

has been proved by the following proof script.

```
val a_thm = STORE_THM("a_thm",
  ‘‘!x. x + 0 = 0 + x‘‘,
  INDUCT_TAC THENL [
  CLAIM [] (holds ‘‘0 + 0 = 0 + 0‘‘) (
    REWRITE_TAC[]
  ),
  CLAIM [assume ‘‘(x :num) + 0 = 0 + x‘‘]
  (holds ‘‘SUC (x :num) + 0 = 0 + SUC x‘‘) (
    REWRITE_TAC[GSYM ADD_SUC] THEN
    CLAIM [] (holds ‘‘SUC (x :num) + 0 = SUC (0 + x)‘‘) (
      POP_ASSUM (fn th => REWRITE_TAC[GSYM th]) THEN
      CLAIM [forget ‘‘(x :num) + 0 = 0 + x‘‘]
      (holds ‘‘SUC (x :num) + 0 = SUC (x + 0)‘‘) (
        REWRITE_TAC [ADD_0]
      )
    )
  )]
);
```

We now want to prove the following new goal.

```
val b_thm = STORE_THM("b_thm",
  ``!x. x * 0 = 0 * x``,
```

This new goal could be proved if the above proof was generalized according to the inductive structure of the proof, and the generalized proof was appropriately instantiated to fit the new goal. In the following, however, we try to reuse the above proof script by partially modifying the text of the script, as we often do when we use the HOL. Since the proof should be similar to that for the addition, we just rewrite the goal in the original proof script and use the `check-claim-specs` command at the first `CLAIM` and the second `CLAIM`. The arguments of the `CLAIM` are then rewritten and the entire proof script becomes as follows.

```
val b_thm = STORE_THM("b_thm",
  ``!x. x * 0 = 0 * x``,
  INDUCT_TAC THENL [
  CLAIM [] (holds ``0 * 0 = 0 * 0``) (
    REWRITE_TAC[]
  ),
  CLAIM [assume ``(x :num) * 0 = 0 * x``]
  (holds ``SUC (x :num) * 0 = 0 * SUC x``) (
    REWRITE_TAC[GSYM ADD_SUC] THEN
    CLAIM [] (holds ``SUC (x :num) + 0 = SUC (0 + x)``) (
      POP_ASSUM (fn th => REWRITE_TAC[GSYM th]) THEN
      CLAIM [forget ``(x :num) + 0 = 0 + x``]
      (holds ``SUC (x :num) + 0 = SUC (x + 0)``) (
        REWRITE_TAC [ADD_0]
      )
    )
  )]
);
```

Since the rewritten subgoal ``0 * 0 = 0 * 0`` can been proven by the original tactic , we invoke the `check` command. Our interface then replies with `OK`. But the second one can been proven by `MULT_SUC`, we replace `GSYM ADD_SUC` with `MULT_SUC`. In a similar fashion, we repeat the `check-claim-specs` command and rewrite those parts that do not prove the corresponding `CLAIM`. After we invoke the `check-claim-specs` command at the last `CLAIM` and replace `ADD_0` with `MULT_0`, the `check` command produces a new subgoal as follows.

```
val b_thm = STORE_THM("b_thm",
```

```
  ``!x. x * 0 = 0 * x``,
  INDUCT_TAC THENL [
  CLAIM [] (holds ``0 * 0 = 0 * 0``) (
    REWRITE_TAC[]
  ),
  CLAIM [assume ``(x :num) * 0 = 0 * x``]
  (holds ``SUC (x :num) * 0 = 0 * SUC x``) (
    REWRITE_TAC[MULT_SUC] THEN
    CLAIM [] (holds ``SUC (x :num) * 0 = 0 + 0 * x``) (
      POP_ASSUM (fn th => REWRITE_TAC[GSYM th]) THEN
      CLAIM [forget ``(x :num) * 0 = 0 * x``]
      (holds ``SUC (x :num) * 0 = 0 + x * 0``) (
        REWRITE_TAC [MULT_0] THEN
        CLAIM [] (holds ``0 = 0 + 0``) (
          ...
        )
      )
    )
  )]
);
```

Notice that this result is different from the corresponding part for the addition. We must add the tactic `REWRITE_TAC[ADD_0]` here. The proof is then completed.

When the user changes not only goals but also axioms and theorems, tools for maintaining the dependencies among them, such as those in [77], are useful. We think that such tools can be combined with our interface.

We also noticed some demerits of our interface. One is that proof scripts written with our interface tend to become long because they contain many `CLAIM`s. In particular, when a goal term is large, each `CLAIM` may consume many lines. In such cases, the user has to delete `CLAIM`s by hand. But the user has to pay attention to the effect of deleting a `CLAIM` from a finished script; replacing a `CLAIM` with a `BY` is safe, but deleting a `CLAIM` may be harmful because the specification of the goal at a `CLAIM` is relative. The user can resolve this conflict by using the `check-claim-specs` command.

Another inconvenient point is that when an error arises, the user has to investigate the cause of the error by looking at the output of HOL. Since typographic errors or careless mistakes often occur, we consider that error handling is a serious problem to be solved.

9.1.5 Discussion

We developed an Emacs interface for writing HOL proof scripts based on the Computing-as-Editing paradigm. By using this interface, the user can edit structural proof scripts while interacting with the HOL prover. The interface also supports proof-by-pointing to input tactics.

The user interface developed by Théry [96] helps to construct structured tactics that can be incomplete. An incomplete tactic contains a hole for an unproved subgoal. To fill in the hole, the user can perform one of the following actions:

- directly fill in the hole by writing a tactic,
- select a candidate tactic from a menu,
- do proof-by-pointing,
- rewrite a subterm in the subgoal by choosing from possible rewrites shown in a menu.

His user interface, however, is structure-oriented. Incomplete tactics can only be edited according to their structure. In our interface, which is text-based, the user can edit incomplete tactics as text. Although we do not support menu-based actions of Théry's interface, it is easy to incorporate them in our interface. Similar comments apply to Griffin's interface [37].

We think that directly editing tactics under Emacs, which is a structure-free text editor, is more flexible, and allows for reusing existing tactics. One can copy an existing proof script and modify it as text. Subgoals shown with `CLAIM` make proof scripts more readable and easy to reuse.

TkHol [89] is a graphical user interface for HOL. By using TkHol, the user automatically gets a structured proof script from a sequence of input tactics.

As Merriam and Harrison [65] pointed out, one problem of proving with graphical user interfaces (GUIs) is their weakness in reusability. When the user modifies the top goal of a proof script, he or she may want to reuse as many parts of the old proof script as possible, but it is difficult to do so under GUIs because the history of interaction is either lost or not editable. Supporting editable records of interaction is essential for reuse.

In our proof-by-pointing interface, a user's pointing is inserted into a proof script as a tactic, and the user can reuse these tactics as other parts of a proof script. This suggests a way to utilize various kinds of GUI (e.g., drag-and-drop rewriting [7]) in our interface.

In order to utilize GUIs, some conditions are required:

- one operation on a GUI has to be translated into one tactic,
- the translated tactic has to cause the same effect as the operation, and
- from the translated tactic, it should be possible to replay the operation.

Although there are many ways to satisfy the first condition, the reusability of translated tactics is heavily affected by the design of the translation. In our case of proof-by-pointing, "pointing of a subterm" could be represented by one number, the position of the subterm from the beginning of the entire goal. A tactic with such limited information can hardly be reused. For such reasons, independence of translation from the context of interaction is important.

To achieve the second condition in the case of proof-by-pointing was relatively easy, and we were able to implement the required function in only a hundred lines.

Once translation satisfying these conditions is implemented, a new GUI becomes available in our interface. We expect that various kinds of GUI will be introduced in this manner.

Our proof-by-pointing tactic is almost the same as `FINGER_TAC` proposed by Théry [96]. In his interface, the current goal is displayed in the goal buffer and pointing in the goal buffer by the user generates an application of `FINGER_TAC`. The only difference is that `FINGER_TAC` accepts the address of the subterm as a sequence of `f`, `a` and `b`, where `f` denotes the left part of an application, `a` the right part of an application, and `b` the body of an abstraction. In our tactic, on the other hand, the pointed subterm is explicitly shown, so the tactic is more readable.

Bertot, Schreiber and Sequeira have also developed an interface for proof-by-pointing on XEmacs [9]. In their interface, the current goal is displayed in the goal buffer and pointing in the goal buffer by the user generates plain proof commands for the LEGO proof assistant, and inserts them in a buffer for saving the operation in a proof script. Their interface translates a pointing into a sequence of primitive commands such as `intros` and `impE`.

In contrast, our interface as well as Théry's interface inserts a proof-by-pointing tactic, so that it is possible to replay the operation for proof-by-pointing using the inserted tactic. As mentioned above, we think that replayability is a key to reusing graphical operations. Proof-by-pointing tactics also make proof scripts more compact and readable.

Our interface was designed and implemented for tactics written in SML. Since the interface does not understand SML, it only extracts structures of tactics by `CLAIM`, `BY` and parenthesis. Extending the interface to extract

more structures seems hopeless because SML is a general-purpose programming language. Therefore, our interface seems more suitable for proof scripts written in a special-purpose language for formal proofs, such as Mizar [66]. In Mizar, proofs are written with predefined structures, each of which can be easily recognized. We think that the Mizar mode for the HOL [41] will be helpful for constructing a user interface by our approach.

In the future, we plan to implement drag-and-drop rewriting [7] and other kinds of GUI under our framework.

Our experience so far suggests that the cost of communication between Emacs and HOL is not high, but it is important to investigate how the communication cost increases as we treat huge goals with our interface.

It is also important to solve the problem that text inserted by `check` becomes long for huge goals. The representation of terms needs to be improved, for example, by embedding huge terms into hyper text [94] or by using annotations and referring to huge terms by labels [51].

As mentioned in Section 9.1.4, error handling is another important issue.

9.2 Proof by Pasting

In category theory, commutative diagrams are extensively used to help understand (informal) proofs written in natural language and logical formulas. So, it seems natural to require the use of commutative diagrams in *formal* theorem proving in category theory too. In fact, the difficulty in understanding formal proof is one of the major obstacles in formal mathematics, so assistance in obtaining an intuitive idea of what is written would be of great value. Therefore, we are developing a user interface for proof assistant systems, based on commutative diagrams, and we report its functional specification.

In the course of developing our interface, we encountered a general user interface framework based on relations, rather than by simpler functions. This seems a new framework and we believe it is a contribution to visual logical systems, and we describe it in considerable depth.

What we wish is to represent formulas such as

$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow \\ \quad\quad\quad (\exists h)(\mathrm{source}(h) = X \wedge \mathrm{target}(h) = A \times B) \wedge \\ \quad\quad\quad\quad (f = h\,;p_1) \wedge (g = h\,;p_2)\end{array} \tag{9.1}$$

by a diagram such as that in Figure 9.9. The problem of how to deal with

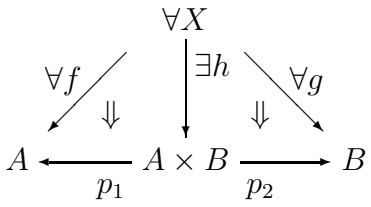


Figure 9.9: An example of diagram.

very large formulas is now a major topic in formal proof development. Our aim is to propose a solution of this problem by providing a good user interface based on commutative diagrams to proof assistants.

While trying to model our commutative diagram interface, we encountered a difficulty which seems to occur when modeling many other user interfaces, too. Our model of user interface have two attributes; items which are the object of computation and items used to communicate with the user. We call the former "computed objects" and the latter "communication objects." The user gives a communication object to the system which then translates it to a *corresponding* computed object and manipulates (or computes) it. Or alternatively, the system has a computed object as a result of some computation, translates it to a *corresponding* communication object, and then shows it to the user. In the case of our system, computed objects are logical formulas and communication objects are commutative diagrams. We have two "correspondences" here, and it would seem natural to search for *functions* to describe them. However, there are no such functions which are satisfactory. Instead, we have to live with *relations*, and the model must be more complicated. So, we propose what we call a *relational model*, and we describe it in Section 9.2.3.

We give a formulation of commutative diagrams in Section 9.2.1, in terms of directed graphs with cells. Next, we describe the commands of the system. The commands are classified into draw commands and proof commands. We list up all draw commands, and then show how they work by example in Section 9.2.2. We then discuss the general framework of the user interface, and propose our own based on relations in Section 9.2.3. In Section 9.2.4, we explain the proof commands, which are also classified into "pasting" commands, to prove equations, and "logical" commands for each logical inference rule, to prove logical formulas. We discuss related work in Section 9.2.5.

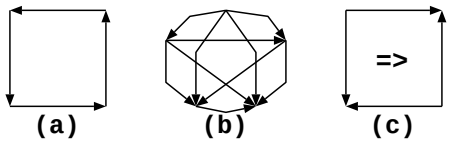


Figure 9.10: Non-examples of diagrams.

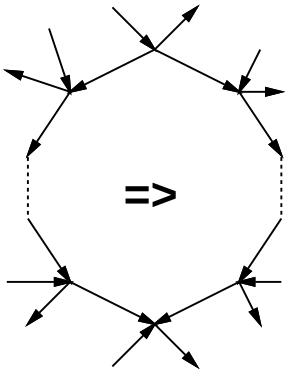


Figure 9.11: An example of commutative cell.

9.2.1 Commutative Diagrams

Figure 9.9 shows a commutative diagram. But what exactly are commutative diagrams? In order to treat them by machines, we need their formal definition. We define commutative diagrams to be directed acyclic planar graphs (i.e., planar DAG) with labels attached to some nodes and edges, together with specified *commutative cells*. A directed graph consists of nodes and edges with the assignment of the source and target node to each edge. By "acyclic," we mean that there are no cycles. So Figure 9.10(a), for example, is not a commutative diagram. "Planar" means the graph can be embedded in the plane. Figure 9.10(b) has no such embedding, so it is not a commutative diagram. Labels are attached to nodes and edges. Typical labels are variable names, possibly quantified by $\forall$ or $\exists$.

To define commutative cells (Figure 9.11), we need some auxiliary terminology. We define *cells* to be the minimal region surrounded by edges. Also, *paths* of a graph are defined as usual. We construct a *commutative cell* by placing a double arrow in a cell surrounded by just two paths. Figure 9.10(c) is not a commutative cell, because it is surrounded by four paths.

For example, Figure 9.9 is a diagram in this sense; it has two commutative cells.

command	description
$\cdot$	put a node
$\rightarrow$	draw a edge
$A, \forall X, \exists h$	labeling to a node or edge
$\Rightarrow$	draw a double arrow

Figure 9.12: Draw commands.

9.2.2 Draw Commands

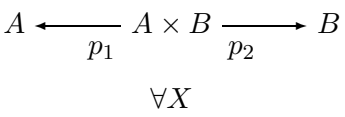
Figure 9.12 shows the list of draw commands. These commands are used to draw commutative diagrams. The commands are "to put a node," "to draw an edge from a node to another," "to put a label on a node or edge," and "to draw a double arrow in a cell, to make a commutative cell." Clearly, an arbitrary diagram can be drawn using a sequence of these commands.

Figure 9.13 shows how our example diagram from Figure 9.9 can be drawn. In (9.2), we put nodes "A," "$A \times B$," and "B," and draw edges "p_1" and "p_2." We then put another node and attach a label "$\forall X$" to it in (9.3). We then draw an edge from $\forall X$ to A, and attach a label "$\forall f$" to that edge in (9.4). Similarly we draw two edges "$\forall g$" in (9.5) and "$\exists h$" in (9.6). Since the left cell is surrounded by two paths, we put a double arrow in the cell, to make it a commutative cell in (9.7). The right cell is similarly made commutative in (9.8). Thus, we get the diagram as in Figure 9.9.

Before proceeding further, we give an intuitive meaning to the commutative diagrams. Suppose a node in the diagrams represents a set; so, X, A, B, and $A \times B$ are sets in our example. Also, suppose an edge represents a function from its source to its target, so, f is a function from set X to set A in Figure 9.9. A path is a composition of functions represented by the component edges. A commutative cell asserts the equality of functions represented by the surrounding paths. The diagram in Figure 9.9 says the composition of h and p_1 is equal to the function f. Meaning behind the diagram in Figure 9.9 thus is "for any set X, for any function f from X to A, for any function g from X to B, there exists a function h from X to $A \times B$ such that f is equal to the composition of h and p_1 and g is equal to the composition of h and p_2." The formula (9.1) is equivalent to this.

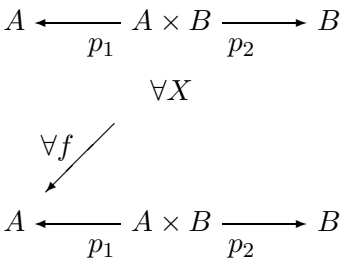
The draw commands create not only diagrams but also incrementally create formulas with a specified position, as depicted by the cursor $\triangle$ in Figure 9.13. We explain with the last drawing example how a formula in each step is generated in Figure 9.13.

(9.2): the diagram contains no predicate, so the corresponding formula

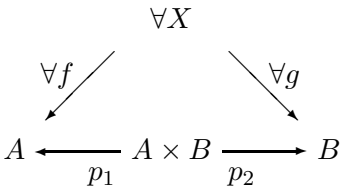


$$\triangle\ \top \tag{9.2}$$

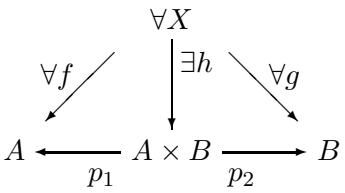
$$(\forall X)\mathrm{Obj}(X) \Rightarrow\ \triangle\ \top \tag{9.3}$$



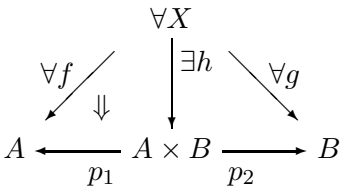
$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow\ \triangle\ \top\end{array} \tag{9.4}$$



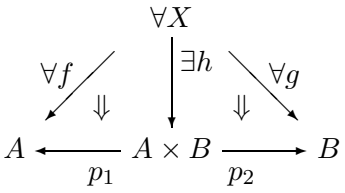
$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow\ \triangle\ \top\end{array} \tag{9.5}$$



$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow \\ \quad\quad\quad (\exists h)(\mathrm{source}(h) = X \wedge \mathrm{target}(h) = A \times B) \wedge\ \triangle\ \top\end{array} \tag{9.6}$$



$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow \\ \quad\quad\quad (\exists h)(\mathrm{source}(h) = X \wedge \mathrm{target}(h) = A \times B) \wedge \\ \quad\quad\quad\quad (f = h\,; p_1) \wedge\ \triangle\ \top\end{array} \tag{9.7}$$



$$\begin{array}{l}(\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow \\ \quad\quad\quad (\exists h)(\mathrm{source}(h) = X \wedge \mathrm{target}(h) = A \times B) \wedge \\ \quad\quad\quad\quad (f = h\,; p_1) \wedge (g = h\,; p_2) \wedge\ \triangle\ \top\end{array} \tag{9.8}$$

Figure 9.13: An example of drawing.

is the truth $\top$ with the specified position left to itself. (9.3): "$(\forall X)$" is first inserted to the left of the cursor position, resulting the formula $(\forall X) \mathrel{\triangle} \top$, and then "$\mathrm{Obj}(X) \Rightarrow$" is inserted to the left of the cursor position. (9.4): likewise, "$(\forall f)$" is first inserted to the left of the cursor, then "$(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow$" is inserted. We put the precedence of the quantifier to be the weakest, so this formula is parsed as $(\forall X)\Big(\mathrm{Obj}(X) \Rightarrow (\forall f)(((\mathrm{source}(f) = X) \wedge (\mathrm{target}(f) = X)) \Rightarrow \top)\Big)$. The creation of formulas goes on in a similar fashion. Note that h is quantified by an existential quantifier, so the logical connective used is the conjunction, instead of the implication. The draw command for the double arrow creates equations such as $f = h; p_1$ ($h; p_1$ denoting the composition of h and p_1), and we finally get the requisite formula.

9.2.3 Relational Model of User Interface

Our system displays a logical formula as a character sequence in a traditional way, or as a commutative diagram, as explained earlier. Now, given a commutative diagram typically displayed by the system, it is natural to expect that there is a unique formula represented, so that the user can determine which formula the diagram is showing. In other words, we can determine a function from diagrams to formulas. In terms of implementation, we can also reasonably expect a function from formulas to diagrams, a function which guides the implementation of the display algorithm.

This leads to what we call a *functional model of user interface*, as shown in Figure 9.14. We not only require the existence of functions f and g back and forth between diagrams and formulas, but also the canonical equivalence relations R between diagrams and S between formulas, as well as the following properties for all formulas ϕ and diagrams δ:

$$f(g(\delta))\ R\ \delta, \qquad \text{and} \qquad g(f(\phi))\ S\ \phi.$$

Here, g followed by f maps a diagram to an equivalent diagram, and f followed by g maps a formula to an equivalent formula. Also, we request a function p from draw command sequences to diagrams.

However, there is no successful definition of such functions! They can be very well defined, but the result (by our definition, at least) has been always unsatisfactory. There are clearly no reasonable functions from commutative diagrams to formulas (Figure 9.17), but there are functions from formulas to diagrams. However, all the functions that we could define map some formulas to a very cumbersome diagrams. But such over-complicated diagrams are

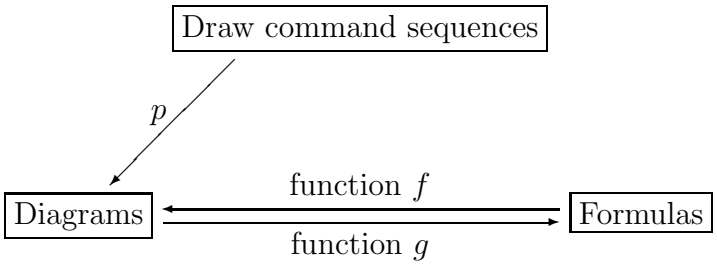


Figure 9.14: A functional model of user interface.

pointless; diagrams are introduced in order to help, not hinder, understanding (Figure 9.16).

This shows that the functional model does not work for our case. Instead, we propose a *relational model of user interface*, shown in Figure 9.15. Here, there is a relation, rather than two functions, between formulas and diagrams. In addition to function p, we also have a function q from draw command sequences to formulas. In the functional model, q was available as the composition of p followed by g, but in the relational model, we need to give q explicitly. We also need canonical equivalences R between diagrams and S between formulas, as in the functional model. For all draw command sequences σ and σ', we postulate the following property:

$$q(\sigma)\ S\ q(\sigma') \Rightarrow p(\sigma)\ R\ p(\sigma'). \tag{9.9}$$

There are only finitely many command sequences mapped to a given commutative diagram. Also, given a logical formula, there are only finitely many command sequences mapped to it. Moreover, those formulas are equivalent in many cases. For example, by changing $(\forall f)$ and $(\forall g)$ draw commands, the resulting formula is equivalent. So, we conjecture that in fact there is a limited, tractable number of logical formulas corresponding to a given commutative diagram.

Since a similar difficulty in finding functions arises in the design of other user interface [95], we believe our relational model is worth generalizing to some class of user interfaces. Interestingly, property (9.9) recalls us logical relations, the central notion in the theory of parametricity of programming languages, which might support the use of our relational model.

9.2.4 Proof Commands

There are two types of proof commands: pasting commands for proving equations and logical inference commands for proving compound formulas.

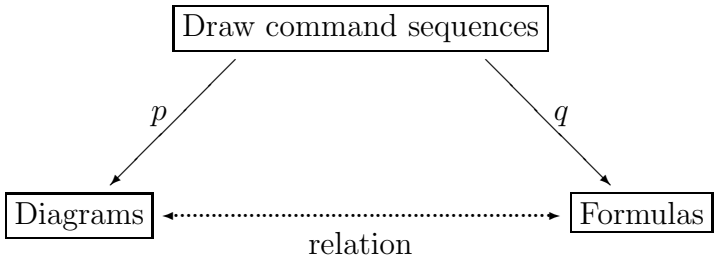


Figure 9.15: A relational model of user interface.

$$A \xleftarrow[p_1]{} A \times B \xrightarrow[p_2]{} B \qquad \triangle\ \top \tag{9.10}$$

$$\vdots$$

$$\begin{array}{c} \forall X \\ \forall f \swarrow \\ A \xleftarrow[p_1]{} A \times B \xrightarrow[p_2]{} B \end{array} \qquad \begin{array}{l} (\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \ \triangle\ \top \end{array} \tag{9.11}$$

$$\begin{array}{c} \forall X \qquad\quad X \\ \forall f \swarrow \\ A \xleftarrow[p_1]{} A \times B \xrightarrow[p_2]{} B \end{array} \qquad \begin{array}{l} (\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \ \triangle\ \top \end{array} \tag{9.12}$$

$$\vdots$$

$$\begin{array}{c} \forall X \qquad\quad X \\ \forall f \swarrow \ \Downarrow \ \searrow \exists h \quad h \swarrow \ \Downarrow \ \searrow \forall g \\ A \xleftarrow[p_1]{} A \times B \xrightarrow[p_2]{} B \end{array} \qquad \begin{array}{l} (\forall X)\mathrm{Obj}(X) \Rightarrow \\ \quad (\forall f)(\mathrm{source}(f) = X \wedge \mathrm{target}(f) = A) \Rightarrow \\ \quad\quad (\forall g)(\mathrm{source}(g) = X \wedge \mathrm{target}(g) = B) \Rightarrow \\ \quad\quad\quad (\exists h)(\mathrm{source}(h) = X \wedge \mathrm{target}(h) = A \times B) \wedge \\ \quad\quad\quad\quad (f = h\,; p_1) \wedge (g = h\,; p_2) \wedge \ \triangle\ \top \end{array} \tag{9.13}$$

Figure 9.16: A formula does not determine diagram.

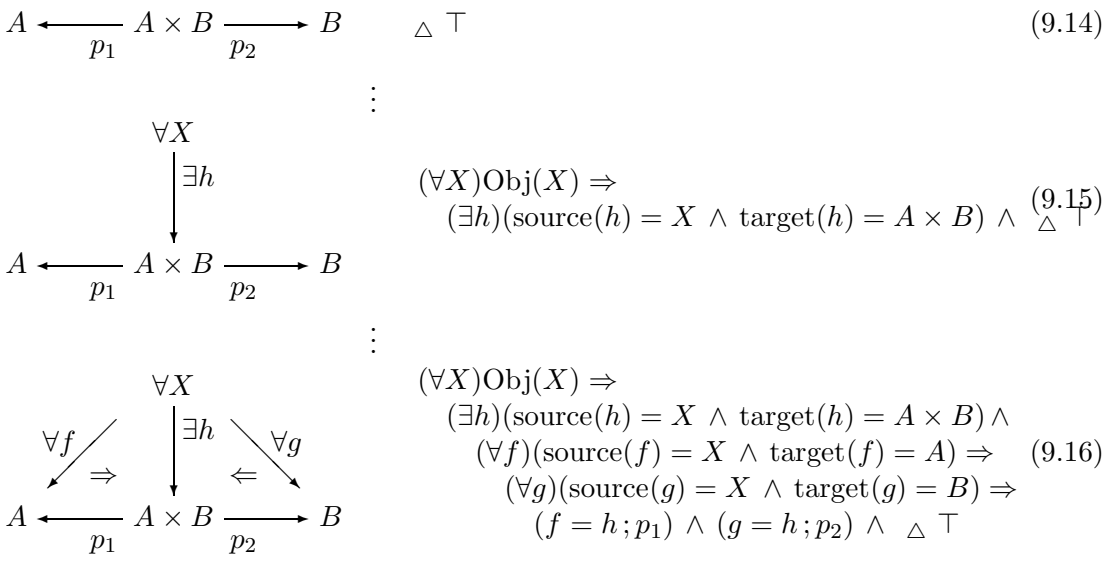


Figure 9.17: A diagram does not determine formula.

Pasting Command—proving equations

There are commands to prove equations between paths of diagrams by *pasting* the cells. As an example, assume we wish to prove $f = g$ from four assumptions $f = (\mathrm{id}\,;f)$, $\mathrm{id} = (g\,;h)$, $(h\,;f) = \mathrm{id}$ and $(g\,;\mathrm{id}) = g$. An informal proof would look like the following.

> Assumptions: $f = \mathrm{id}\,;f$, $\mathrm{id} = g\,;h$, $h\,;f = \mathrm{id}$, $g\,;\mathrm{id} = g$.
> Goal: $f = g$.
> Proof: $f = \mathrm{id}\,;f = g\,;h\,;f = g\,;\mathrm{id} = g$.

We show how to prove this using pasting commands. The proof goes on in a "goal-oriented" manner. There is always a goal formula (usually only one) and some assumptions. Our assumptions and goals are shown in Figure 9.18. The goal diagrams are pasted using some assumptions. In Figure 9.19, we explain how the pasting command works by example. We paste the diagram (1) in the assumption to the goal diagram. Edge f matches, so we paste it along edge f. Pasted cells are colored; uncolored cells become the new goal. This means that we can now prove $id; f = g$ instead of $f = g$. Similarly, the diagrams (2) and (3) are pasted. Finally, the goal cell is the same as in diagram (4) in the assumption. In this case, we can paste the whole diagram. The goal diagram is now completely colored, so it has been proved. The pasting command is used to prove the diagram by pasting.

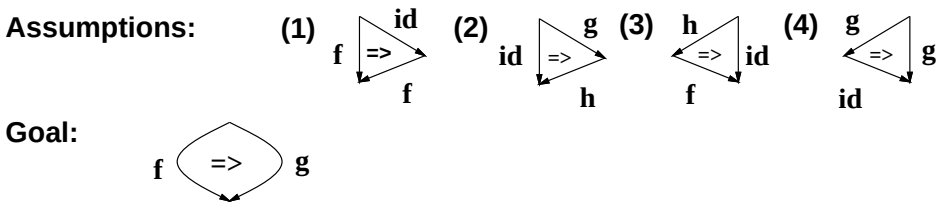


Figure 9.18: A representation using diagrams.

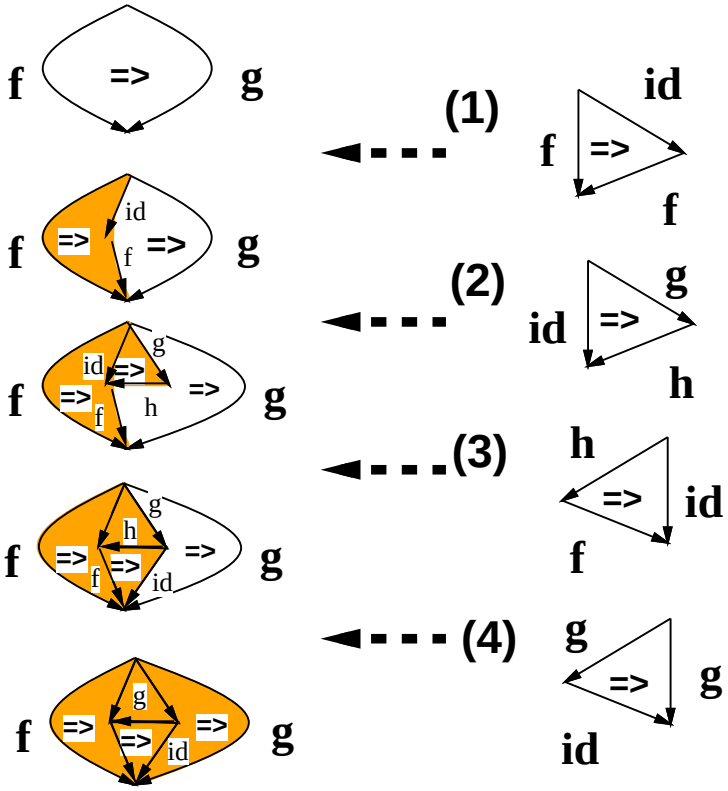


Figure 9.19: An example of pasting.

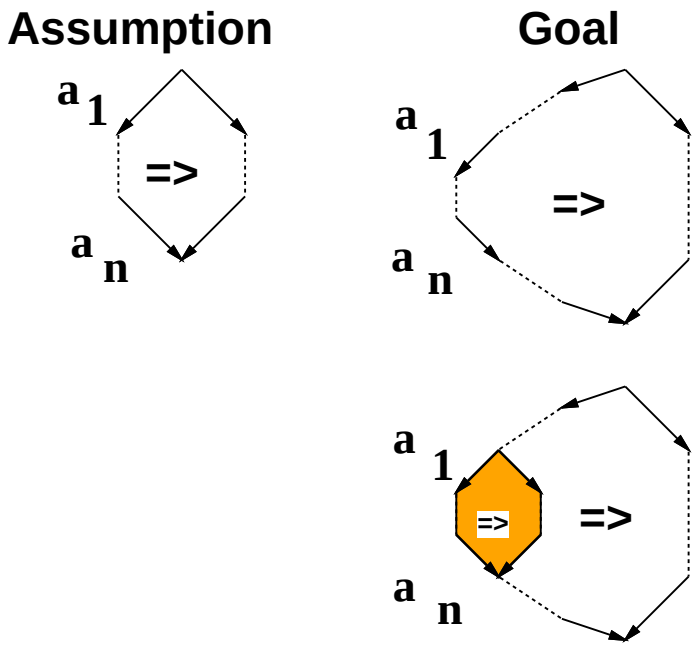


Figure 9.20: Condition of pasting.

We now describe the exact condition under which the pasting commands can be applied. If the assumption diagram exactly matches the goal diagram, the whole diagram is then pasted. In general, when the whole left path of the assumption diagram matches a part of the left path of the goal diagram, we can paste. In this case, the assumption diagram is pasted in the goal diagram like in Figure 9.20. The pasted part is colored, and the uncolored part becomes a new goal commutative cell which is surrounded by two paths.

As in Figure 9.21, if a part of left path of diagram matches the goal, we can not paste it. Our example proof is written within the realm of proof trees shown in Figure 9.22. It is clear that there are many steps and that it is more cumbersome than the proof by pasting!

Proof Commands for Compound Formulas

So far, we dealt only with equations, but we also treat compound formulas as appears in the system LK introduced by Gentzen [31]. We make a usual simplification to LK: we consider the hypotheses of sequents to be a *multiset* rather than a sequence of formulas.

As we showed in the previous section, our pasting command supports a forward reasoning of equations. In contrast, our proof command supports

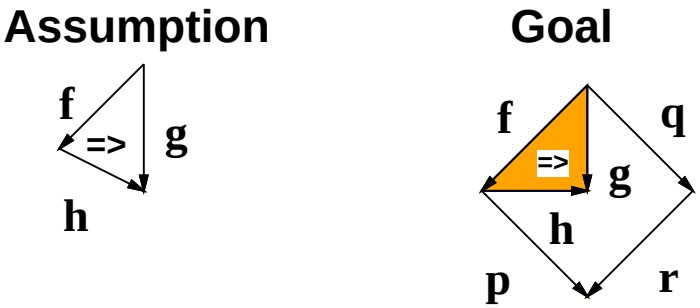


Figure 9.21: A non-example of pasting.

$$\dfrac{\dfrac{\dfrac{\mathrm{id} = g\,;h \qquad \dfrac{h\,;f = \mathrm{id}}{g\,;h\,;f = g\,;\mathrm{id}} \quad g\,;\mathrm{id} = g}{\mathrm{id}\,;f = g\,;h\,;f \qquad\qquad g\,;h\,;f = g}}{f = \mathrm{id}\,;f \qquad\qquad \mathrm{id}\,;f = g}}{f = g}$$

Figure 9.22: Proof tree.

backward reasoning in the level of predicate calculus. In a proof session, the system maintains a multiset of goal sequents, which are displayed as windows called *tasks*. A task window has typically three subwindows: the *Hypo* and *Consequence* windows containing the multiset of formulas in the hypothesis and consequence part, respectively, of the sequent, and the *Data* window containing the set of variables, which should be made into eigenvariables of the proof tree of the current goal (Figure 9.23). So, task windows are displayed during the proof session, but the system attaches a sequent to each task window; that sequent is called the *underlying sequent* of the task window. Such seemingly redundant book keeping is in fact not redundant because of the argument we gave in Section 9.2.3.

A proof session proceeds by applying a logical rule (backwards) to the current goal. For instance, if $\Gamma \vdash \Delta, \mathcal{A} \wedge \mathcal{B}$ is the current goal, one may apply the Right-$\wedge$ rule to split the current goal to be $\Gamma \vdash \Delta, \mathcal{A}$ and $\Gamma \vdash \Delta, \mathcal{B}$. If the current goal is in the form of $\mathcal{A} \vdash \mathcal{A}$, it can be resolved by applying the Axiom rule. One repeats such backward inferences until all goal is resolved.

Now let the current task T be as follows: its Data set and Hypo multiset

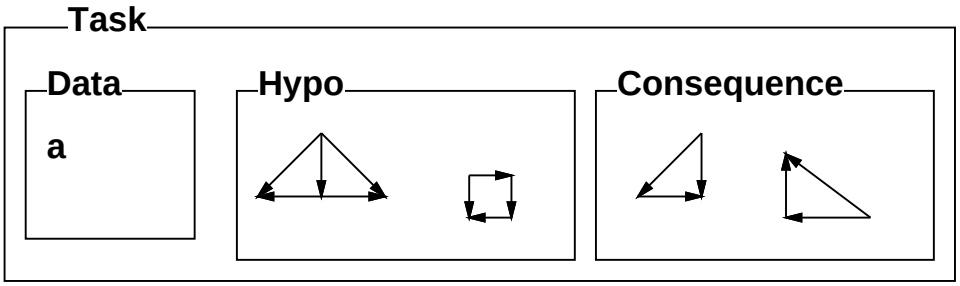


Figure 9.23: An example of Task window.

are empty, its Consequence is

$$\begin{array}{ccccc} & & X & & \\ & f \swarrow \Downarrow & \downarrow h & \Downarrow \searrow g & \\ A & \xleftarrow[p_1]{} & A \times B & \xrightarrow[p_2]{} & B \end{array}$$

and the underlying sequent is $\vdash (f = h\,;p_1) \wedge (g = h\,;p_2)$. An application of Right-$\wedge$ command with respect to the unique conjunction formula of the underlying sequent splits T into two tasks T_1 and T_2 which are described as follows. The Data and Hypo of T_1 and T_2 are the same as those of T, but the Consequence of T_1 is

$$\begin{array}{ccc} & X & \\ f \swarrow \Downarrow & \downarrow h & \\ A & \xleftarrow[p_1]{} & A \times B \end{array}$$

and its underlying sequent is $\vdash f = h\,;p_1$, while the Consequence of T_2 looks like

$$\begin{array}{ccc} X & & \\ h \downarrow & \Downarrow \searrow g & \\ A \times B & \xrightarrow[p_2]{} & B \end{array}$$

and its underlying sequent is $\vdash g = h\,;p_2$.

In this way, the execution of proof command changes not only the underlying sequent of the task, but also the diagram being displayed.

9.2.5 Discussion

We introduced the functional specification of our interface using a commutative diagram for proving predicates in category theory. In the design process, we encountered the notion of relational models for user interfaces, making this worth of further investigation.

Freyd and Scedrov defined diagrammatic representation for categorical property [29]. In his representation, a sequence of diagrams represents a unique formula. Our system only use one diagram for formula, so one diagram don't decide one formula. The Freyd-Scedrov diagrams are bigger than ours, but the meaning is clearer. Freyd-Scedrov diagrams have an explicit order with which the quantifiers should appear in the logical formula, but the order is ambiguous in our diagrams. We think the merits of our small representation is more important than the demerits of the ambiguities.

Shin gave a formal system for Venn diagrams, VENN, whose primitive objects are diagrammatic, not linguistic [85]. He also gave a semantic analysis for VENN. Cooper argued for a particular treatment of generalized quantifiers [21]. Barwise and Etchemendy developed a visual logical system Tarski's World which treats first-order logic [5, 4]. Yamamoto *et al.* showed the formalization of planar graphs which seems to be applicable to our system [99]. Kinoshita and I gave a theoretical background of pasting commands [55]. They used an algebraic structure over groupoids arising from the pasting style proofs.

Chapter 10

Conclusion

In this thesis, we investigated the following three subjects:

- Abstract model checking for formal verification.
- Synthesis by model checking and abstraction.
- Verification of the entire process of abstract model checking.

In this chapter, we conclude the thesis by briefly summarizing the results of the previous chapters along with the above three subjects, and then stating the future work.

10.1 Summary

The first subject of the thesis was on abstract model checking for formal verification. We verified the correctness of the algorithms for concurrent garbage collection by abstract model checking in Chapter 3. The validity of abstract model checking was also formulated with a technique in program semantics augmented with the new notion "cumulative" in Chapter 4. We proposed an abstraction method of link structures by using regular expressions in Chapter 5. Using the results of Chapter 4 and those of Chapter 5, we could solve the problems that were raised in Chapter 3.

The second subject was on synthesis by model checking and abstraction. We applied model checking to the discovery of algorithms for concurrent GC and mutual exclusion in Chapter 6. In Chapter 7, we improved the method in Chapter 6 by using BDDs. We also applied approximation of BDD to reduce the search space of algorithms.

The third subject was on verification of the entire process of abstract model checking. We formalized and proved the abstraction in Chapter 3

using HOL in Chapter 8. The safety of the concurrent GC algorithm was thus completely verified by the validity of abstract model checking shown in Chapter 4 and the formal proof constructed in this chapter. As for formal proof construction, we proposed two user interfaces of theorem provers in Chapter 9. The proof in Chapter 8 was actually constructed under an interface developed in Chapter 9.

10.2 Future Work

In this section, we discuss the problems that should be solved in the future work, including those already discussed in the previous chapters.

Concerning the validity of abstract model checking, only the safety property was analyzed in Chapter 4. There are many other important temporal properties such as liveness. In order to apply abstract model checking to such properties, it is necessary to find conditions that guarantee the validity of abstract model checking for those properties. We have a conjecture concerning ACTL* formulas. Model checking of ACTL* formulas is essentially achieved by a depth first search, which is similar to the model checking algorithm for the safety. Therefore, the conditions we found in Chapter 4 is expected to guarantee the validity of model checking for all ACTL* formulas. For this purpose, we have to extend our analysis to ACTL* formulas.

We had to formally prove that the abstract transitions defined in Chapter 3 satisfy the conditions for the validity given in Chapter 4 in order to complete the verification of the concurrent GC algorithm. As mentioned in Chapter 8, the construction of the formal proof took long time, though it is almost straightforward. Therefore, we hope that it will be possible to automatically construct abstract transitions that are guaranteed to satisfy the conditions for the validity.

Automatic construction of abstract transitions would have the advantage that a system is automatically verified by abstract model checking once the user gives an abstraction relation. This should be a big progress, because in many cases an appropriate abstraction relation for verification is unknown in advance, and the user has to try many abstraction relations before finding an appropriate one.

We proposed an abstraction method of link structures in Chapter 5. To define an abstraction relation, the user should merely select several regular expressions in our method. It is therefore very easy to try many abstraction relations. However, abstract transitions are not constructed automatically. In Chapter 5, we gave only a guideline for constructing abstract transitions so that they satisfy the conditions for the validity. Automatic construction

of abstract transitions for our method should be developed in future work.

We used regular expressions for abstraction of link structures. Can other classes of expressions be used? That is also future work.

Needless to say, abstraction of link structures by using regular expressions is not almighty. Some shapes of link structures cannot be judged by regular expressions. For example, we cannot judge whether a cell in either a binary tree or a list. The shape analysis [81] can judge such properties, and abstraction of link structures using the shape analysis is expected to work for some problems.

Synthesis of algorithms by model checking described in this thesis is only a first step towards automatic discovery of algorithms. Much improvement is required in the future work. In our case studies, we obtained several algorithms that satisfy the given specifications. However, many of them have turned to be equivalent to the original algorithm. In order to increase the efficiency of search, we should apply an equivalence checker to obtained algorithms, or reduce candidates by equivalence in advance.

In order to find algorithm that are completely new, the candidate space must be very big, and model checking should be efficient. If the abstract domain is smaller, the time took by abstract model checking is shorter. However, abstract model checking tends to fail even if the original concrete system satisfies the specification, because we required $\mathcal{M}' \models \phi \Rightarrow \mathcal{M} \models \phi$ for the abstract model $\mathcal{M}'$ of $\mathcal{M}$. In such a situation, we cannot find a new algorithm even if the candidate space contains that one. Therefore, we should require a dual condition $\mathcal{M}' \not\models \phi \Rightarrow \mathcal{M} \not\models \phi$ to use abstract model checking for discovery. This condition is rewritten to $\mathcal{M} \models \phi \Rightarrow \mathcal{M}' \models \phi$. We have to investigate this dual abstraction with respect to the safety or other properties.

Bibliography

[1] M. D. Aagaard, R. B. Jones, and C-J. H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340. Springer Verlag, September 1999.

[2] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proceedings of the 25th Annual Symposium on Theory of Computing*, pages 592–601. ACM Press, 1993.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Gim'enez, H. Herbelin, G. Huet, C. noz, C. Murthy, C. Parent, C. Paulin, A. Saibi, and B. Werner. The Coq proof assistant reference manual – version v, 1997.

[4] J. Barwise and J. Etchemendy. *The Language of First-Order Logic (including the Macintosh program, Tarski's World).* Number 23 in CSLI Lecture Notes. CSLI, 1990.

[5] J. Barwise and J. Etchemendy. Hyperproof: Logical reasoning with diagrams. In *AAAI Spring Sympo. Series, Reasoning with Diagrammatic Representations*, pages 80–84, 1992.

[6] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[7] Y. Bertot. Direct manipulation of algebraic formulae in interactive proof. In *User-Interfaces for Theorem Provers, UITP'97*, pages 17–24, 1997.

[8] Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 141–160. Springer-Verlag, 1994.

[9] Y. Bertot, T. Kleymann-Schreiber, and D. Sequeira. Implementing proof by pointing without a structure editor. Research Report RR-3286, INRIA, 1997.

[10] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Universite de Liege, 1998.

[11] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proceedings of Formal Methods Europe*, volume 1501 of *Lecture Notes in Computer Science*, pages 456–478. Springer-Verlag, 1996.

[12] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs (extended abstract). In *Static Analysis, 4th International Symposium, SAS '97*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 1997.

[13] G. Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.

[14] R. E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[15] C. Chou and D. Peled. Formal verification of a partial-order reduction technique for model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 241–257, 1996.

[16] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic, 1981.

[17] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[18] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[19] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):77–104, 1996.

[20] E. M. Clarke, S. Jha, Y. Lu, and D. Wang. Abstact BDDs: A technique for using abstraction in model checking. In *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 172–186, 1999.

[21] R. Cooper. Generalized quantifiers and resource situations. In *Situation Theory and its Applications, Vol. 3*, number 37 in CSLI Lecture Notes, pages 191–211, 1993.

[22] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[23] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, July 1996.

[24] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2), 1997.

[25] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[26] E. A. Emerson and E. M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[27] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[28] F. A. Emerson and A. P. Sistla. Symmetry and model checking. *Formal Methods in System Design: An International Journal*, 9(1/2):105–131, 1996.

[29] P. Freyd and A. Scedrov. *Categories, Allegories*. North-Holland, 1990.

[30] N. Fujinami. Superoptimization of scheduling machine instruction pipelines. *Computer Software*, 16(2):80–84, 1999. in japanese.

[31] G. Gentzen. Untersuchungen über das logische schliessen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1934–5.

[32] P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag Inc., 1996.

[33] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In *Computer Aided Verification*, pages 438–449, 1993.

[34] G. Gonthier. Verifying the safety of a practical concurrent garbage collector. In *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 462–465. Springer-Verlag, 1996.

[35] M. Gordon and T. Melham. *Introduction to HOL: A Theorem-proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[36] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. In *PLDI'92, Proceedings of the conference on Programming language design and implementation*, pages 341–352, 1992.

[37] T. Griffin. An environment for formal systems. Technical Report 87-846, Department of Computer Science, Cornell University, 1987.

[38] M. Hagiya and H. Kakuno. Proving as editing. In *User-Interfaces for Theorem Provers, UITP'96*, pages 35–42, 1996.

[39] M. Hagiya and K. Takahashi. A verification of concurrent garbage collection by abstract model checking. In *Second Workshop on Systems for Programming and Applications*, 1999. `http://www.osss.is.tsukuba.ac.jp/spa99proc`. in japanese.

[40] M. Hagiya and K. Takahashi. Discovery and deduction. In *Discovery Science, Third International Conference, DS 2000*, volume 1967 of *Lecture Notes in Artificial Intelligence*, pages 17–37. Springer-Verlag, 2000.

[41] J. Harrison. A Mizar mode for HOL. In *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220. Springer-Verlag, 1996.

[42] K. Havelund. Mechanical verification of a garbage collector. In *Fourth International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMTPTA '99)*, pages 1258–1283, 1999.

[43] K. Havelund and N. Shankar. A mechanized refinement proof for a garbage collector. *Formal Aspects of Computing*, 9(3), 1997.

[44] P. Hitchcock and D. M. R. Park. Induction rules and termination proofs. In *Automata, Languages, and Programming*, pages 225–251. North-Holland/American Elsevier, 1972.

[45] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1990.

[46] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[47] T. Hune, J. Romijn, M. Stoelinga, and F. Vaandrager. Linear parametric model checking of timed automata, 2001.

[48] M. R. A. Huth and M. D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[49] P. B. Jackson. Verifying a garbage collection algorithm. In *Theorem Proving in High Order Logics*, volume 1479 of *Lecture Notes in Computer Science*, pages 225–244. Springer-Verlag, 1998.

[50] J. J. Joyce and C. H. Seger. Linking BDD-based symbolic evaluation to interactive theorem proving. In *Proceedings of the 30th Design Automation Conference*, pages 469–474. Association for Computing Machinery, 1993.

[51] S. Kalvala, M. Archer, and K. Levitt. Implementation and use of annotations in HOL. In *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 International Workshop, Leuven, September 1992*, IFIP Transactions A-20, pages 407–426. North-Holland, 1993.

[52] M. Kaufmann and J. S. Moore. ACL2: An industrial strength version of nqthm. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 23, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[53] Y. Kinoshita and J. Power. A general completeness result in refinement. In *Proceedings of the 14th International Workshop on Algebraic Development Techniques*, number 1827 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[54] Y. Kinoshita and K. Takahashi. Cumulatives for safety. *Electronic Notes on Theoretical computer Science*.

[55] Y. Kinoshita and K. Takahashi. Gropoid of equational proofs. *Bulletin of the Electrotechnical Laboratory*, 60(11):711–718, 1996.

[56] Y. Kinoshita and K. Takahashi. Proving through commutative diagrams. In Lawrence S. Moss, Jonathan Ginzburg, and Maarten de Rijke, editors, *Logic, Language and Computation*, volume 2, pages 128–142. CSLI publication, 1999.

[57] R. P. Kurshan. Analysis of discrete event coordination. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430. Springer-Verlag, 1989.

[58] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design: An International Journal*, 6(1):11–44, January 1995.

[59] D. E. Long. bdd - a binary decision diagram (bdd) package, 1993. `http://www.cs.cmu.edu/~modelcheck/code.html`.

[60] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[61] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[62] P. Manolios. *Computer-Aided Reasoning: ACL2 Case Study*, chapter 7, pages 93–111. Kluwer Academic Publishers, May 2000.

[63] P. Manolios, K. S. Namjoshi, and R. Summers. Linking theorem proving and model-checking with well-founded bisimulation. In *Computer Aided Verification*, pages 369–379, 1999.

[64] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS II*, pages 122–126, 1987.

[65] N. Merriam and M. Harrison. What is wrong with GUIs for theorem provers. In *User-Interfaces for Theorem Provers, UITP'97*, pages 67–74, 1997.

[66] The mizar project. `http://www.cs.ualberta.ca/~piotr/Mizar/`.

[67] T. Mizukami. Report for exercise III in information science, 2000. in japanese.

[68] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 1–16, 1995.

[69] T. Nipkow and L. Paulson. Isabelle/HOL. the tutorial. `http://www4.informatik.tu-muenchen.de/~nipkow/pubs/tutorial.pdf`, 1998.

[70] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system, 1992.

[71] L. C. Paulson. *Isabelle, A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[72] D. Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, volume 818, pages 377–390. Springer-Verlag, 1994.

[73] A. Perrig and D. Song. A first step on automatic protocol generation of security protocols. In *Proceedings of Network and Distributed System Security*, 2000.

[74] A. Pnueli. A temporal logic of programs. *Theoretical Computer Science*, 13:45–60, 1981.

[75] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 179–190, Austin, Texas, 1989.

[76] O. Pons. Undoing and managing a proof. In *User-Interfaces for Theorem Provers, UITP'97*, pages 85–92, 1997.

[77] O. Pons, Y. Bertot, and L. Rideau. Notions of dependency in proof assistants. In *User-Interfaces for Theorem Provers UITP'98*, pages 130–138, 1998.

[78] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proceedings of the 7th International Conference On Computer Aided Verification*, volume 939, pages 84–97, Liege, Belgium, 1995. Springer Verlag.

[79] A. W. Roscoe. *The Theory and Practive of Concurrency.* Prentice Hall, 1997.

[80] D. M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[81] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL'96: The 23th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, 1996.

[82] D. A. Schmidt. Data-flow analysis is model checking of abstract interpretations. In *POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[83] D. A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. In *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer-Verlag, 1998.

[84] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *International Conference on Concurrency Theory*, pages 1–16, 2000.

[85] S. Shin. A situation-theoretic account of valid reasoning with Venn diagrams. In *Situation Theory and its Applications, Vol. 2*, number 26 in CSLI Lecture Notes, pages 581–605, 1991.

[86] K. Slind. *HOL98 User's Manual Athabasca Release Preliminary Draft*, 1998. `http://www.cl.cam.ac.uk/Research/HVG/HOL/HOL.html`.

[87] D. X. Song. Athena: a new efficient automatic checker for security protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 192–202, 1999.

[88] C. Sprenger. A verified model checker for the modal μ-calculus in Coq. In Bernhard Steffen, editor, *Tools and algorithms for the construction and analysis of systems: 4th International Conference, TACAS'98*, pages 167–183, 1998.

[89] D. Syme. A new interface for HOL - ideas, issues, and implementation. In *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 324–339. Springer-Verlag, 1995.

[90] K. Takahashi and M. Hagiya. Searching for mutual exclusion algorithms using BDDs. In *Progresses in Discovery Science*, Lecture Notes in Computer Science. to appear.

[91] K. Takahashi and M. Hagiya. Proving as edition HOL tactics. *Formal Aspects of Computing*, 11:343–357, 1999.

[92] K. Takahashi and M. Hagiya. Abstraction of link structures by regular expressions and abstract model checking of concurrent garbage collection. In *First Asian Workshop on Programming Languages and Systems*, 2000.

[93] K. Takahashi and M. Hagiya. Abstract model checking of concurrent garbage collection by regular expressions. *IPSJ Transactions on Programming, Information Processing Society of Japan*, 42(SIG2(PRO 9)):61–70, 2001. in japanese.

[94] Y. Takebe and M. Hagiya. A user interface for controlling term rewriting based on computing-as-editing paradigm. In *User-Interfaces for Theorem Provers, UITP'97*, pages 93–100, 1997.

[95] J. Tanaka, December 1995. Personal communication.

[96] L. Théry. A proof development system for the HOL theorem prover. In *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop*, volume 780 of *Lecture Notes in Computer Science*, pages 115–128. Springer-Verlag, 1994.

[97] G. van Rossum. Python tutorial, 1998. `http://www.python.org/doc/tut/tut.html`.

[98] K. N. Verma. Reflecting symbolic model checking in Coq. In *Mémoire de DEA DEA Programmation*, September 2000.

[99] M. Yamamoto, S. Nishizaki, M. Hagiya, and Y. Toda. Formalization of planar graphs. In *8th International Workshop on Higher-Order Logic Theorem Proving System and Its Applications*, volume 971, pages 369–384, 1995.

[100] S. Yu and Z. Luo. Implementing a model checker for LEGO. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997)*, volume 1313, pages 442–458. Springer-Verlag, 1997.

[101] T. Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, 1990.

[102] T. Yuasa. Algorithms for realtime garbage collection. *IPSJ Magazine, Information Processing Society of Japan*, 35(11):1006–1013, 1994. in japanese.