

Static Byte-Code Analysis for State Space Reduction

Gustavo Quirós Araya
Matr.-Nr. 253320

Master Thesis

Submitted to the Faculty of Mathematics,
Computer Science and Natural Sciences
of RWTH University, Aachen, Germany,
in March 2006

Lehrstuhl für Informatik II,
Prof. Dr. Joost-Pieter Katoen,
Prof. Em. Dr. Klaus Indermark

This work was done with partial financial support from the Ministry of Science and Technology (Ministerio de Ciencia y Tecnología, MICIT) and the National Council for Scientific and Technological Research (Consejo Nacional para Investigaciones Científicas y Tecnológicas, CONICIT) of Costa Rica.

I sincerely thank Prof. Dr. Joost-Pieter Katoen for supporting the subject of this thesis and for accepting the role of supervisor and first referee. I also thank Dr. Thomas Noll for offering to act as a second reader and referee. I thank Prof. Em. Dr. Klaus Indermark for his superb teaching and his kind support.

I am very grateful to Dr. Michael Weber for his initial suggestion of the subject for this thesis, his accurate guidance and his most sincere interest in the success of this work. I am also in debt with Stefan Schürmans for his invaluable help with the innermost details of the NIPS virtual machine. Also, I would like to thank Michael Rohrbach for providing some of the examples used in this thesis, as well as Volker Stolz for his assistance during the initial phase of this work.

I thank MICIT and CONICIT for providing financial support throughout the duration of my studies in Aachen.

I am ever so grateful to my family Arturo, María, Alicia and Roberto, as well as Doña Sonia, Don Edgar and Kekas, for their unconditional love and support in every possible way. ¡Gracias!

Most importantly, to my wife Catalina, for sharing with me, baring with me, believing in me and supporting me in every step of the way.

I certify that I have written the present work independently and that I have used none other than the stated resources and aids.

Düsseldorf, March 13, 2006.

Contents

Preface	vii
1 The NIPS Virtual Machine for State Space Generation	1
1.1 The virtual machine	1
1.2 The byte-code language	2
1.2.1 Byte-code instructions	4
1.2.2 Procedures	5
1.3 States	5
1.4 An example of NIPS	6
2 Static Analysis for State Space Reduction	9
2.1 State space reduction in NIPS	9
3 A Model of NIPS	13
3.1 Programs	13
3.2 Control-flow graphs	14
3.3 State-transition systems	16
4 Path Reduction	21
4.1 Path reduction for WHILE	22
4.1.1 A comparison of WHILE and NIPS	23
4.2 Path reduction for NIPS	24
4.2.1 Analysis of breaking points	24
4.2.2 Merging steps	28
4.2.3 Byte-code transformation	33
4.2.4 Cycles and path reduction	35
4.3 An example of path reduction	37
4.4 Relationship between path reduction and partial order reduction	37
5 Dead Variable Reduction	41
5.1 Variables in NIPS	42
5.2 Live variable analysis for NIPS	42
5.2.1 Local storage size analysis	43
5.2.2 Local live variable analysis	44
5.2.3 Data flow analysis of procedures	45
5.3 Dead variable reduction for NIPS	45
5.3.1 Death points	45

5.3.2	Dead assignments	46
5.3.3	Specification-visible variables	47
5.3.4	The code transformation	47
5.4	An example of dead variable reduction	49
6	Step Confluence Reduction	51
6.1	Equivalent steps	51
6.2	Step confluence analysis	52
6.2.1	Confluence points	53
6.2.2	Confluent instructions	53
6.2.3	Confluent steps	54
6.3	Step confluence transformation	56
7	Reduction of Sequential NIPS Models	57
7.1	Sequential NIPS models	57
7.2	Path reduction for sequential models	58
7.3	Dead variable reduction for sequential models	59
7.3.1	Global live variable analysis	59
7.3.2	Global dead variable reduction	60
7.4	Step confluence reduction and sequential models	61
8	Correctness of the Reductions	63
8.1	Equivalences	63
8.2	Correctness of path reduction	64
8.2.1	Single step reduction	64
8.2.2	Multiple step reduction	70
8.3	Correctness of dead variable reduction	71
8.4	Correctness of step confluence reduction	73
9	SARN: Static Analysis and Reduction for NIPS	77
9.1	The SARN class library	77
9.2	SARN commands	79
10	Experimental Results and Evaluation	83
10.1	Experimental procedure	83
10.2	Spin examples	84
10.3	An embedded program	88
10.4	LUNAR models	89
10.5	Summary	90
11	Conclusions	93
11.1	Related work	93
11.2	Implementation	94
11.3	Experimental results	95
11.4	Future work	95

List of Figures

1.1	A NIPS-based system.	2
1.2	The NIPS virtual machine.	3
1.3	An example of NIPS: original model in Promela.	6
1.4	An example of NIPS: generated NIPS byte-code.	7
1.5	An example of NIPS: generated state-transition system.	8
2.1	Application of state space reduction based on static byte-code analysis.	11
3.1	Example of a NIPS control-flow graph: the block of instructions.	16
3.2	Example of a NIPS control-flow graph.	17
4.1	Elementary paths in NIPS.	28
4.2	Simple detection of elementary paths.	30
4.3	Invisible steps and blocking of non-breaking steps.	32
4.4	Difference between invisible steps and removed steps.	34
4.5	An example of path reduction.	38
4.6	Comparison of path reduction and partial order reduction.	40
5.1	An example of dead variable reduction.	50
6.1	Equivalent steps in NIPS.	52
8.1	Scenario for path reduction	65
8.2	Proof of Lemma 8.1, commuting of non-breaking steps.	67
8.3	Stuttering equivalence between π and π'	69
10.1	Average percentages of generated states.	90

List of Tables

8.1	Equivalences and temporal logic languages preserved by the reductions. . .	63
10.1	Results of running the Spin examples (1).	85
10.2	Results of running the Spin examples (2).	86
10.3	Results of running a sequential model of an embedded program.	88
10.4	Results of running the LUNAR models.	89

Preface

NIPS is a virtual machine for the generation of state-transition systems of concurrent models, and is designed to serve as a foundation for the development of model checking applications. Like any explicit-state representation, the transition systems produced by NIPS suffer from the *state-explosion problem*, by which models have exponentially large state spaces. This work is dedicated to the reduction of the effect of the state-explosion problem in NIPS.

Explicit-state model checking works by obtaining the state-transition system of a model, and then testing whether the system satisfies a given specification or not, the specification stated in some temporal logic language. Two state-transition systems are said to be *equivalent under a given specification*, if the specification may not distinguish between the two systems (that is, both systems either satisfy or do not satisfy the specification). Furthermore, two systems are *equivalent under a given temporal logic* if they are equivalent under any specification written in the logic. Using this, state space reduction works by obtaining a *reduced* state-transition system from a model, which is equivalent to the original system under a given specification, or a given set of specifications. model checking may be applied to the reduced system instead of to the original one, thus obtaining the same result in a more efficient manner.

State space reduction may be applied to the full state-transition system, producing the reduced system from it. However, this approach is not practical because the peak memory requirements are not improved, and they are precisely the most important limitation of model checking applications. In order to present a practical and useful solution, the reduction must be done *on-the-fly*, producing the reduced system directly from the model without the need to construct the full state-transition system. For this, static analysis of the code is usually employed in order to gather information which will be used when computing the reduced system.

NIPS features a byte-code language for encoding concurrent models. High-level modeling languages may be compiled to this byte-code, and the interpretation of the byte-code by the machine results in the construction of the model's state space representation. This architecture allows us to implement on-the-fly reductions merely by transforming the byte-code to another whose state-transition system is a reduced version of the original, and employing information gathered from static analysis of the byte-code in the process. The main goal of this work is to implement state space reductions for NIPS in this way.

This document is structured as follows: Chapter 1 introduces the NIPS virtual machine and its mechanics; Chapter 2 presents an overview of static analysis for state space reduction in NIPS; Chapter 3 presents a formalization of NIPS, which constitutes the framework on which the subsequent chapters are built; Chapter 4 presents *path reduction*, which works by merging sequences of steps from the same process into one step;

Chapter 5 presents *dead variable reduction*, which works by eliminating differences in states from the valuations of dead variables; Chapter 6 presents *step confluence reduction*, which identifies equivalent program locations and unifies them into a single location; Chapter 7 presents a version of the reductions specially designed for sequential systems; Chapter 8 demonstrates the correctness of the reductions by determining the types of temporal logic properties which are preserved by them; Chapter 9 presents SARN, which is a tool set for static analysis and state space reduction for NIPS, and which implements the reductions presented in this work; Chapter 10 presents some empirical results of the reductions and a general evaluation of them; finally, Chapter 11 presents the concluding remarks of this work.

Chapter 1

The NIPS Virtual Machine for State Space Generation

This chapter introduces NIPS [WS05, Sch05], which is a virtual machine for the generation of state-transition systems of concurrent models developed at the Chair of Computer Science II of RWTH Aachen University by Michael Weber and Stefan Schürmans. It is designed to be a convenient and flexible platform for the development of model checking tools: the virtual machine encapsulates the task of generating the model's state space, and delegates process scheduling decisions to a host application. It is also designed to be *language-independent*: NIPS features a simple byte-code language which is used to encode the operations performed by the processes of the concurrent system. By providing a translator from a given high-level modeling language to this byte-code, it is possible to process models written in this high-level language with any tool that is designed to work with the machine, which would typically be in the form of a model checker or a simulator. The byte-code serves then as an abstraction layer between modeling languages and tools. This allows NIPS to provide a convenient and flexible way to implement and reuse high-level modeling languages in novel model checking applications.

Figure 1.1 depicts a NIPS-based system. A model written in some high-level language is compiled to the NIPS byte-code language. This byte-code is then loaded by the virtual machine for its interpretation. An application (model checker, simulator, etc.) interacts with the machine by requesting the generation of the successor states from a given machine state, and by attending callbacks from the machine when a successor state is produced. The application has then the option to apply the desired scheduling policy and manipulation of states (interactive simulation, exploration of the state-transition system with depth-first or breadth-first search, etc.).

1.1 The virtual machine

The NIPS virtual machine is both a stack-based and a register-based machine. It supports the execution of concurrent processes which communicate with each other through channels.

Figure 1.2 shows the composition of the state of the virtual machine. The machine holds a global state, which corresponds to a state of the model's state space. The global state contains a set of global variables, a set of processes and a set of channels. Global

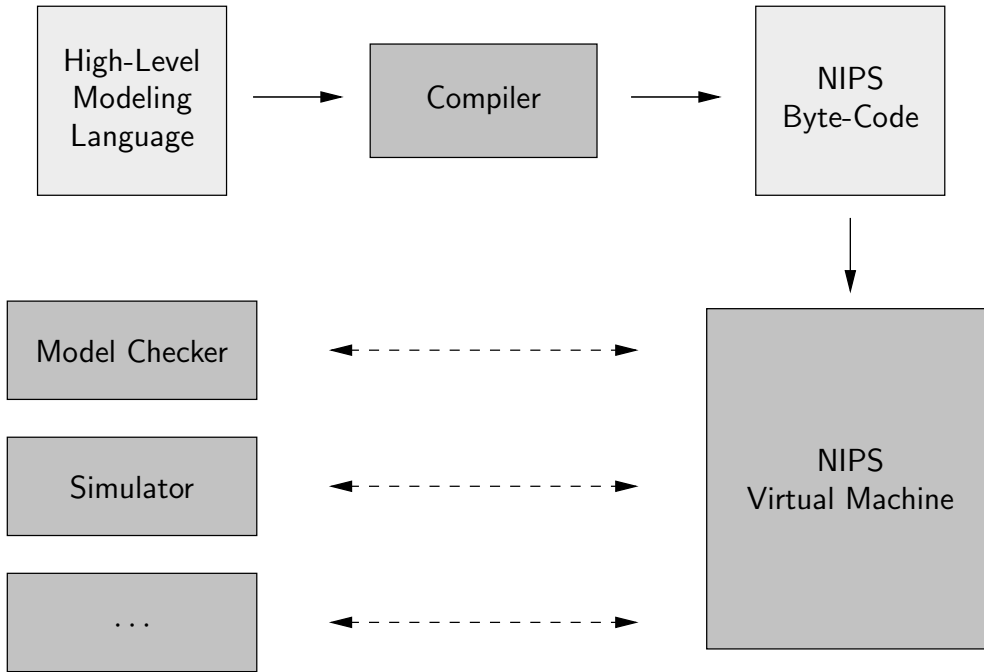


Figure 1.1: A NIPS-based system.

variables are identified by their offset in an array of memory locations, whereas processes and channels have corresponding unique numeric identifiers. Each process state contains a program counter which holds the position of the byte-code instruction that the process will execute next, as well as a set of process-local variables which are arranged for each process in the same manner as the global variables.

During execution, an active local state is created within the machine, which consists of an operand stack and a set of registers. Only one process may execute at a time, having exclusive use of the active local state. When the process pauses its execution in order to allow others to run, its active local state is lost. Therefore, this transfer of activity may only happen in selected points of the code, namely, in the conclusion of process steps, as we will see in the following section.

1.2 The byte-code language

The NIPS byte-code language is a simple low-level assembly-like language for the encoding of concurrent models. It resembles the byte-code languages of other stack-based virtual machines like the JavaTM virtual machine. However, because it is designed for the encoding of concurrent systems and the generation of state-transition systems, the language additionally has features which are common in modeling languages, such as the Promela language used in the Spin model checker [Hol97, Hol03]. Thus, it provides operations for the following tasks:

- *Dynamic creation of concurrent processes.* An executing process may spawn another process that will run concurrently with its creator.

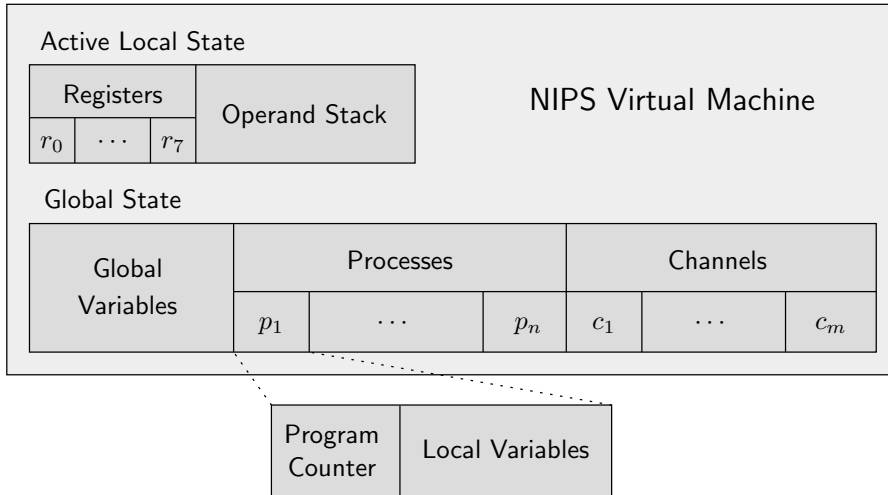


Figure 1.2: The NIPS virtual machine.

- *Dynamic creation of synchronous and asynchronous first-class communication channels.* A process may create a new communication channel at runtime. These channels may be synchronous -requiring the sending and receiving of messages to be done simultaneously-, or asynchronous -allowing sent messages to be stored in the channel for later reception-. Finally, channel identifiers may themselves be sent over channels, as in the π -calculus [Mil99].
- *Global and local variable operations.* Processes may read and write global variables as well as their own local variables.¹
- *Non-deterministic local control-flow.* In addition to the inherent non-determinism of concurrent execution, each process may present local non-deterministic execution by the use of non-deterministic jumps.
- *Guarded commands (speculative execution).* Processes may block their execution until a required condition is met. This is implemented by the speculative execution of code which may conditionally roll-back its effect.
- *Explicit control of state visibility.* Active processes execute without interruption until a step is completed, which signals that a visible global state has been obtained. From this state, any active process may resume its execution. Step completion is encoded directly in the byte-code, which allows for great control of step granularity from within the code.

A program for the NIPS virtual machine is stored in a byte-code file, which contains a single block of instructions. The execution of the machine begins at an initial state where there are no channels and only one active process. This process, known as the initial process, begins its execution with the first instruction in the byte-code. This process is responsible for the creation of the additional processes of the system, which

¹Additionally, a special process may *peek* at the local variables of other processes for verification purposes.

may in turn create new processes themselves. Also, processes may create communication channels for exchanging messages with other processes.

1.2.1 Byte-code instructions

The byte-code language features simple instructions which may be used to encode the more complex high-level constructs of modeling languages which feature concurrency, synchronous and asynchronous interprocess communication, shared memory and non-determinism. Some instructions are parametrized with modifiers, which specify some of the operation's characteristics, as well as arguments, which are literal values that appear in the code. Those instructions of NIPS which are most relevant to this work may be classified as follows:

- *Variable operations*: Loading and storing both global and local variables to and from the top of the stack (LDV, STV, LDVA, STVA).
- *Jumps*: Conditional and unconditional jump instructions (JMP, JMPZ, JMPNZ, LOOP).
- *Non-deterministic jumps*: A non-deterministic jump to the given target address (NDET). It behaves like a JMP instruction when it chooses to jump. Otherwise, it performs no operation and continues with the instruction that follows it.
- *Process creation operations*: The spawning of a new process starting at the given address (RUN, LRUN).
- *Procedure calls*: The call to a procedure starting at the given address, as well as the procedure's return (CALL, LCALL, RET).
- *Channel creation operations*: The creation of a new channel with a given message type and a maximum capacity (CHNEW).
- *Channel operations*: Allocating, reading, writing and deallocating messages in channels, as well as obtaining the channel's capacity and current size (CHADD, CHSET, CHGET, CHDEL, CHMAX, CHLEN).
- *Step-completion operations*: The finalization of a process step (STEP). Additionally, a *terminating* step operation signals the finalization of the last step of the process and its corresponding termination.
- *Conditional and unconditional step-abortion operations*: The indication that the current executing process may not execute the current step, and that the execution should backtrack to the previous visible state (NEX, NEXZ, NEXNZ).
- *Speculative control operations*: Operations that guide the flow of execution control in the case of step-abortion operations (ELSE, UNLESS).
- *Other operations*: Arithmetic and boolean operations, stack operations, and output operations.

In this way, the instruction set of NIPS byte-code covers most of the language features which are common in modeling languages. For instance, guards may be implemented by the use of speculative control operations which can cause the process to block at a given state if a condition is not met. This may be combined with channel operations or global variable operations in order to implement process synchronization schemes. Non-deterministic control structures and statements may be implemented with the help of the `NDET` instruction. The channel operations may implement asynchronous as well as synchronous or rendezvous message-passing among processes. Finally, the rest of the instruction set of NIPS covers most “traditional” language features such as arithmetic and boolean operations, conditional and unconditional jumps, scalar variables as well as arrays, and even *print* instructions for the output of strings and integers.

1.2.2 Procedures

As seen in the previous section, NIPS features instructions for calling procedures and returning from them. When a procedure is called, the location of the instruction that follows the call instruction, commonly known as the “return address”, is pushed on the operand stack. Likewise, the execution of a procedure return pops this address from the stack and jumps to it. Since the operand stack is not part of the visible machine state and is lost when the process completes the execution of a step, a usual constraint is that a called procedure must return to its caller before the process step is concluded. A process call can however span more than one step, but it requires the explicit saving and restoration of the return address by the process, possibly in a local variable. However, this is discouraged since the size of an address in NIPS is not officially defined and may be changed without notice.

Another limitation of using procedures in NIPS is the fact that since the return address is on the top of the stack when the called procedure begins its execution, then it needs to perform stack operations in order to save the return address and access any parameters which are passed to it in the stack. At the end of its execution, the procedure needs to restore this return address before the return instruction is executed.

Therefore, although it is possible to use procedures in NIPS, they will normally be used as simple subroutines rather than as fully-featured procedures.

1.3 States

As mentioned earlier, NIPS considers both *visible* and *invisible* states. A *visible* state corresponds to the global state of the virtual machine, and constitutes a state of the generated state-transition system. One of these states is created as the initial state of the computation, and the rest are generated by the execution of `STEP` instructions. These states are composed of the state of the global variables, the states of all channels, the states of all active processes (process ID, program counter and local variables), and additional control information such as the execution modes of the processes.

Additionally, the execution of the virtual machine creates transient *invisible* states which have the same composition as of visible states, augmented with an operand stack and a set of registers which are local to the currently executing process (the active local

```

chan c = [1] of {int};

active proctype send () {
    do
        :: c ! 1
        :: c ! 2
    od
}

active proctype receive () {
    do
        :: c ? 1
        :: c ? 2
    od
}

```

Figure 1.3: An example of NIPS: original model in Promela.

state). From an invisible state only this process may execute its next step. When a visible state is reached, the other processes may execute as well. In this way, the semantics of the transitions between visible states are encoded as sequences of operations of a single process on invisible states.²

Finally, some visible states may be treated by the virtual machine as invisible in some cases, and therefore, do not form a part of the reachable state space of the program. This will be covered in more detail in Chapter 4.

1.4 An example of NIPS

The following is a small example of a model encoded in NIPS byte-code. The original model is written in the Promela modeling language and is shown in Figure 1.3. When the model is compiled with the Promela compiler provided with NIPS [WS05], a NIPS byte-code file is produced. Figure 1.4 shows the byte-code program contained in this file. Finally, this file may be loaded by the NIPS virtual machine for its interpretation. The sample application `nips_vm` included with NIPS provides the user with the option of performing an interactive or random simulation, or a full state space exploration using a breadth-first or a depth-first search. It also allows the user to produce a graphical representation of the state-transition system, such as the one shown in Figure 1.5 which corresponds to the state-transition system generated by this example.

²As an exception, two processes may participate in synchronous communication in a single transition, which may be seen as a joint transition of the two processes.

	GLOBSZ 2		
	LDC -31		P_receive:
	CHNEW 1 1		L4:
	STVA G 2u 0		NDET L6
	LRUN 0 0 P_receive		LDVA G 2u 0
	POP r0		TOP r0
	LRUN 0 0 P_send		CHLEN
	POP r0		NEXZ
	STEP T 0		PUSH r0
			PUSH r0
P_send:			PUSH r0
L0:			CHGETO 0
	NDET L2		LDC 1
	LDVA G 2u 0		EQ
	TOP r0		NEXZ
	CHFREE		POP r0
	NEXZ		CHDEL
	push r0		JMP L7
	CHADD	L6:	
	PUSH r0		LDVA G 2u 0
	LDC 1		TOP r0
	CHSETO 0		CHLEN
	JMP L3		NEXZ
L2:			PUSH r0
	LDVA G 2u 0		PUSH r0
	TOP r0		PUSH r0
	CHFREE		CHGETO 0
	NEXZ		LDC 2
	PUSH r0		EQ
	CHADD		NEXZ
	PUSH r0		POP r0
	LDC 2		CHDEL
	CHSETO 0	L7:	
L3:			STEP N 0
	STEP N 0		JMP L4
	JMP L0	L5:	
L1:			STEP T 0
	STEP T 0		

Figure 1.4: An example of NIPS: generated NIPS byte-code.

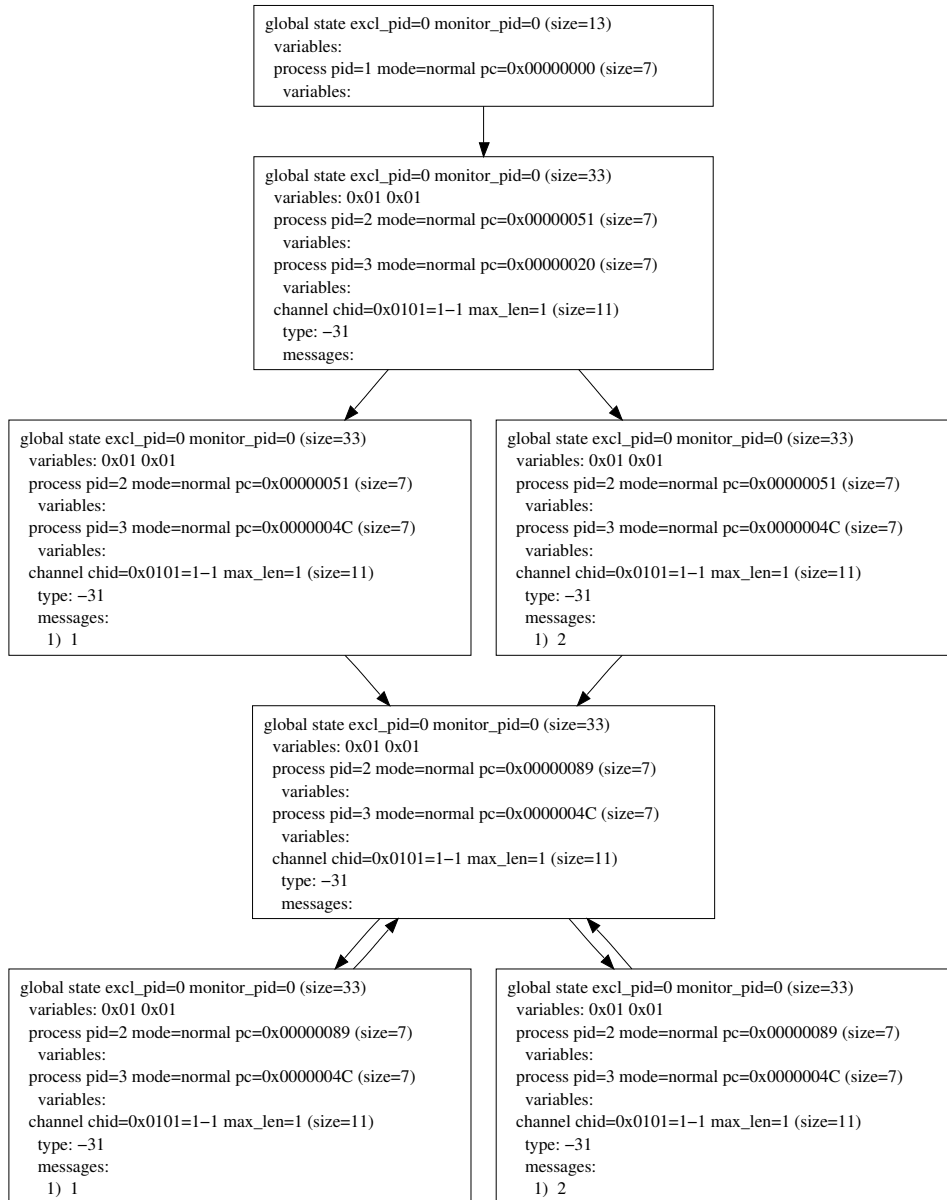


Figure 1.5: An example of NIPS: generated state-transition system.

Chapter 2

Static Analysis for State Space Reduction

NIPS, like any explicit-state exploration tool, suffers from the *state-explosion problem* [CGP99]. The problem lies in the fact that models have exponentially large state-transition systems, that is, that the number of states of a concurrent system grows exponentially as the number of concurrent elements increases. Many techniques for dealing with this problem have been proposed [HP94, CGMP98, BFG99, YG04, Pel05]. These techniques are called *state space reductions*, or *reductions* for short, and they have the goal of reducing the size of the state-transition system of a program without losing the parts of this system which are necessary for temporal logic model checking. Thus, the model checking algorithm will produce the same result when applied to either the original state-transition system or to the reduced transition system, and therefore, the reduced system may be used instead of the original one. This allows for savings in runtime and memory requirements, and in some cases, it even allows the exploration of the entire state-transition system when the full system does not fit in the memory that is available. Therefore, state space reductions constitute a very active area of research, as they support the feasibility of explicit state model checking as an automatic verification technique [CGP99, Pel05].

2.1 State space reduction in NIPS

The purpose of the present work is to alleviate the effect of the state-explosion problem in NIPS, and its central idea is based on the observation that a NIPS byte-code program not only encodes the model, but also the algorithm for constructing its representation as a state-transition system. This is made evident by the existence of the **STEP** operation which instructs the machine to produce a visible state. The great control that a byte-code model has over its state-transition system representation offers the opportunity of applying static program analysis and code optimization to the byte-code, as a way to reduce the size of its reachable state space while preserving equivalence under a given temporal logic specification.

The advantages of such an approach are many, and we detail the most important ones in the following.

1. *Simple byte-code transformation.* The state space reduction is implemented as a source-to-source transformation of the NIPS program, which is a relatively easy task given the simple structure of the byte-code. A byte-code program is analyzed and a modified version of the program is produced, which preserves the original program’s semantics and temporal logic properties, but which induces a smaller state-transition system than the original program. Also, the virtual machine need not be modified or configured in any special way in order to generate the reduced state space.
2. *Static reduction without runtime overhead.* Because the reduction phase occurs off-line, the generation of states is freed from any runtime overhead due to the state space reductions.¹ This also applies to any tool that uses the virtual machine for the generation of states, such as a model checker.
3. *On-the-fly state space reduction.* Once the virtual machine interprets the reduced byte-code, it will create the reduced transition system on-the-fly, without the need to create the original system in part or in whole. This is of great importance as the real limitation of explicit-state exploration tools is the peak memory requirements for the state space, which is not overcome by reduction schemes which require the construction of the full system prior to the reduction phase [Pel05].
4. *Reduce once, run everywhere.*² The reduction may be performed only once on a given byte-code file, yet the reduced byte-code file is usable many times thereafter.
5. *Independence of modeling languages.* The reduction technique operates exclusively on the NIPS byte-code program and is independent of any high-level language which may be implemented in NIPS byte-code. Therefore, the same reductions are applicable to any language that is compiled to the byte-code.
6. *Simple combination of reductions.* Several different types of reductions may be implemented under this framework, and multiple reductions may be “chained” together by setting the output of one reduction as the input of the next. This makes the combination of the effects of multiple reductions a very simple task.

Figure 2.1 shows the general scheme in which the reduction optimizations work. The original byte-code program is given to the reduction tool for its analysis, along with the temporal logic specification that will be used for model checking. With this information, the analysis may accurately determine how to apply the reduction to the model, such that the validity of the given temporal logic specification is preserved in the reduced model. However, it is still possible to apply the reductions without supplying a specification, in which case the reduction may be done in full or conservatively in order to support a large set of temporal logic formulas.³ In any case, the reduction will normally depend

¹That is, other than the interpretation of byte-code instructions which are added to the reduced program for reduction purposes.

²In analogy to the phrase “write once, run everywhere” used in the promotion of Java™ as an architecture-independent computing platform.

³The temporal logic language supported depends on the type of reduction applied, as will be detailed in Chapter 8.

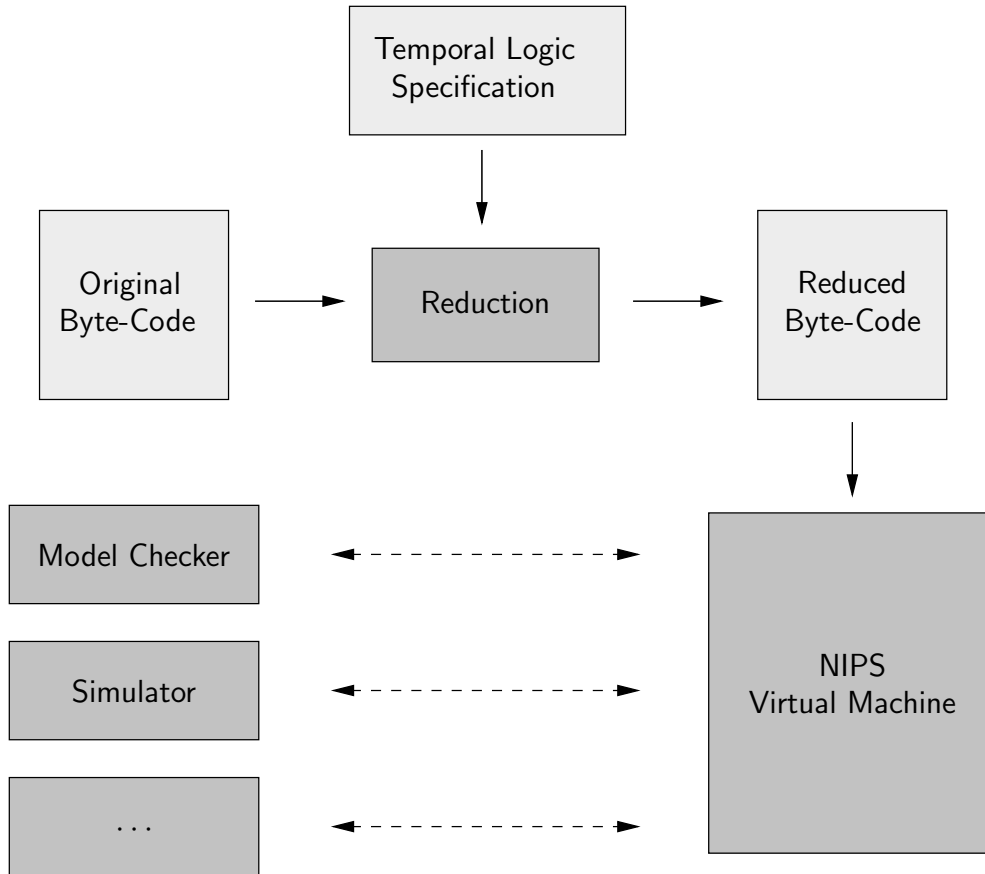


Figure 2.1: Application of state space reduction based on static byte-code analysis.

on just some of the characteristics of the temporal specification, such as the program variables and locations that it refers to. Thus, a single invocation of the reduction can apply to more than one specification. Finally, the result of the analysis is used to perform a code transformation on the given program and produced a reduced version of it. This byte-code file may then be loaded by the virtual machine for its interpretation and used in the same manner as was shown in Figure 1.1.

The chapters that follow will present a framework for static analysis and optimization of NIPS byte-code, as well as the state space reduction techniques *path reduction*, *dead variable reduction* and *step confluence reduction*, which feature all of the advantages mentioned earlier and thus work entirely by static program transformation and construct the reduced transition system *directly* from the model, avoiding the construction of the full state-transition system. Also, these techniques are mostly orthogonal which allows an even better reduction to be obtained when they are combined.

In Chapter 9, an implementation of this state space reduction framework will be presented, and Chapter 10 presents some empirical results that have been obtained from the application of the state space reductions presented here.

Chapter 3

A Model of NIPS

Chapter 1 introduced the NIPS virtual machine and its byte-code language. This chapter presents a formalization of the NIPS byte-code language, its syntax, static and dynamic semantics. It forms a base for the static analyzes, code transformation algorithms and correctness proofs that will be presented in the following chapters.

3.1 Programs

A NIPS byte-code file contains the encoding of a concurrent model that we will refer to as a *program*. A program is composed of instructions, which are the only syntactic constructs of the language. We begin by defining the set of NIPS instructions.

Definition 3.1 (NIPS instructions) *The set of NIPS instructions I over $\langle O, M, A \rangle$ is defined as*

$$I := \{ om_1 \dots m_p a_1 \dots a_q \mid o \in O, m_i \in M, a_j \in A, p, q \geq 0 \}$$

where O is the set of operations, M is the set of modifiers and A is the set of parameters. For a given instruction $i \in I$ we denote the operation of i by $o^i = O$, its set of modifiers by $M^i = \{m_1, \dots, m_p\}$, and its set of parameters by $A^i = \{a_1, \dots, a_q\}$.

The set of operations O corresponds to the set of byte-code mnemonics of NIPS, such as ADD, STV, RUN, etc. We will sometimes refer to a family of operations with an $*$ symbol, such as *CALL for CALL and LCALL. The set of modifiers M contains keywords to characterize the operation, such as the expected data types of the operands, or an indication of local or global variable referencing. The set of parameters A consists of the union of all atomic data-types handled by NIPS (1-, 2- and 4-byte signed integers, 2- and 4-byte program addresses, and register indexes). We assume that instructions are well-formed with respect to modifiers and parameters, that is, that instructions only contain modifiers which are accepted by the instruction's operation, and that also contain a correct number of parameters of the correct data types.

We may now define NIPS programs as follows:

Definition 3.2 (NIPS programs) *The set of NIPS programs P over $\langle O, M, A \rangle$ is defined as*

$$P := \{ i_1 \dots i_n \mid i_j \in I, 1 \leq j \leq n, n \geq 0 \}$$

where I is the set of NIPS instructions over $\langle O, M, A \rangle$. For a given program $p \in P$ where $p = i_1 \dots i_n$, we denote the set of instructions contained in p by $I^p = \{i_1, \dots, i_n\}$.

As before, we assume that programs are well-behaved when handling the stack, registers, variables, channels and program addresses. These assumptions on the well-formedness of code are valid because we will work on the *optimization* of code that has been produced by a compiler or by hand, and that is already functional in its original form. The optimizations will preserve the original model’s semantics, along with any errors it may contain.

As seen in Definition 3.2, programs are simply sequences of instructions. However, within this simple structure lies the encoding of a concurrent system, whose processes have structured control-flow. This requires an instruction addressing system, which we model through the concept of *program locations*.

Definition 3.3 (Program locations) *Given a NIPS program $p = i_1 \dots i_n$, where $1 \leq j \leq n$ and $n \geq 0$, we say that instruction i_j has the location j , which is denoted by Loc_p^j .*

Program locations correspond to the ordinal position of instructions. Program addresses, on the other hand, correspond to the actual byte offset of each instruction. While both systems may be used to unambiguously identify instructions, we choose to work with the more natural program locations, and assume there exists a simple way to map both systems. This mapping will be implicitly expressed via the equivalence operator \equiv .

3.2 Control-flow graphs

The static semantics of a NIPS program is expressed by its control-flow graph. The nodes of the graph correspond to the program’s instructions, and its edges denote the transfer of control from one instruction to another. This transfer of control occurs within the execution of a single process, and in most cases coincides with the sequential execution of the byte-code instructions of the program.

Definition 3.4 (Control-flow graph for NIPS) *The control-flow graph G^p of a NIPS program $p \in P$ is a tuple $\langle N, X, T, E \rangle$ where N is the set of nodes, $X \subseteq N \times N$ is the next-node relation, $T \subseteq N \times N$ is the target-node relation, and $E = N \cup T$ is the edge relation. The control-flow graph G^p denotes a labeled directed graph with one node $n^i \in N$ for each byte-code instruction $i \in I^p$, and one edge from n to n' for each $(n, n') \in E$ labeled with “next” if $(n, n') \in X$ or with “target” if $(n, n') \in T$.*

The *next-node* and *target-node* relations are determined by the control semantics of the byte-code language, and in turn define the edge structure of the control-flow graph. They obey the fact that after any instruction, control may pass to either the next instruction, or to the instruction that corresponds to a given address parameter (we call such an instruction the *target* of the current instruction)¹. We now define these relations formally, first defining their instruction-level counterparts.

¹As an exception, after a procedure call, control passes to the first instruction of the called procedure (i.e. the call’s target), and execution then continues at the instruction following the call once the procedure returns. However, procedure calls will be considered as having sequential control-flow, and will only be handled specially when necessary

Definition 3.5 (Next-instruction relation) For a NIPS program $p = i_1 \dots i_n$ where $n \geq 0$, the next-instruction relation is defined as

$$X_I := \{ (i_j, i_{j+1}) \mid 1 \leq j < n \}.$$

Definition 3.6 (Target-instruction relation) For a NIPS program $p = i_1 \dots i_n$ where $n \geq 0$, the target-instruction relation is defined as

$$T_I := \{ (i_j, i_k) \mid 1 \leq j \leq n, 1 \leq k \leq n, \exists a \in A^{i_j}, k \equiv a \}.$$

Definition 3.7 (Next-node relation) For a NIPS program $p \in P$, the next-node relation is defined as

$$X := \{ (n^i, n^j) \mid (i, j) \in X_I, i \notin S \}$$

where

$$S = \{ \text{JMP } a, \text{ LJMP } a, \text{ RET}, \text{ NEX}, \text{ STEP } T \}$$

is the set of instructions that do not transfer control to its next instruction.

Definition 3.8 (Target-node relation) For a NIPS program $p \in P$, the target-node relation is defined as

$$T := \{ (n^i, n^j) \mid (i, j) \in T_I, i \in S \}$$

where

$$S = \{ \text{JMP } a, \text{ LJMP } a, \text{ JMPZ } a, \text{ JMPNZ } a, \text{ NDET } a, \text{ ELSE } a, \text{ UNLESS } a, \text{ LOOP } aa' \}$$

is the set of instructions that may transfer control to an instruction which is not its next.

The next-node relation is defined for all instructions which may pass control to the next instruction, which excludes unconditional jumps, procedure returns, unconditional step abortions and terminating steps. The target-node relation is defined for all instructions which may pass control to an instruction other than the next instruction. This includes jumps, ELSE, UNLESS and LOOP instructions. Notice that not all instructions that have a target instruction have a node that has a corresponding target node. The most notable cases of this exception are procedure calls and process creation instructions.

The control-flow graph, as defined in Definition 3.4, is really a collection of graphs, one for each process and procedure of the NIPS program. The start node of each graph corresponds to the initial instruction of the process or procedure. This is formalized in the following definitions.

Definition 3.9 (Processes) For a given program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of processes $\text{Prs}^p \subseteq N$ of p is defined as

$$\text{Prs}^p := \{ n^j \mid j \in I^p, \text{Loc}_p^j = 1 \vee (\exists i \in I^p, (i, j) \in T_I, o^i = *RUN) \}.$$

For a given process $s \in \text{Prs}^p$, the set of process-nodes of s , $N_{\text{Prs}^p}^s \subseteq N$ is defined such that $n \in N_{\text{Prs}^p}^s$ iff there exists a control-flow path from s to n .

[0x00] [0] GLOBSZ 4	[0x02b] [43] STEP N 0
[0x02] [2] LDC 0	[0x02d] [45] LDC 1
[0x07] [7] STVA G DWORD 0	[0x032] [50] TRUNC -31
[0x09] [9] LDC 2	[0x034] [52] STVA G DWORD 0
[0x0e] [14] POP 1	[0x036] [54] JMP 14
[0x010] [16] LRUN 0 0 31	[0x039] [57] LDVA G DWORD 0
[0x017] [23] POP 0	[0x03b] [59] NEXZ
[0x019] [25] LOOP 1 -13	[0x03c] [60] STEP N 0
[0x01d] [29] STEP T 0	[0x03e] [62] LDC 0
[0x01f] [31] NDET 23	[0x043] [67] TRUNC -31
[0x022] [34] LDVA G DWORD 0	[0x045] [69] STVA G DWORD 0
[0x024] [36] LDC 0	[0x047] [71] STEP N 0
[0x029] [41] EQ	[0x049] [73] JMP -45
[0x02a] [42] NEXZ	[0x04c] [76] STEP T 0

Figure 3.1: Example of a NIPS control-flow graph: the block of instructions.

Definition 3.10 (Procedures) For a given program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of procedures $\text{Prd}^p \subseteq N$ of p is defined as

$$\text{Prd}^p := \{ n^j \mid j \in I^p, \exists i \in I^p, (i, j) \in T_I, o^i = \text{*CALL} \}.$$

For a given procedure $d \in \text{Prd}^p$, the set of procedure-nodes of d , $N_{\text{Prd}}^d \subseteq N$ is defined such that $n \in N_{\text{Prd}}^d$ iff there exists a control-flow path from d to n .

In this way, processes and procedures are determined by the existence of a corresponding call or instantiation, given the simple structure of the byte-code. Additionally, the first instruction of a NIPS program is always regarded as a process, because the virtual machine starts its execution by instantiating a process at this address. These nodes represent the set of “start” nodes of the control-flow graph. Here we are referring to static process and procedure *types*, rather than to their runtime counterparts process and procedure *instantiations*. For simplicity, we will not make an explicit distinction when referring to these concepts, as it is usually clear from context.

As an example, Figure 3.1 shows the instruction block of a NIPS program, and its corresponding control-flow graph is depicted in Figure 3.2.

3.3 State-transition systems

A NIPS byte-code program induces a state-transition system which corresponds to the program’s dynamic semantics. Building upon the detailed semantics of NIPS presented in [WS05], we will consider transition systems with a set of states Σ_{NIPS} equivalent to the set of global states Γ of the NIPS virtual machine, and with a transition relation \rightarrow corresponding to the *external transition relation* $\rightarrow_{\text{sched}}$. The following definitions model the states of NIPS according to [WS05]:

Definition 3.11 (NIPS states) The set of states of NIPS Σ_{NIPS} corresponds to the set of visible states Γ of the NIPS virtual machine.

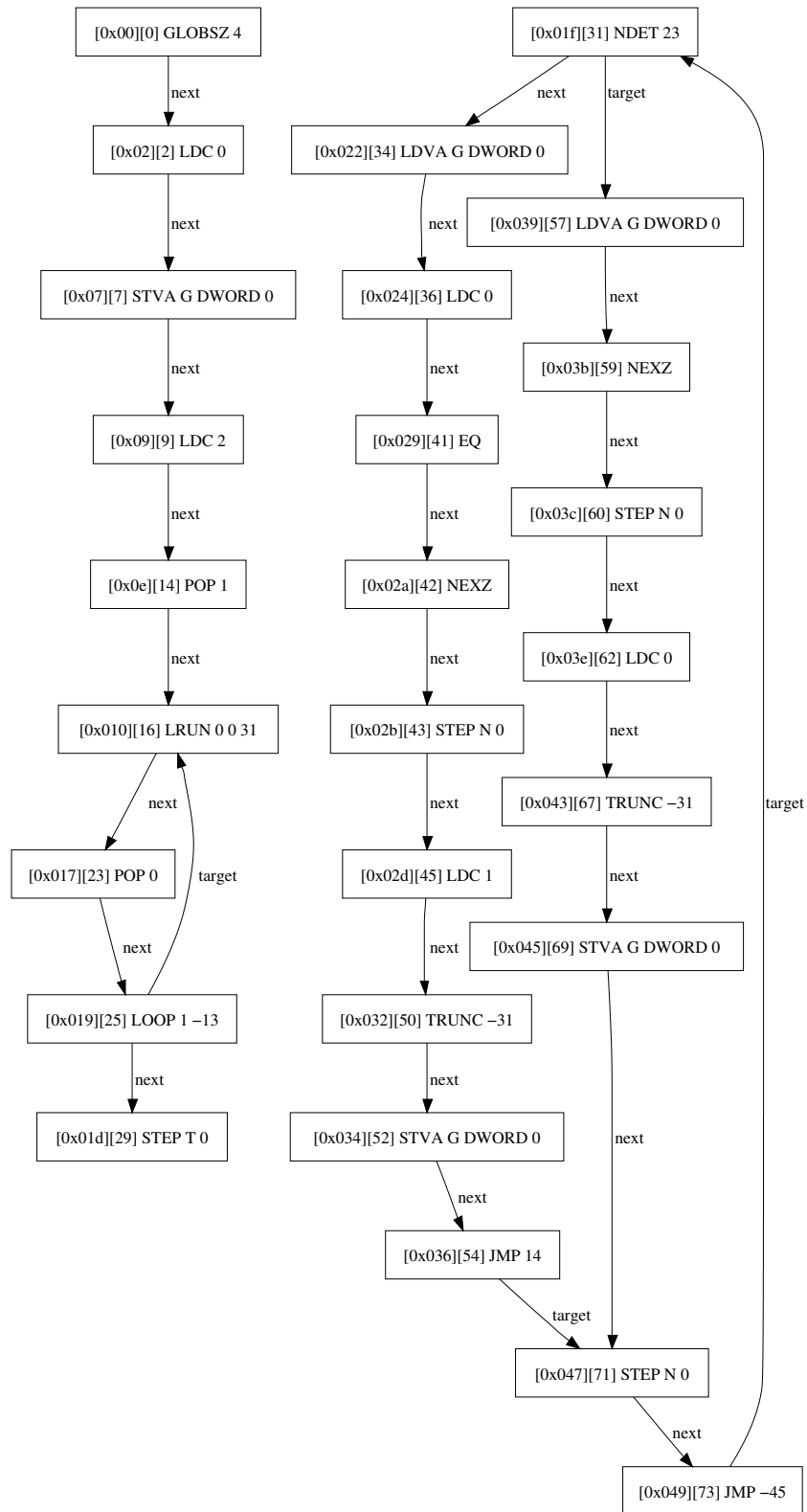


Figure 3.2: Example of a NIPS control-flow graph.

Definition 3.12 (Process identifiers) For a state $\sigma \in \Sigma_{NIPS}$, the set of active process identifiers of σ is denoted by $PID_\sigma \subset \mathbb{N}_0$.

Definition 3.13 (Program counters) For a state $\sigma \in \Sigma$, the function $PC_\sigma : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is defined such that $PC_\sigma(r) = a$ iff $r \in PID_\sigma$ and the address a is the program counter of r in σ .

Definition 3.14 (Variables of a state) For a state $\sigma \in \Sigma_{NIPS}$, the set $GVar_\sigma \subset \mathbb{N}_0$ is defined such that $v \in GVar_\sigma$ iff v is the offset of a global variable in σ . Likewise, the set $LVar_\sigma^r \subset \mathbb{N}_0$ is defined such that $v \in LVar_\sigma^r$ iff $r \in PID_\sigma$ and v is the offset of a local variable of r in σ .

Definition 3.15 (Variable valuations) For a state $\sigma \in \Sigma_{NIPS}$, the function $GVal_\sigma : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is defined such that $GVal_\sigma(v) = u$ iff $v \in GVar_\sigma$ and u is the value of v in σ . Likewise, the function $LVal_\sigma^r : \mathbb{N}_0 \rightarrow \mathbb{N}_0$ is defined such that $LVal_\sigma^r(v) = u$ iff $r \in PID_\sigma$, $v \in LVar_\sigma^r$ and u is the value of v for process r in σ .

Every state $\sigma \in \Sigma_{NIPS}$ is composed of the states of the active processes of σ , the states of all the channels existent in σ , and the global variables of σ . In general, a model checking application may use introspection on a given state σ and allow atomic propositions which reason about any part of the state. However, we will restrict the scope of atomic propositions via the following definition.

Definition 3.16 (Atomic propositions) The set AP_{NIPS} of all atomic propositions of NIPS consists of all boolean formulas over Σ_{NIPS} which only refer to local variables, global variables, the program counters of active processes and the contents of channels.

This scope should cover most specifications used in practice. We need to make this restriction in order to guarantee the equivalence of the original and reduced state-transition systems under a temporal logic specification. We may now define the state-transition system of a NIPS program.

Definition 3.17 (State-transition systems of NIPS programs) A state-transition system TS^p of a NIPS program $p \in P$ is a tuple $\langle \Sigma, \sigma_0, \rightarrow, L \rangle$ where $\Sigma \subseteq \Sigma_{NIPS}$ is a set of states, $\sigma_0 \in \Sigma$ is the initial state, $\rightarrow \subseteq \Sigma \times \Sigma$ is the transition relation, and $L : \Sigma \rightarrow 2^{AP_{NIPS}}$ is a labeling function that assigns to every state a set of atomic propositions true in that state. It holds that $(\sigma, \sigma') \in \rightarrow$ iff $\sigma \rightarrow_{\text{sched}} \sigma'$. We write $\sigma \rightarrow \sigma'$ for $(\sigma, \sigma') \in \rightarrow$.

We say that a transition $\sigma \rightarrow \sigma'$ is *enabled* in state σ , which implies that there exists an active process in σ that may execute from σ and produce state σ' . Additionally, a transition may represent a synchronous communication operation involving two processes. We will model this special case as a sequence of two transitions $\sigma \rightarrow \sigma''$ and $\sigma'' \rightarrow \sigma'$, such that the only transition enabled in σ'' is the second.² The intermediate state σ''

²This is exactly how NIPS handles synchronous communication: a channel of capacity 0 admits a temporary message and produces an “invalid” state, from which the message needs to be received immediately by another process in order to reach a “valid” state. The invalid state is discarded and a transition is created from the state where the communication began to this valid state where the communication has ended.

does not appear in the transition system, but by assuming its existence we may safely assume furthermore that each transition is caused by the execution of exactly one process: the sending process executes the first transition, and the receiving process executes the second one.

Finally, we need to model how NIPS byte-code produces states and transitions. We begin by identifying the blocks of instructions which induce transitions, which correspond to the set of *steps* of the program.

Definition 3.18 (Steps) *Given a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a control path $\pi = n \dots$ where $n \in Prs^p$, the step partition of π is a partition $\pi_{St} = s_1 s_2 \dots$ where $s_i = n^{i_1} \dots n^{i_k}$ and $o^{ij} = \text{STEP} \iff j = k$ for $1 \leq j \leq k$ and $k \geq 1$. The set St^p of all steps of p is defined such that $s \in St^p$ iff there exists a control path π as defined above with a step partition $\pi_{St} = \dots s \dots$.*

A step, as given by the above definition, is a finite control-flow sub-path that has only one **STEP** instruction, namely the last one. Each step either begins at the initial node of a process or follows the **STEP** instruction of the previous step. Therefore, every path of the control-flow graph which starts at the initial node of a process is a sequence of steps, and each step is responsible for inducing transitions in the state-transition system.

Cycles

We assume that every control-flow path is either infinite and “ends” in a cycle or is finite and ends in a **STEP** instruction. For the case of paths with cycles, we assume that each cycle either contains at least one **STEP** instruction, or it contains no such instruction but its execution is guaranteed to eventually exit the cycle and reach a **STEP** instruction (that is, the infinite control path will never represent an actual infinite computation). Otherwise, the above definition would be incomplete, as it suggests that the execution of *any* step will eventually end with the execution of a **STEP** instruction. This assumption is supported by the design of the NIPS virtual machine, because otherwise the machine would not terminate when computing successor states.

Steps and transitions

The notion of steps allow us to relate the structure of NIPS byte-code with the corresponding state-transition system, which we do via the following definitions.

Definition 3.19 (Enabled instructions and steps) *Given a NIPS program $p \in P$ and its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, we say that the instruction $i \in I^p$ (resp. its location) is enabled at state $\sigma \in \Sigma$ iff there exists an $r \in PID_\sigma$ where $PC_\sigma(r) \equiv Loc_p^i$. The set of all instructions enabled at state σ is denoted by I^σ . A step $s \in St^p$ is enabled at σ iff $s = n^{i_1} \dots n^{i_k}$ and $i_1 \in I^\sigma$.*

Definition 3.20 (Step semantics) *Given a NIPS program $p \in P$ and its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, we associate every step $s \in St^p$ with its partial semantic function $[s] : \Sigma \rightarrow \Sigma$. If s is enabled in $\sigma \in \Sigma$ for an active process $r \in PID_\sigma$ and $[s](\sigma) = \sigma'$, then s induces a transition $(\sigma, \sigma') \in \rightarrow$, which we denote as $\sigma \xrightarrow{s}_r \sigma'$.*

The semantics of a step naturally corresponds to the execution of its instructions from a given state by the corresponding process. With these definitions, we clearly mark the relationship between the byte-code and the states and transitions: a state is related to a set of enabled instructions (one for each active process) and thus to a set of enabled steps, and a transition corresponds to the execution of a step by its corresponding process. Notice that only the first instruction of a given step may appear as enabled in a given state. Also, this instruction may be the first instruction for more than one step: because of the local non-determinism present in NIPS, two or more steps of the same process may execute from the same state starting at the same enabled instruction. We consider these steps to be enabled at the same time, and because of this, every individual step is deterministic.

An enabled step s at state σ need not induce a transition $\sigma \xrightarrow{s} \sigma'$ because it may block by executing a NEX* instruction; in this case, we consider $[s](\sigma)$ to be undefined. For completeness, we also consider $[s](\sigma)$ to be undefined if s is not enabled at σ .

Invisible steps and invisible states

NIPS provides a state space reduction mechanism by which a process may execute a sequence of steps as one atomic step and thus avoid the exploration of executions where the steps of other processes are interleaved within this sequence. This is possible by the use of the STEP I instruction, or *invisible step*, which creates *invisible states*.

An invisible state σ is created after the execution of an invisible step s from a state σ' . At σ , the only active process is the one that executed s . If this process executes another step s' from σ and reaches a state σ'' , then the invisible state σ is removed from the state space and a transition $\sigma' \xrightarrow{ss'} \sigma''$ is created. Consequently, this also happens for more than two invisible steps that execute consecutively, always removing the intermediate invisible states. However, if the process blocks in the execution of s' from state σ , then σ is made visible by the virtual machine, and other processes are allowed to execute from σ . Also, if s represents a synchronous send operation, then σ represents the invisible state between the send and receive operations of a synchronous communication operation. In this case, all other processes which are running concurrently with the process that executed s are given a chance to execute one step s'' in order to complete the communication, and for every s'' that executed the matching receive operation and ended in a state σ'' , a transition $\sigma' \xrightarrow{ss''} \sigma''$ is created. Therefore, a STEP I has no effect during normal execution, and behaves as a visible step in the cases of blocking and synchronous communication.

This model of the execution semantics of NIPS is by no means complete, but it suffices for the formulation of the state space reductions that will be presented in the following chapters. For a detailed definition of the semantics of NIPS the reader is referred to [WS05].

Chapter 4

Path Reduction

The first state space reduction technique that we will consider is *path reduction* [Yor00, YG04]. Similar reduction techniques have also been presented under the names of *transition compression* [KLY02] and *statement merging* [Hol99]. Path reduction works by compressing or merging sequences of steps from a given process into one visible step, thus reducing the amount of reachable states while preserving the model's properties with respect to a temporal logic specification.

Path reduction is motivated by the following observations. First, the *granularity* of the steps of a concurrent model has a direct effect on the size of its reachable state space: the more visible steps a process takes in order to perform a computation, the more states that will be needed to represent this computation. Second, in the *interleaving model of concurrency* [CGP99] the execution of a concurrent system is represented by the set of all possible orderings of process steps that may execute in parallel, and the reachable state space corresponds to the cross-product of the state spaces of the individual processes. This is the cause of the state explosion problem. However, it is also well known that a temporal logic specification cannot usually distinguish between different orderings of steps which do not affect each other, and this is the motivation behind *partial order reduction* [HP94, CGMP98]. Third, the effect of step granularity is amplified by this step interleaving: variations in the step granularity of the individual processes will have an exponential effect on size of the full state-transition system once all possible orderings of these steps have been obtained.

With this under consideration, the goal of path reduction is to automatically reduce the step granularity of single processes with the help of static program analysis, in such a way that the equivalence under a given temporal logic specification is preserved. Step granularity is reduced by joining or merging consecutive steps into one visible step under some conditions. When the full state-transition system is computed from the reduced processes, a smaller reachable state space will be obtained because of the observations given above. In a way, path reduction computes an “optimal” step granularity for a concurrent model in a fully automatic manner, under consideration of a given temporal logic formula.

Path reduction tackles the state explosion problem in two ways: first, it compresses computation paths consisting of several transitions into one transition, thus removing the intermediate states from the reachable state space. However, in doing so it also eliminates all step orderings where other processes executed some step in between the steps of

the reduced path. In this sense, an effect which is similar to partial order reduction is obtained. However, this is done statically and requires no runtime component or a modification of the state space search algorithm.

We will first present the version of path reduction from [YG04], on which this work is inspired. We will later present path reduction as it has been applied to NIPS.

4.1 Path reduction for WHILE

The path reduction technique presented in [YG04] is applied to a simple high-level non-deterministic concurrent language. A program in this language is a parallel composition of sequential processes, where each process may be composed of deterministic and non-deterministic local variable assignment statements, **if-then-else** conditional statements, **while-do** looping statements, non-recursive procedure calls and synchronous **send** and **receive** communication statements. From here on we will refer to this language as “WHILE”.

The temporal logic specification language considered in [YG04] is CTL*-X, a subset of CTL* without the X (next step) operator. This operator must be omitted because its validity may not always be preserved when compressing sequences of steps into one visible step [Lam83]. The atomic propositions considered are expressions over local program variables. Given a formula, those program variables that appear in its expressions are called *visible variables*.

The reduction consists of first constructing the control-flow graph of each individual process. Then, the set of *breaking-points*—a subset of the set of nodes of the control-flow graph—is determined, according to the following rules: a node is a *breaking-point* if

1. it is the starting or ending process location.
2. it is an assignment that changes a visible variable.
3. it is a non-deterministic assignment.
4. it is the condition of a **while-do** loop.
5. it is a procedure call, or the statement immediately following a procedure call.
6. it is a **send** or **receive** statement, or the statement immediately following one of these communication statements.

The selection of breaking points determines the set of *elementary paths* in the control-flow graph. An elementary path is a process path that starts and ends at a breaking point, and contains no other breaking points. In this way, only the first statement in the path may influence the specification (because of 2.). Furthermore, it is possible to statically compute the path’s semantics (as a reachability condition and a state transformation function) in a single traversal (because of 3. and 4.). Finally, procedure calls and communication statements are isolated from any other statements in their respective elementary paths (because of 5. and 6.).

The reduced transition system of the program is then produced directly from the control-flow graphs of the processes. The set of breaking points corresponds to the set

of possible program locations of the processes in the states of the transition system, and every elementary path induces a transition of this system. For every such path, two functions are statically computed from the program's syntax: the *reachability condition* is a boolean function over states, that determines if the path may be traversed from a given state, and the *state transformation function* denotes the semantics of the path as a mapping from input states to output states. Together, these functions are used to produce the reduced transition system of each individual processes. For each procedure, a reduced transition system is produced in the same manner, and is inserted in place of the transition of every path that starts at the corresponding procedure call. Finally, the complete transition system is produced as the cross product of the transition systems of the individual processes, additionally creating a single transition out of every matching `send` and `receive` pair.

The reduced transition system created in the above manner is stuttering equivalent [Lam83] to the original transition system, and also equivalent with respect to CTL*-X. Experimental results show reduced transition systems which have state space sizes of 8 to 32% of the original ones.

4.1.1 A comparison of WHILE and NIPS

With the intention of comparing and contrasting the NIPS byte-code language and the WHILE language, we may point out the following aspects of both languages:

- *Variables*: WHILE has only variables which are local to each process, while NIPS has local variables as well as global variables which are shared among processes.
- *Non-determinism*: WHILE presents non-determinism in the form of non-deterministic variable assignments, which restricts local non-determinism to the values of variables. NIPS presents non-deterministic local control-flow, but deterministic variable valuations. Both languages present the global non-determinism inherent to the execution of concurrent processes.
- *Process creation*: WHILE presents static process creation (all process instances are created at startup). NIPS presents dynamic process creation.
- *Communication*: WHILE presents only synchronous communication, while NIPS presents both synchronous and asynchronous communication.
- *Speculative execution*: WHILE presents non-executability (blocking) only for communication statements. NIPS allows any step to block through abortion and roll-back using `NEX*` instructions.

From this analysis, it becomes clear that WHILE spans a semantic subset of NIPS, as it is simple to conceive a straightforward translation from WHILE programs to NIPS byte-code.¹ The non-deterministic assignments of WHILE may be encoded in NIPS as non-deterministic jumps to deterministic variable assignments.

With this in mind, we will present the path reduction applied to the NIPS byte-code as an adaptation of the path reduction as applied to WHILE.

¹However, such a translation is beyond the scope of this work, and therefore not presented.

4.2 Path reduction for NIPS

Our goal is to adapt the path reduction optimization from WHILE programs to NIPS byte-code programs. We build on the original reduction presented in [YG04] mapping each concept with its equivalent in NIPS.

4.2.1 Analysis of breaking points

Consider how a WHILE program could be translated to NIPS byte-code. Each statement of WHILE would be encoded as a step in NIPS, that is, as a sequence of instructions that implement the semantics of the statement, followed by a `STEP` instruction that indicates the conclusion of the step. Therefore, we may treat steps in NIPS in the same way that statements are treated in WHILE. Whereas each control-flow node in WHILE is a statement, each step in NIPS is a control-flow path conformed of one node per instruction. So we will speak of “breaking points” at the level of instructions, and of “breaking steps” at the level of steps.

From WHILE to NIPS

Let’s analyze the motivation behind the definition of breaking points of WHILE programs. We may first notice that the only criterion that regards the specification is 2, and it only considers the modification of visible variables. Since we have defined a much broader scope for atomic propositions in NIPS, we will need to consider as a breaking point any instruction that may alter the truth value of an atomic proposition. This includes assignments to visible variables, enabled instructions whose location is visible, and channel operations on visible channels.

Looking at the other criteria for determining breaking points, we may see that point 1 is related to the definition of elementary paths, which must both start and end at a breaking point. We will therefore consider the first instruction of every process as a breaking point, which also ensures that every process has at least one breaking point. Since we assume that any finite control-flow path ends in a `STEP T` instruction, we can assume the existence of an imaginary final instruction which follows this `STEP T`, and consider it as a breaking point. This in order to avoid marking the last step in each control-flow path as a breaking point, which is unnecessary and limits the effects of the reduction.

Points 3 and 4 are related to the computation of the semantic functions, which are needed in [YG04] in order to encode the transitions of the state-transition system. From Definition 3.18, we can verify that every step is deterministic, as every step is part of only one control-flow path and the non-deterministic jump instruction `NDET` causes a control fork which “splits” the current step into two different steps.² Thus, our model of steps is deterministic and we may drop point 3 from our definition of breaking points.

Point 4 states that every control-flow cycle begins with a breaking step, and therefore allows the computation of a semantic function for the cycle in a single traversal. Notice that NIPS byte-code is an encoding of a state-transition system not unlike the semantic functions from [YG04]. However, it is also more expressive, as a single step may use a

²We consider both steps to be enabled at the same time.

cycle to encode the semantics of a transition. Therefore, we should not require *every* cycle to contain a breaking point. Furthermore, in the case of cycles that contain more than one step, we would like to assert that the virtual machine will not execute within a cycle infinitely, in accordance to the discussion of cycles from Chapter 3. For this, it would suffice for every cycle to contain one breaking point and not necessarily at its first step. For now, we will skip point 4 when defining breaking points in NIPS, and retake this discussion later in the chapter.

Although NIPS has support for procedures, its limitations were already discussed in Chapter 1. Therefore, we will assume the common scenario where the given NIPS program contains procedures which do not make use of any type of recursion, whether direct or indirect. Also, we will assume that the call to every procedure returns before the step has finished its execution, and therefore disallow the occurrence of **STEP** instructions within procedure bodies. Under these assumptions, we may always insert the body of a procedure, minus the **RET** instruction, in place of its call, and with this drop point 5 of the original definition of breaking points when transferring it to NIPS. We will only simulate this code transformation, and consider any ***CALL** instruction itself a breaking point if the procedure it calls contains at least one breaking point.

With point 6 we need to be careful. It defines every communication statement, and the statement which follows it, as a breaking point, thus isolating the communication in its own elementary path. The isolation is again due to the technique used for state space generation in [YG04], which needs to match pairs of send and receive transitions and would be burdened by the combination of a communication statement with any other operation in these transitions. Because of the way the NIPS virtual machine operates, it has no problem with combining communication operations with other non-communication operations in the same step, even in the case of synchronous communication. So we may drop the isolation requirement for communication operations. However, we should also consider that in NIPS, communication occurs not only by synchronous messages, but also by the use of asynchronous channels and shared variables. Thus, we will consider as a communication operation any operation which either obtains information from outside of the local state of the process, or that may alter this external state in a way which is visible by other processes running concurrently. This includes any operation on global variables, channel operations and a few other special operations. In this way, we have considered any possible form of information interchange among processes.

Finally, a step which contains at least one breaking point will be considered a breaking step, which means that it is the first step of a process, its execution may have an influence on the specification or it may interact with its external environment. A step which contains only non-breaking points will be considered itself as a non-breaking step, and may be safely merged with adjacent steps.

With this under consideration, we define the set of *breaking points* of NIPS programs, analogously to the definition given for WHILE programs. We first require some auxiliary definitions.

Influence on the specification

In order to determine which NIPS instructions may have an effect on the valuation of a temporal logic specification, we identify all instructions whose execution may alter the

values of visible variables and visible channels, and which correspond to visible program locations.

Definition 4.1 (Influential instructions) For a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set $Inf_S^p \subseteq I^p$ of influential instructions of p on S is defined such that $i \in Inf_S^p$ iff at least one of the following conditions hold:

1. S refers to Loc_p^i .
2. S refers to at least one channel and $o^i \in \{\text{CHLEN}, \text{CHFREE}, \text{CHADD}, \text{CHDEL}, \text{CHSET}, \text{CHGET}, \text{CHSETO}, \text{CHGETO}, \text{CHSORT}, \text{CHROT}\}$.
3. S refers to at least one global variable and $i = \text{STV G } t$.
4. S refers to at least one local variable from process $s \in Prs^p$, $n^i \in N_{Prs}^s$ and $i = \text{STV L } t$.
5. S refers to a global variable v and $i = \text{STVA G } t v$.
6. S refers to a local variable v from process $s \in Prs^p$, $n^i \in N_{Prs}^s$ and $i = \text{STV L } t v$.

Because we rely exclusively on static information, Inf_S^p is a safe over-approximation of the set of instructions of p which actually have an influence on the specification S . Could we know, for instance, that a given channel operation will *never* operate on a visible channel, then we would not consider this instruction as influential. A similar case would occur if we were able to determine that an assignment on a visible variable will *always* assign this variable to the value that it already has. These situations might be detectable statically in some cases through abstract interpretation of the byte-code. However, in this work we limit ourselves to a purely static approach.

Interprocess communication

As discussed earlier, any operation in which a process receives information from its external environment, or transmits information to it, must be regarded as a form of communication. This motivates the following definition, in which we classify all instructions as either *external* or *internal*.

Definition 4.2 (Internal and external instructions) Given a NIPS program $p \in P$, an instruction $i \in I^p$ is external iff one of the following conditions hold:

1. i is a global variable operation: $i \in \{\text{GLOBSZ } s, \text{LDV G } t, \text{STV G } t, \text{LDVA G } t v, \text{STVA G } t v\}$.
2. i is a channel state operation: $o^i \in \{\text{CHLEN}, \text{CHFREE}, \text{CHADD}, \text{CHDEL}, \text{CHSET}, \text{CHGET}, \text{CHSETO}, \text{CHGETO}, \text{CHSORT}, \text{CHROT}\}$.
3. i is a special NIPS operation: $o^i \in \{\text{LDS}, \text{PCVAL}, \text{LVAR}, \text{ENAB}, \text{MONITOR}\}$.

An instruction $i \in I^p$ is internal iff it is not external.

Any instruction whose execution represents some form of information exchange between the executing process and its external environment is considered an external instruction, and consequently, any other instruction which interacts only with the local environment of the process is regarded as internal. By identifying all external instructions of a given NIPS model, we may detect all steps which represent some form of interprocess communication.

Procedures

As mentioned earlier, we will base our treatment of procedures in the analysis of breaking points and elementary paths on the assumption that procedures are not directly or indirectly recursive, and that procedure bodies do not contain **STEP** instructions. Therefore, we may always obtain an equivalent program by replacing every procedure call by the body of the corresponding procedure, without including the corresponding **RET** instruction. We simulate this replacement and consider a procedure call as a breaking point if the procedure that it calls contains any breaking points. Such a procedure will be considered a *breaking procedure*, and is defined in the following.

Definition 4.3 (Breaking procedures) *For a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , a procedure $d \in Prd^p$ is a breaking procedure iff there exists a breaking point n such that $n \in N_{Prd}^d$.*

Breaking points and breaking steps

Using the definitions of external instructions, influential instructions and breaking procedures, we may now define the set of breaking points of a NIPS program.

Definition 4.4 (Breaking points) *For a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of breaking points $BP^p \subseteq N$ is defined such that $n^i \in BP^p$ iff at least one of the following conditions hold:*

1. $n^i \in Prs^p$.
2. i is external.
3. $i \in Inf_S^p$.
4. $i = *CALL\ a$, $a \equiv Loc_p^{i'}$ and $n^{i'}$ is a breaking procedure.

Thus, we establish four criteria for the selection of breaking points: a breaking point is either the first instruction of a process, an external instruction, an influential instruction, or the call to a breaking procedure. This covers points 1, 2, 5 and 6 of the original definition of breaking points for **WHILE** programs. From this definition, the definition of breaking steps naturally follows.

Definition 4.5 (Breaking steps) *For a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of breaking steps $BSt^p \subseteq St^p$ is defined such that $s \in BSt^p$ iff $s = n_1 \dots n_k$ and $n_j \in BP^p$ for some j , $1 \leq j < k$.*

Notice that a step may contain more than one breaking point. The occurrence of just one breaking point in a step is sufficient for it to be regarded as a breaking step.

$$\pi = \dots \underbrace{i \dots t}_{s} \underbrace{i \dots b \dots t}_{r} \underbrace{i \dots t}_{s} \dots \underbrace{i \dots t}_{s} \underbrace{i \dots b \dots t}_{r} \dots$$

$$\underbrace{\hspace{10em}}_e$$

$$i \in I^p, n^i \notin BP^p$$

$$t \in I^p, o^t = \text{STEP}$$

$$b \in I^p, n^b \in BP^p$$

$$s \in St^p, r \in BSt^p$$

$$e \in EP^p$$

Figure 4.1: Elementary paths in NIPS.

4.2.2 Merging steps

Now that we have a way to determine which steps are breaking steps, we need to know how this information can be used to apply path reduction to the NIPS model. From the definition of elementary paths given in [YG04], we know that only the first WHILE statement in such a path is a breaking point. Thus, one elementary path starts at every breaking point, and contains as many non-breaking points as possible, that is, up to (but not including) the next breaking point. We will use this same idea in NIPS, establishing one elementary path that starts at the first instruction of each breaking step, and which runs up to (but not including) the first instruction of the next breaking step along each control-flow path. Figure 4.1 shows this idea in detail.

Definition 4.6 (Elementary paths) *Given a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a control path $\pi = n \dots$ where $n \in Prs^p$, the elementary partition of π is a partition $\pi_E = e_1 e_2 \dots$ where $e_i = s_1 \dots s_k$, $s_j \in St^p$ and $s_j \in BSt^p \iff j = 1$ for $1 \leq j \leq k$ and $k \geq 1$.*

The set EP^p of all elementary paths of p is defined such that $e \in EP^p$ iff there exists a control path π as defined above with an elementary partition $\pi_E = \dots e \dots$

Now we have all that is needed for the merging of each elementary path into one visible step. The technique for this is to suppress **STEP** instructions within the path, so that when the virtual machine executes its code, it executes the instructions of every one of its steps without executing their **STEP** instructions, and therefore without creating the corresponding visible states. However, it does need to create a visible state once this execution is completed. Therefore, we need to suppress all *but* the last **STEP** instruction of each elementary path. Doing this effectively merges the steps of each elementary path into one step whose semantics is precisely the composition of the semantics of its constituents. The step granularity of the model will have been reduced in such a way that if any two steps s and s' were breaking steps in the original model, then they are contained in two different “merged” steps in the reduced model (because every elementary

path contains only one breaking step), and also, the reduced model contains only breaking steps (because every step resulting from the merging of the steps of an elementary path contains the breaking points of the first step of this path). Thus, the code transformation associated with path reduction is idempotent, and its result can be seen as a “normal form” with respect to step granularity, one in which processes take the longest steps possible while preserving equivalence, and which is adequate for model checking with respect to the given specification.

The described technique for merging steps coincides with the concept of invisible steps provided by NIPS. In effect, path reduction can be seen as a fully automatic way to transform visible steps to invisible steps, under the consideration of process interaction and the preservation of equivalence under a temporal logic specification. Therefore, the natural way to implement path reduction is as a code transformation that substitutes visible steps for invisible ones. However, as we will see, there are some cases when STEP instructions may be removed altogether from the program’s byte-code.

Detection of elementary paths

In order to implement path reduction, we need to locate the elementary paths of the given program. There exists a simple way to determine which STEP instructions represent the last instructions of each elementary path by examining only the instructions and without the need to identify steps and elementary paths directly. This is the technique that we will use in order to determine which STEP instructions should be modified by the code transformation, and it is based on the following lemma which states that the last STEP instruction of every elementary path always reaches a breaking point through a control-flow path, before reaching another STEP instruction.

Lemma 4.1 *Let $p \in P$ be a NIPS program, $\langle N, X, T, E \rangle$ its control-flow graph, S an specification over AP_{NIPS} and $i, i' \in I^p$. Then $o^i = \text{STEP}$, $n^{i'} \in BP^p$ and $\pi = n^i n^{i_1} \dots n^{i_k} n^{i'}$ is a control path such that $o^{i_j} \neq \text{STEP}$ for $1 \leq j \leq k$ and $k \geq 0$, iff there exists an elementary path $e \in EP^p$ such that $e = s_1 \dots s_q$ and $s_q = \dots n^i$ for $q \geq 1$.*

Proof We show the validity of the lemma in both directions, with the help of Figure 4.2 as an illustration.

- \Rightarrow Every STEP instruction is the last instruction of some step $s \in St^p$, so let i be the last instruction of s . From the structure of π we know that i_1 is the first instruction of a breaking step s' , as it follows a STEP instruction and reaches a breaking point before reaching any STEP instruction. Therefore, s' must be the first step of an elementary path e' , and its predecessor step s must consequently be the last step of the previous elementary path e . So we know that the last instruction of e is the last instruction of s , which is i .
- \Leftarrow Every elementary path ends with a STEP instruction, so assume that e ends in i . If no elementary path follows e , then i must be a STEP T instruction and, by the assumption of the definition of breaking points, there exists an imaginary breaking point b which follows i , and $\pi = n^i n^b$ is a control path of p . On the other hand, if an elementary path e' does follow e , then e' must start with a breaking step

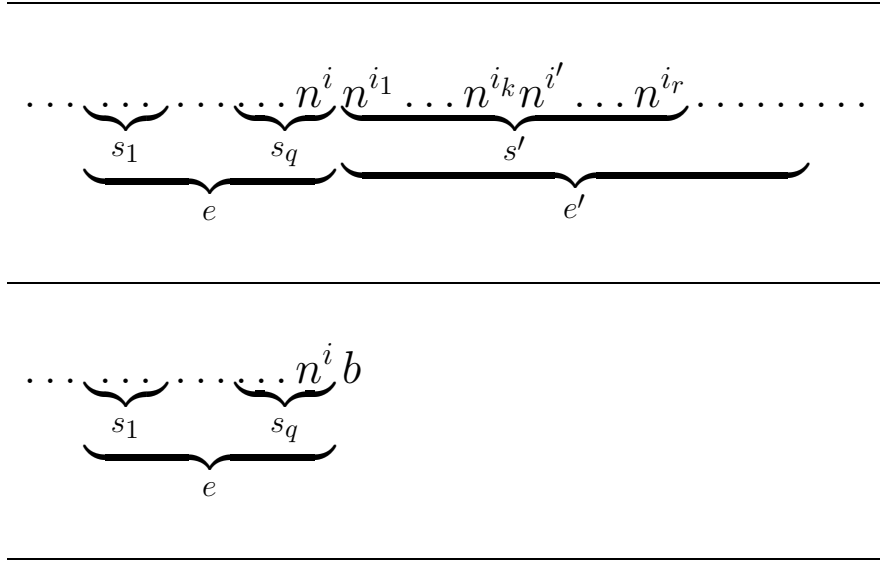


Figure 4.2: Simple detection of elementary paths.

$s' = n^{i_1} \dots n^{i_r}$ which contains a breaking point n^{i_j} for $1 \leq j < r$. So we know that $\pi = n^i n^{i_1} \dots n^{i_j}$ is a control path of p .

□

Lemma 4.1 suggests a simple algorithm for finding the last instruction of every elementary path. These instructions will be called *elementary steps* because they are **STEP** instructions which signal the end of an elementary path.

Definition 4.7 (Elementary steps) For a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of elementary steps $ES^p \subseteq N$ of p is defined such that $n^i \in ES^p$ iff $o^i = \mathbf{STEP}$ and there exists a control path $\pi = n^i n^{i_1} \dots n^{i_k} n^{i'}$ of p such that $n^{i'} \in BP^p$ and $o^{i_j} \neq \mathbf{STEP}$ for $1 \leq j \leq k$ and $k \geq 0$.

Algorithm 1 finds all elementary steps of a program by searching for control paths which match π as stated in Definition 4.7, beginning at each **STEP** instruction. Since any **STEP T** instruction is trivially an elementary step, we don't need to include them in the search. The algorithm traverses the control-flow graph of the program p starting at every **STEP** instruction. If during the traversal a breaking point is found, before finding another **STEP** instruction, then the starting point of the traversal is an elementary step. The search stops at each path when another **STEP** is found or when an already visited node is hit. Thus, each node is visited at most once for each traversal, only one traversal is done for each **STEP** instruction, and the algorithm has linear complexity in the size of the given program p .

Speculative execution and invisible steps

One very important feature of NIPS that has not yet been considered in this analysis is its *speculative execution*. A process execution starting at some state σ may encounter a **NEX*** instruction, which if successful causes the process to backtrack and block at σ . This

Algorithm 1 Elementary steps of a program

Input: $p \in P$, $G^p = \langle N, X, T, E \rangle$, BP^p

```
 $ES^p \leftarrow \{ n^i \in N \mid i = \text{STEP } T \}$ 
for all  $n^i \in N$  with  $o^i = \text{STEP}$  do
   $V \leftarrow \emptyset$  /* Already visited nodes */
   $S \leftarrow \{n^i\}$  /* Nodes scheduled for visit */
  while  $S \neq \emptyset$  and  $n^i \notin ES^p$  do
     $n^{i'} \leftarrow n' \in S$ 
     $S \leftarrow S \setminus n^{i'}$ 
     $V \leftarrow V \cup n^{i'}$ 
    if  $n^{i'} \in BP^p$  then
       $ES^p \leftarrow ES^p \cup \{n^{i'}\}$ 
    else if  $o^{i'} \neq \text{STEP}$  or  $i' = i$  then
      for all  $(n^{i'}, n'') \in E$  do
        if  $n'' \notin S$  and  $n'' \notin V$  then
           $S \leftarrow S \cup \{n''\}$ 
        end if
      end for
    end if
  end while
end for
```

Output: ES^p

means that the corresponding step s is enabled at σ but $[s](\sigma)$ is undefined, and there exists no transition $\sigma \xrightarrow{s} \sigma'$. This suggests that any step s which may block in this way should be considered a breaking step, because if s is a non-breaking step which is part of an elementary path e that starts at the breaking step s' , then once e has been merged into one step, the execution of s can cause e to block at some state σ'' where s' would not normally block. This would incorrectly trim away a portion of the reachable state space, namely, that which occurs after s' executed from σ'' .

However, NIPS handles speculative execution in such a way that it is not necessary to consider every step which blocks as a breaking step. First, notice that a step which blocks does not *always* block, that is, not at every state σ . We could define every step which may block as a breaking step, and obtain a valid reduction because the situation discussed earlier would never happen. However, a better reduction which is still valid is obtained if a non-breaking step is considered as a breaking step only when it blocks. This decision may only be taken at runtime, that is, once the state σ is given, and is precisely the way that invisible steps work in NIPS. If a step blocks at an invisible state, then the state is made visible by the virtual machine and other processes may execute from it. Furthermore, the same step may execute normally from other invisible states which will not be made part of the reachable state space. Then, the use of invisible steps allows us to obtain a good state space reduction while preserving the behavior of non-breaking steps which block.

Notice, however, that non-breaking steps are strictly internal, and therefore cannot interact with other processes. Thus, when a non-breaking step s blocks at a given state

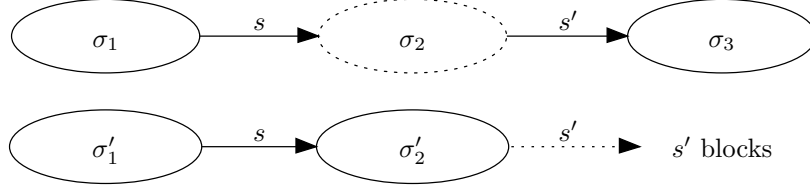


Figure 4.3: Invisible steps and blocking of non-breaking steps.

σ , then s will block at *every* state after σ . This is because the local process state of s will not change with the execution of other processes, so the context that is causing s to block will be preserved. Thus, every time a non-breaking step s blocks at a state σ its process will suffer *starvation* at every state after σ . The use of invisible steps in this case guarantees that if such a situation occurs in the original state-transition system, then the path reduction optimization will preserve this situation because an invisible state σ will be made visible as soon as a non-breaking step s blocks at σ .

Figure 4.3 illustrates this case. From the state σ_1 the step s executes and creates an invisible state σ_2 by executing an invisible step. The next step of this process, s' , executes from this state and creates a visible state σ_3 , as well as a transition $\sigma_1 \xrightarrow{ss'} \sigma_3$. From another state σ'_1 , s executes and again creates an invisible state σ'_2 . However, s' blocks when executing from this second state. This causes σ'_2 to be made visible by the virtual machine, and enables to execution of other processes from σ'_2 . If s' is a non-breaking step, then it will continue to block in the same way at any state after σ'_2 .

In this sense, the use of invisible steps enables us to handle speculative execution without limiting the effects of the reduction, as non-breaking steps which block are only considered breaking steps when necessary. However, there are those non-breaking steps which are known to *never* block. Since we know for sure that an elementary path which only contains these steps will always run to its end, we would like to keep the virtual machine from creating the intermediate invisible states, thus saving some execution time. For this, we identify all steps which may block via the following definition.

Definition 4.8 (Blocking steps) For a NIPS program $p \in P$ and its control-flow graph $\langle N, X, T, E \rangle$, the set of blocking steps $BLS^p \subseteq N$ of p is defined such that $n^i \in BLS^p$ iff $o^i = \text{STEP}$ and there exists a control path $\pi = n^i n^{i_1} \dots n^{i_k} n^{i'}$ of p such that $o^{i'} = \text{NEX*}$, $o^{i_j} \neq \text{STEP}$ for $1 \leq j \leq k$ and $k \geq 0$, and for every edge $(n^i, n^j) \in E$ in π the following holds:

$$\begin{aligned} o^i = \text{ELSE} &\Rightarrow (n^i, n^j) \in T, \\ o^i = \text{UNLESS} &\Rightarrow (n^i, n^j) \in X. \end{aligned}$$

In other words, a STEP instruction is a blocking step if and only if a NEX* instruction is reachable from it through a control path that does not contain any other STEP instruction and that only follows the fall-back edges from the nodes of ELSE and UNLESS operations. This last condition obeys the fact that these instructions “catch” a step abortion caused by the execution of an NEX* instruction which happened along their initial control edge, and redirect execution to their fall-back edge. Thus, the corresponding initial edges need not be checked when looking for NEX* instructions from a possibly blocking step, as the execution of such a NEX* instruction would never cause the step to block.

Notice that Definition 4.8 resembles Definition 4.7. Algorithm 2 computes the set of blocking steps in a similar way in which Algorithm 1 determines the set of elementary steps.

Algorithm 2 Blocking steps of a program

Input: $p \in P$, $G^p = \langle N, X, T, E \rangle$

```

 $BLS^p \leftarrow \emptyset$ 
for all  $n^i \in N$  with  $o^i = \text{STEP}$  do
   $V \leftarrow \emptyset$  /* Already visited nodes */
   $S \leftarrow \{n^i\}$  /* Nodes scheduled for visit */
  while  $S \neq \emptyset$  and  $n^i \notin BLS^p$  do
     $n^{i'} \leftarrow n' \in S$ 
     $S \leftarrow S \setminus n^{i'}$ 
     $V \leftarrow V \cup n^{i'}$ 
    if  $o^{i'} = \text{NEX*}$  then
       $BLS^p \leftarrow BLS^p \cup \{n^i\}$ 
    else if  $o^{i'} \neq \text{STEP}$  or  $i' = i$  then
      if  $o^{i'} = \text{ELSE}$  then
         $E' \leftarrow T$ 
      else if  $o^{i'} = \text{UNLESS}$  then
         $E' \leftarrow X$ 
      else
         $E' \leftarrow E$ 
      end if
      for all  $(n^{i'}, n'') \in E'$  do
        if  $n'' \notin S$  and  $n'' \notin V$  then
           $S \leftarrow S \cup \{n''\}$ 
        end if
      end for
    end if
  end while
end for

```

Output: BLS^p

4.2.3 Byte-code transformation

Simply stated, the byte-code transformation associated with path reduction is the suppression of all STEP instructions which are not elementary steps, that is, all STEP instructions which are the inner steps of an elementary path.

The way NIPS handles invisible steps and states fits our requirements for path reduction. Given an elementary path $e = s_1 \dots s_k$, we can replace the corresponding STEP instructions of the steps s_1, \dots, s_{k-1} by STEP I instructions, and with this cause the execution of e to occur in one step and induce only one transition when none of the steps s_1, \dots, s_k block and when s_1 does not perform a synchronous send operation. If s_1 blocks at some state σ , then e will block as a whole at σ and possibly be able to execute at some

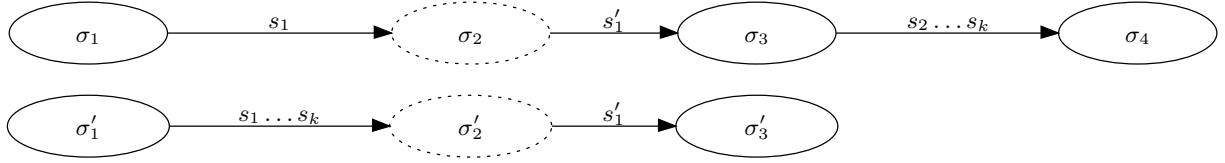


Figure 4.4: Difference between invisible steps and removed steps.

state after σ . If one of the steps s_2, \dots, s_k block, then the state σ where this step blocks will be treated as a visible and reachable state. In this case, e executed only partially and its corresponding process will not be able to continue executing past this point, as already discussed.

However, consider the case where s_1 performs a synchronous send operation. Then, the virtual machine will allow the corresponding receive operation of another process, s'_1 , to execute right after s_1 and before s_2 . This situation would cause the elementary path to create two transitions in the state-transition system: one due to s_1 and the other due to $s_2 \dots s_k$. This situation, depicted in Figure 4.4, limits the effect of the reduction as it produces a larger state space than if $s_1 \dots s_k$ were to execute as a single step. Therefore, we may choose to completely remove the **STEP** instruction corresponding to s_1 rather than replacing it by **STEP I**, because this effectively forces $s_1 \dots s_k$ to occur in one step and to produce just one transition. The corresponding receive operation may be executed by another process once this step has concluded.

Thus, for the case when a **STEP** instruction is neither an elementary step nor a blocking step, we may go further and completely remove it from the program, rather than replacing it with **STEP I**. This saves execution time by avoiding the creation and later deletion of intermediate invisible states, and forces the execution of elementary paths to occur in one step when its corresponding breaking point represents a synchronous send operation.

However, we will also define a version of path reduction which limits itself to the replacement of **STEP** instructions by **STEP I**, and does not remove any such instructions. It produces a program with a somewhat larger state space, but maintains the step structure of the original program, which could be of interest in some situations.

The last two definitions of the analysis required for path reduction determine the sets of **STEP** instructions which are to be replaced by **STEP I** or removed. These are called respectively the sets of *hideable* and *removable steps*.

Definition 4.9 (Hideable steps) For a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of hideable steps $HS^p \subseteq N$ of p is given by

$$HS^p := \{ n^i \mid o^i = \text{STEP} \} \setminus ES^p.$$

Definition 4.10 (Removable steps) For a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of removable steps $RS^p \subseteq HS^p$ of p is defined as

$$RS^p := HS^p \setminus BLS^p.$$

This means that hideable steps are all those **STEP** instructions which are not elementary steps, and removable steps are all those hideable steps which are not blocking steps.

With all this in place, we may define path reduction as a byte-code transformation. We will consider two versions of path reduction: one which replaces all hideable steps with **STEP I** instructions and removes all removable steps, and another which limits itself to the replacement of hideable steps with **STEP I**.

Definition 4.11 (Path reduction for NIPS) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the path reduction of p produces a program $p' \in P$ which is equal to p with each instruction i replaced with **STEP I** iff $n^i \in HS^p \setminus RS^p$, and each instruction i removed iff $n^i \in RS^p$.*

Definition 4.12 (Path reduction for NIPS without step removal) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the path reduction of p without step removal produces a program $p' \in P$ which is equal to p with each instruction i replaced with **STEP I** iff $n^i \in HS^p$.*

According to these definitions, path reduction only alters the step structure of the given program. Depending on which version of path reduction was applied to a given program, some **STEP** instructions now appear as **STEP I** instructions in the reduced program, and some others may have disappeared completely.

4.2.4 Cycles and path reduction

We now turn back to the discussion of point 4 of the definition of breaking points for **WHILE**, and its implications under NIPS. Our main concern is not how to encode cycles –as NIPS represents them as actual cycles in the byte-code–, but rather the *safety* of these cycles.

When applying path reduction to a NIPS byte-code model, there exists the possibility that a cycle in the control-flow graph of the model is left without a single visible **STEP** instruction. As discussed in Chapter 3, a step which executes endlessly within such a cycle will cause the virtual machine not to terminate when computing successor states. A cycle which does not contain visible **STEP** instructions may exhibit this behavior, because no visible state is produced during its execution. Thus, the path reduction optimization may produce a program which, unlike its unreduced counterpart, causes the machine to hang. Were there at least one visible **STEP** instruction in the problematic cycle, then this instruction would cause a visible state to be produced and the infinite cycle would appear in the state-transition system rather than in the machine’s own execution. There are two ways to solve this problem:

1. Ensure that every control-flow cycle contains at least one elementary step. That way, the code transformation which corresponds to path reduction will not suppress these steps, and every cycle will have at least one visible **STEP** instruction.
2. Apply path reduction without considering cycles, but ensure that no cycle in the control-flow graph will cause an infinite execution of a single step. In other words, any step that contains a cycle will eventually exit the cycle and reach a **STEP** instruction.

The first solution may be implemented by a static analysis tool, so we choose to provide it. The second solution is undecidable in the general case, so we avoid it. However, since the second solution provides a better state space reduction than the first, the user may wish to apply it in cases when it is well known (by the user) that the execution of cycles will be safe. Therefore, the implementation of path reduction performs a safety check corresponding to the first solution, which may be disabled at the user's request.

We first define *unsafe cycles*, which are cyclic control paths which do not contain elementary steps.

Definition 4.13 (Unsafe cycles) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , an unsafe cycle of p is a finite path $\pi = n^{i_1} \dots n^{i_k}$ of G^p such that $(n^{i_k}, n^{i_1}) \in E$ and $n^{i_j} \notin ES^p$ for $1 \leq j \leq k$, $k \geq 1$.*

Once we know which cycles are unsafe, we may implement the solution to the aforementioned problem by forcing a visible **STEP** instruction contained in each unsafe cycle to be an elementary step. Of course, it may happen that no such instruction is contained in some unsafe cycle. If this is the case, then the original NIPS model already contained this possibly problematic cycle, which we leave as it is.³ This is because our goal is to *avoid introducing* errors when applying the optimization, but not to *eliminate* possible errors already present in the given program.

Since it is possible for instructions to be shared among cycles, a globally optimal solution would force the least amount of elementary steps such that every unsafe cycle with at least one visible step contains an elementary step. However, in this work we present a simpler solution which analyzes each unsafe cycle individually, and obtains a solution which might not be a global optimum. For every unsafe cycle, we look for a blocking step contained in it. If one is found, we choose this step and set it as an elementary step. The reason for this is that since a blocking step is not removable, it is preferable to set this step as an elementary step instead of any removable step because in this way we obtain a better reduction in the cycle. If the cycle does not contain a single blocking step, then we choose the first visible step in the cycle and set it as an elementary step.

For the implementation of the solution, we need to extend Definition 4.7 so that cycles are now considered when determining the set of elementary steps. We do so with the following definitions.

Definition 4.14 (Forced elementary steps) *For a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of forced elementary steps $FEN^p \subseteq N$ is defined such that $n \in FEN^p$ iff there exists an unsafe cycle $\pi = n^{i_1} \dots n^{i_k}$ in G^p where $n = n^{i_j}$ for $1 \leq j \leq k$, $n^{i_j} = \mathbf{STEP}$, $i_j \neq \mathbf{STEP}$ I and all of the following hold:*

- $n^{i_j} \in BLS^p \Rightarrow n^{i_q} \notin BLS^p$ for $1 \leq q < j$.
- $n^{i_j} \notin BLS^p \Rightarrow n^{i_q} \notin BLS^p$ for $1 \leq q \leq k$.

³In the implementation, a warning message is issued upon this case.

We can verify that FEN^p contains the first blocking **STEP** instruction of every unsafe cycle present in G^p , or the first visible **STEP** instruction if none exists. With this, we may augment the set of elementary steps by including every forced elementary step.

Definition 4.15 (Elementary steps considering cycles) For a NIPS program $p \in P$, its control-flow graph $\langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of elementary steps considering cycles $ES_{CYC}^p \subseteq N$ is defined as $ES_{CYC}^p := ES^p \cup FEN^p$.

The path reduction optimization under the consideration of cycles is obtained by replacing ES^p with ES_{CYC}^p in definitions 4.9 through 4.12. The implementation of the reduction, which will be discussed in Chapter 9, offers the application of path reduction with or without the consideration of cycle safety. Either method may prove more convenient than the other in given situations.

4.3 An example of path reduction

Figure 4.5 shows an example of the code transformation due to path reduction. The code on the left corresponds to the original byte-code program, and the code on the right is the same code after the application of path reduction. We assume that the specification does not refer to the local variables of the process starting at label L1. The program contains a total of four **STEP** instructions, and each of these has been marked with a comment that indicates its type according to the analysis of path reduction. The first is a terminating **STEP** instruction, so it is always regarded as an elementary step. The second is a removable step, because the only control path that leaves this instruction reaches another **STEP** instruction without traversing a single breaking point or a step abortion operation. Thus, the reduced byte-code does not contain this instruction, as represented by the empty line. The third **STEP** is also followed exclusively by non-breaking points, but also by one step abortion operation. Thus, this step is a blocking step and is therefore hideable and not removable, and the reduced byte-code contains an invisible **STEP** instruction in its place. Finally, the fourth **STEP** instruction is followed by a global variable operation (at label L2), and must therefore be regarded as an elementary step.

4.4 Relationship between path reduction and partial order reduction

As mentioned in the introduction to this chapter, there is a strong relationship between path reduction and the well known partial order reduction [HP94, CGMP98, KLM⁺98]. This section will study the nature of this relationship, as well as the similarities and differences between both state space reduction techniques.

Partial order reduction is motivated by the fact that a temporal logic specification cannot distinguish between two executions of a concurrent system which differ only in the ordering of independent transitions. Two transitions $\sigma \xrightarrow{s_1} \sigma_1$ and $\sigma \xrightarrow{s_2} \sigma_2$ are considered *independent* if the following two conditions hold:

1. *Enabledness*: s_2 is enabled at σ_1 and s_1 is enabled at σ_2 .

<pre> GLOBSZ 4 LDC 0 STVA G 4 0 LRUN 4 0 L1 POP r0 STEP T 0 ; Elementary L1: LDC 0 STVA L 4 0 L2: LDVA G 4 0 TRUNC -31 STVA L 4 0 STEP N 0 ; Removable LDVA L 4 0 LDVA L 4 0 MUL TRUNC -31 STVA L 4 0 STEP N 0 ; Blocking LDVA L 4 0 LDC 0 GT NEXZ STEP N 0 ; Elementary LJMP L2 </pre>	<pre> GLOBSZ 4 LDC 0 STVA G 4 0 LRUN 4 0 L1 POP r0 STEP T 0 L1: LDC 0 STVA L 4 0 L2: LDVA G 4 0 TRUNC -31 STVA L 4 0 LDVA L 4 0 LDVA L 4 0 MUL TRUNC -31 STVA L 4 0 STEP I 0 LDVA L 4 0 LDC 0 GT NEXZ STEP N 0 LJMP L2 </pre>
---	---

Figure 4.5: An example of path reduction.

2. *Commutativity*: $\sigma_1 \xrightarrow{s_2} \sigma_3$ and $\sigma_2 \xrightarrow{s_1} \sigma_3$.

Then, two transitions are independent if the execution of one does not disable the other, and the same state is reached after the execution of both transitions in any order. Consequently, two transitions are *dependent* if they are not independent, which means that either the execution of one disables the other or the state which is reached after their combined execution depends on the order in which the transitions are executed.

Partial order reduction modifies the state space search algorithm by following only a subset of the transitions which are enabled at each state σ . This subset is known as the *ample set* of σ , denoted as $ample(\sigma)$, and is defined such that the state space explored is reduced with respect to the full reachable state space, but while preserving the validity of the temporal logic specification of interest. This is obtained by defining the choice of transitions to be contained in the ample set of a state in accordance with some conditions which are not mentioned here but are explained in [CGMP98]. Most importantly, these conditions give a higher priority to those transitions which are independent, causing the reduction to “leave out” any transition which may be deferred from a given state σ and be taken at a later state σ' , such that all the transitions from σ to σ' are independent of this transition. Also, the validity of the temporal logic formula is preserved by ensuring that $ample(\sigma)$ contains all the transitions enabled from σ if at least one of these transitions has an effect on the specification. Finally, every cycle in the produced transition system has at least one state σ such that all its transitions are in $ample(\sigma)$, causing the cycle to be closed and guaranteeing that every transition enabled at any state is actually taken at *some* state. With all this, stuttering equivalence is obtained between the full system

and the reduced system.

Since partial order reduction analyzes *transitions* and not *steps*, its corresponding analysis is inherently dynamic, as it considers the state σ in order to compute the set $\text{ample}(\sigma)$. Although a static version of partial order reduction is presented in [KLM⁺98], it performs some of the analysis statically at compile time, and produces code which completes the determination of the ample sets during the construction of the state-transition system using dynamically generated information. Thus, partial order reduction is able to work with more precise information than path reduction which is based on a purely static analysis.

Several analogies can be made between partial order reduction and path reduction. First, non-breaking steps are always independent of any concurrent step, and thus, will always be contained in the ample set of those states where they appear as enabled. Since path reduction merges non-breaking steps with the previous step from the same process, then this reduction has a similar effect as partial order reduction in the sense that the transition corresponding to the non-breaking step will always be taken before any other concurrent transition. However, path reduction also *eliminates* the state in between both transitions from the same process, which is something that partial order reduction does not do. Second, any step which may affect the specification is considered a breaking step in path reduction, which means that all the possible orderings of this step and other concurrent steps are considered. Similarly, if $\text{ample}(\sigma)$ contains a transition which may affect the specification, then $\text{ample}(\sigma)$ contains every transition enabled at σ , which again explores all possible orderings of this transition and other concurrent transitions. Finally, both reductions must consider cycles in order to ensure that the state exploration algorithm does not explore the states of a cycle infinitely.

However, both reduction techniques *are* different and neither one contains the other completely. In order to see this, consider the example shown in Figure 4.6. Two processes execute concurrently, the first consists of the steps a and b and the second of the steps c and d . The steps a and c are breaking steps, whereas b and d are not. Furthermore, all steps are independent of each other. In the initial state, both processes are about to execute their first steps, and we represent this by $ab|cd$. In the final state, all processes are terminated so we represent this by a blank state. Since all steps are independent of each other, the first state-transition system of Figure 4.6 is produced.

When partial order reduction is applied to this example, the second transition system is generated, where only one of the possible paths of the full system has been explored. This is valid because all steps are independent of each other and therefore their ordering does not alter the temporal logic properties of the system. So the chosen path is a valid representative for all other paths.

Consider now the third transition system of Figure 4.6. Path reduction has been applied to the system, and therefore the steps a and b have been merged into one step, as well as the steps c and d . The system has fewer states than the original system, and even fewer states than the one produced by partial order reduction. Since a and c are breaking steps, the two paths contained in the reduced system represent all possible interleavings of a and c , disregarding those differences among paths due only to the relative ordering of non-breaking steps. Thus, the path $abcd$ represents $acbd$ and $acdb$, and likewise, the path $cdab$ represents both $cabd$ and $cadb$. This is the smallest representation of this system that can be constructed while relying on static information alone.

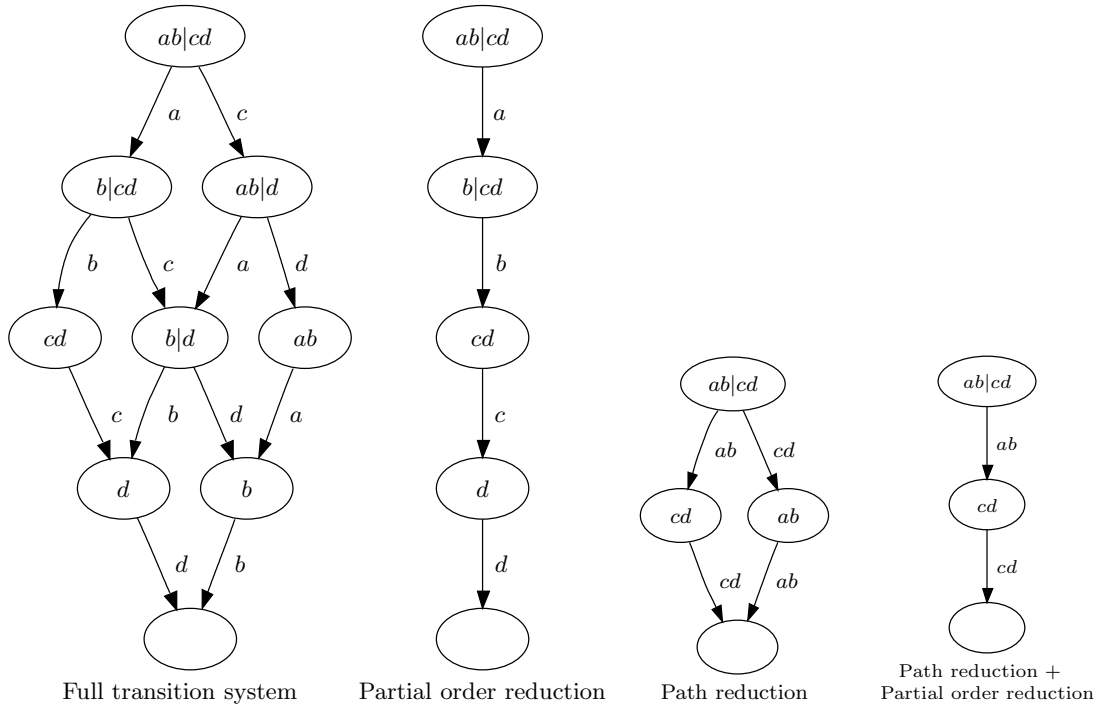


Figure 4.6: Comparison of path reduction and partial order reduction.

Finally, the fourth system of Figure 4.6 presents the combination of path reduction and partial order reduction. It features the compressed steps ab and cd due to path reduction, and additionally contains just one interleaving of these two steps and not both. This happens because the partial order reduction algorithm determines that the steps a and c , as well as their respective compressed steps, are independent, and thus only one of these steps needs to be taken from state $ab|cd$. As may be seen by this example, the combination of both reductions obtains a better reduction than that which is obtained by either reduction alone.

Thus, even though both reductions work by identifying equivalent computation paths which differ only in the ordering of independent events, path reduction compresses computation paths for single processes based on static information while partial order reduction limits the number of explored process interleavings based on dynamic information. Therefore, their respective results will depend on the kind of state-transition system which is subject to the reduction, and in general, a very good result should be achieved when both reductions work together.

Chapter 5

Dead Variable Reduction

The second form of state space reduction that will be presented is called *dead variable reduction*. It is based on the classic *live variables* data-flow analysis for programs [ASU86], and has already been applied to state space reduction of transition systems [BFG99, Hol99, Yor00, YG04].

The main idea behind dead variable reduction may be stated as follows: if two states σ and σ' differ *only* in the valuation of a set of variables V , none of the variables in V are referred to by the specification, and every variable in V is *dead* at both σ and σ' (that is, all variables in V will be defined after σ and σ' before being used), then σ and σ' are *equivalent*¹. From every set of states equivalent in this manner, choose a representative state with an arbitrary valuation for the variables in V . This representative state is used instead of every one of its equivalent states in the state-transition system.

Intuitively, it is clear that this reduction is valid. If two states differ only in the valuation of dead variables, then their “futures” must also be equivalent and will differ at most in the valuation of dead variables, as the values of dead variables cannot affect the execution of the system. Such differences among states are superfluous in state-transition systems, as long as none of the dead variables considered are visible by the specification; only then would we be interested in their value.

Dead variable reduction may be seen as a form of *symmetry reduction* [Pel05]. Symmetry reductions trim the state space by computing sets of equivalent states and choosing a representative from each equivalence set. Dead variable reduction operates precisely in this way.

Dead variable reduction works in two phases: first, a live variable analysis is performed on the model, commonly by means of an iterative data-flow algorithm [ASU86]. Then, the reduction is applied to the model. When the state-transition system is constructed from a high-level language on-the-fly, the system is constructed in such a way that in every generated state, every dead variable receives a standard value (usually 0). In the case of NIPS, the byte-code is modified such that this exact effect is obtained.

¹Bisimilar, as will be shown later

5.1 Variables in NIPS

Variables in NIPS are stored in byte-addressable arrays; one such array for all global variables, and one array of local variables for each process. Variables are manipulated by *load* and *store* operations, which may handle signed and unsigned 1- and 2-byte integers, as well as signed 4-byte integers. Variable operations require a modifier which determines if the corresponding variable is global (**G**) or local (**L**). In both cases, these operations work in the same manner, and their semantics are described in the following (the top of the stack is represented at the right side):

- **LDV**: with a stack $[\dots a]$, load the value v of the variable at address a , leaving the stack as $[\dots v]$.
- **STV**: with a stack $[\dots va]$, store the value v to the variable at address a , leaving the stack as $[\dots]$.
- **LDVA** a : with a stack $[\dots]$, load the value v of the variable at address a , leaving the stack as $[\dots v]$.
- **STVA** a : with a stack $[\dots v]$, store the value v to the variable at address a , leaving the stack as $[\dots]$.

We may observe several important details about the way NIPS handles variables. First, since the address a represents a byte offset and variables may be longer than 1 byte, a model need not be “well-behaved” when handling variables, as it may choose to load and store variables that overlap with each other, or access only part of a variable. Therefore, a safe memory model of variables should consider each individual byte as a variable. Multi-byte accesses may be considered as working with multiple variables at once. Second, since the operations **LDV** and **STV** obtain the address of the variable from the stack, then it is not possible to statically determine which variable is accessed by these operations without performing some type of abstract interpretation on the byte-code. Even then, the value of this address may not be statically determined in cases when it corresponds to a runtime value. In such a case, a static analysis is forced to consider that *any* variable may be accessed by the operation. This may be safely restricted, however, to any global variable in the case that the operation is a global variable operation, and to any local variable of the current process in the case that the operation is a local variable operation. The operations **LDVA** and **STVA** do not present this problem, as the address of the variable is provided as a constant parameter of the corresponding instruction, and therefore, is always statically known.

5.2 Live variable analysis for NIPS

The first step in applying dead variable reduction to NIPS is defining a static analysis of live variables for NIPS. This will be done by adapting the classic live variable analysis of high-level languages presented in [ASU86] to NIPS, under consideration of the way that NIPS handles variables.

A variable v is said to be *live* at a given state σ if for any execution that follows from σ , v is used before it is defined. A variable is defined when a value is assigned to it, and likewise, it is used when its value is read. A static analysis of live variables computes an over-approximation of the set of live variables at a given state, by analyzing the control-flow graph of the program and determining a set of live variables for each program location. In this sense, a variable v is *live* at a given program location l if for any control path that starts at l , v is used before it is defined. It is an over-approximation because a variable v may be live at a given location l because of its use in a control path which cannot be executed from a given state σ where l is enabled. In this case, v would be live at l and yet would not be live at σ . However, it is assured that if v is live at σ , then it is live at the program location enabled at σ . Finally, a variable v which is not live at some state σ (resp. location l) is therefore said to be *dead* at σ (resp. l).

In the case of NIPS, the live variable analysis must determine the set of live variables at each byte-code instruction. The algorithm presented here is limited to *local* variables only, and there are a couple of reasons for this. First, local variables in NIPS correspond to the variables of a sequential program, and the data-flow analysis problem for this case is well-studied and has a simple solution. Although data-flow analysis for concurrent programs has been studied [LC91, IZ93], it usually relies on specific models of synchronization. Processes in NIPS may synchronize themselves through the use of synchronous and asynchronous channels and shared variables, and they are not forced to use a given synchronization scheme. This makes it difficult to determine an accurate and precise data-flow analysis involving global variables in NIPS. Also, global variables are normally used for synchronization and information sharing among processes, and therefore will rarely be dead at a given state. Local variables, on the other hand, are used mostly as temporary storage of values and will likely be dead more often. Finally, local variables are instantiated once for every process instantiation, and therefore usually represent a greater part of the visible state than global variables.

5.2.1 Local storage size analysis

In order to define the live variable analysis for NIPS, we first need to define an auxiliary analysis, called *local storage size analysis*. This analysis computes the size of the local variable array, in bytes, at every byte-code instruction. The size of the local variable array may be set in two ways by the byte-code: as the first parameter of the `*RUN` operation that instantiated the process, or by the process itself via the `LOCSZ` operation.

Definition 5.1 (Local storage size analysis) *The local storage size analysis for a NIPS program $p \in P$ with control-flow graph $G^p = \langle N, X, T, E \rangle$ consists of the solution to the data-flow equation*

$$size(n) = \max\{ gen(n), in(n) \},$$

with

$$in(n) = \max_{(n',n) \in E} size(n'),$$

$$gen(n^i) = \begin{cases} s & \text{if } i = \text{LOCSZ } s, \\ s & \text{if } \exists i' = *RUN \ s \ a \ l, \ l \equiv Loc_p^i \\ 0 & \text{otherwise} \end{cases}$$

where $size(n) \in \mathbb{N}_0$ and $n \in N$.

The local storage size analysis may be solved by a iterative forward data-flow solver, as the value of $size(n)$ depends on the value of $size(n')$ for every predecessor n' of n . The values for this analysis are generated at every `LOCSZ` instruction, and at the start of every process that is instantiated with a `*RUN` instruction. NIPS presents two ways to set the size of the local variable array because the initial process, which starts at address 0, is not instantiated by a `*RUN` operation and needs to use a `LOCSZ` operation in order to set its own local storage size.

5.2.2 Local live variable analysis

We may now define the live variable analysis for NIPS byte-code, which uses the result of the local storage size analysis, and may be solved by an iterative backward data-flow solver.

Definition 5.2 (Local live variable analysis) *The local live variable analysis for a NIPS program $p \in P$ with control-flow graph $G^p = \langle N, X, T, E \rangle$ consists of the solution to the data-flow equation*

$$live(n) = gen(n) \cup (in(n) \setminus kill(n)),$$

with

$$\begin{aligned} in(n) &= \bigcup_{(n,n') \in E} live(n'), \\ gen(n^i) &= \begin{cases} \{a, \dots, a + sizeof(t) - 1\} & \text{if } i = \text{LDVA L } t \ a \\ \{0, \dots, size(n) - 1\} & \text{if } i = \text{LDV L } t \\ \emptyset & \text{otherwise} \end{cases} \\ kill(n^i) &= \begin{cases} \{a, \dots, a + sizeof(t) - 1\} & \text{if } i = \text{STVA L } t \ a \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

where $live(n) \subseteq 2^{\mathbb{N}_0}$, $n \in N$ and $sizeof(t)$ is the size in bytes of the data type t .

This analysis computes a safe over-approximation of the set of live variables at each control-flow node of the byte-code model. First, because it uses the union of the sets of live variables of each successor node as the confluence operation in the definition of $in(n)$, thus considering a variable v as live in n if it is live at *any* successor of n . Second, because it considers that $gen(n)$ corresponds to the set of *all* local variables when n corresponds to an LDV operation, and that $kill(n)$ is the empty set when n corresponds to an STV operation. As already discussed, the use of abstract interpretation may permit a more accurate analysis in some cases. However, these two operations are normally used for the compilation of array accesses in high-level languages, because the actual byte offset of the array cell that is accessed usually needs to be computed at runtime. Therefore, the cases where an abstract interpretation technique could yield improved results are probably few. Since the access to atomic variables is usually implemented with LDVA and STVA operations, this analysis should yield good results when most of the variables in use are atomic.

5.2.3 Data flow analysis of procedures

A data flow analyzer for NIPS must consider procedures when solving a data flow problem for a NIPS program. As in the case of path reduction presented in Chapter 4, we will assume a common scenario where procedures are not directly or indirectly recursive, and where procedure bodies do not contain any **STEP** instructions. Thus, we may replace every procedure call by the body of the corresponding procedure, excluding the **RET** instruction, and apply a data flow analysis that disregards procedures. However, since procedures may be called from more than one location in the code, every node in a procedure body will receive several data flow values, one corresponding to the location of each call. We will allow this to occur, and then compute the unique data flow value for each procedure node in a second phase as a confluence of all the values assigned to the node in the first phase. In this way, the data flow information of the procedures is influenced by all the corresponding calling environments, but in such a way that each individual call to a procedure is handled separately and thus avoids that two different environments exchange information through a commonly called procedure.

5.3 Dead variable reduction for NIPS

The local live variable analysis of NIPS provides the function $live : N \rightarrow 2^{\mathbb{N}_0}$, which assigns a set of live variable locations to each node $n \in N$ of the control-flow graph G^p . This information is all that is needed in order to apply the dead variable reduction to the program p .

5.3.1 Death points

We first define *death points*, which are control-flow nodes where a variable is dead and with an incoming edge where the variable “dies”, that is, an edge that comes from a node where the variable is live.

Definition 5.3 (Death points) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of death points $DP^p \subseteq \mathbb{N}_0 \times N$ of p is defined such that $(v, n) \in DP^p$ iff $(n', n) \in E$, $v \in live(n')$ and $v \notin live(n)$.*

Death points represent those locations in the code where the variable v becomes dead during execution. This may only happen on an edge (n', n) in one of the following cases:

1. n' is a variable load operation. This operation is the last access to the current value of variable v in the current control-flow path.
2. n' is a control fork, and n starts a control path where the current value of v will not be used. In this case, there exists an edge (n', n'') where $n \neq n''$ and n'' starts a control path where the value of v can be used. Thus, v is live at n'' and therefore live at n' , but v is not live at n .

Consider the second case mentioned above. The control fork which occurs at n' may be due to a non-deterministic jump instruction (**NDET**), to a conditional jump instruction

(JMPZ, JMPNZ), or to instructions that handle the flow of speculative execution (ELSE, UNLESS). We have defined death points to be located just after the corresponding fork, which represents the first control-flow edge of the path where the variable is dead, and therefore, the earliest point in the code where we can be sure that the variable is dead. However, we may ask ourselves if it is possible to somehow predict the branch that this fork takes before the actual branching instruction executes. If so, we could find more locations where the variable v is dead, namely, those control-flow nodes before the fork, and this could result in a greater reduction. In the case of non-deterministic jumps, it is impossible to predict which branch of the fork will be followed before the jump executes, so the corresponding death point must be located just after the fork. In the case of conditional jumps and control-flow operations for speculative execution, it may be possible to predict which branch will be followed by using abstract interpretation. This solution is, however, not implemented in this work, where we limit ourselves to reductions based on a purely static approach. Thus, we will always locate death points just after a control-fork when the corresponding variable is live in only one of the control branches. An improvement of the presented solution that uses abstract interpretation to try to predict the result of control forks is nevertheless possible.

5.3.2 Dead assignments

As mentioned earlier, the goal of dead variable reduction is to produce a system where every local variable always holds a default value when it is dead. Death points determine those locations in the code where a variable that was live becomes dead during execution. However, we still need to consider the following cases which do not correspond to death points:

1. *A variable may be dead at the first location of a process.* In this case, the variable will hold the value of 0 because it is the initial value of every variable in NIPS.
2. *A variable may be dead immediately after a corresponding assignment.* In this case, the variable must also be dead at the given assignment, and therefore, this assignment stores a given value to a dead variable, which continues to be dead after the assignment. This could cause a dead variable to obtain a value other than the chosen default value.

The first case does not represent a problem as long as we choose 0 to be the representative value for all dead variables, because then, every variable which is dead at the starting location of a process will already hold this default value. The second case could cause a dead variable to obtain an undesired value. However, it is known that any local variable assignment where the assigned variable is dead immediately after its execution represents dead code [ASU86], and thus may be safely removed from the program. Therefore, and we may solve this problem by removing these assignment instructions, which we define as *dead assignments*.

Definition 5.4 (Dead assignments) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of dead assignments $DA^p \subseteq N$ is defined such that $n^i \in DA^p$ iff for $(n^i, n') \in E$ one of the following conditions hold:*

- $i = \text{STVA L } t \ v$ and $\text{live}(n') \cap \{v, \dots, v + \text{sizeof}(t) - 1\} = \emptyset$.
- $i = \text{STV L } t$ and $\text{live}(n') = \emptyset$.

Notice that for any variable store instruction n there exists a single control edge (n, n') , and thus only one successor n' .

The set of dead assignments is defined as a safe under-approximation of the set of all assignments whose assigned variable remains dead after their execution. First, because it is based on $\text{live}(n')$ which is itself an over-approximation of the set of live variables at n' and can include variables which are not really live at n' at runtime, and second, because it considers an **STV** operation as a dead assignment only if *every* local variable is dead after its execution. Notice that, in the case of **STVA**, every single byte of the assigned variable must be dead at the next instruction in order for it to be a dead assignment, in accordance to the safe memory model that was discussed earlier.

5.3.3 Specification-visible variables

As mentioned earlier, a variable v may only be subject to reduction by dead variable reduction if it is not referred to by the given specification S . Otherwise, the validity of S may be altered by the reduction. Consider the simple case of a safety property that requires that a given local variable v never has some value u when control is at location l . If v is dead at l , then after dead variable reduction, we know that v will always have the value chosen as a representative when l executes. Assume that this value is different from u . Then, the safety property will always be satisfied by the reduced model, even though it could have been violated by the original model.

Therefore, we need to restrict dead variable reduction to those variables which are not visible by the specification S .

5.3.4 The code transformation

Since we want to produce an equivalent program that induces a transition-system where all dead local variables have a fixed valuation, we need to transform code at the locations of each death point and of each dead assignment. The code transformation to be done at a death point is straight-forward: insert code that sets the dying variable to a fixed value. We choose 0 to be this fixed value in order to agree with the initial valuation of variables in NIPS. Likewise, the code transformation associated with dead assignments is the removal of these assignments. However, since variable store operations consume elements from the stack, we cannot just remove these instructions, as this would cause the corresponding elements to remain on the stack and be incorrectly accessed by the instructions that follow. So we need to replace every dead assignment by code that removes the corresponding elements from the stack. This transformation is detailed in the following definition.

Definition 5.5 (Dead variable reduction for NIPS) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the dead variable reduction of p produces a reduced program $p' \in P$ which is equal to p after the following transformations have been applied to it:*

- For every $i \in I^p$ where $(v, n^i) \in DP^p$ and v is not visible by S , insert c before i where

$$c = \text{LDC } 0; \text{TRUNC } 8; \text{STVA L UBYTE } v$$

- For every $n^i \in DA^p$ where $i = \text{STVA L } t \ v$ and v is not visible by S , replace i by c where

$$c = \text{POPX}$$

- For every $n^i \in DA^p$ where $i = \text{STV L } t$, $\exists r \in Prs^p$, $n^i \in N_{Prs}^r$ and no variable of r is visible by S , replace i by c where

$$c = \text{POPX}; \text{POPX}$$

Let us examine what this code transformation accomplishes. In the first case, every death point of a non-visible variable v causes the insertion of a sequence of instructions that stores the value 0 to the byte of memory corresponding to v . This code is inserted before the instruction i of the node n^i contained in the death point, which means that its execution will assign 0 to v before executing i . Therefore, if the death point was caused by a control-flow edge $(n^{i'}, n^i)$, then v will have its original value at i' , where it is still live, and will have the value of 0 at i , where it is dead. When the inserted code finishes its execution it leaves the stack, registers and all variables other than v in the same state as when it started, which is required for the preservation of the original code's semantics.

In the second case, every local **STVA** instruction which constitutes a dead assignment of a non-visible variable is replaced with a **POPX** instruction, which only removes the topmost element from the stack. As a result, when the modified code executes, it will remove from the stack the value that was to be stored in the variable, and will continue without modifying the variable's value.

The third case is similar to the second one, except that it adds two **POPX** instructions instead of one in order to remove both of the parameters for the **STV** instruction from the stack. Additionally, since this instruction does not provide information as to which variable it will modify, we are forced to limit the code transformation to those cases when the specification S does not refer to any of the local variables of the process that contains the assignment, because it is the only way to be certain that the assignment will not affect the value of a visible variable.

As a result, when the reduced program is executed, all non-visible local variables which can be statically determined to be dead at some program location l will have a default value of 0 when the program is executing at l . If a live variable becomes dead during execution, then the inserted code sequence for the corresponding death point will assign it to a value of 0. If a variable is dead at the initial location of the process, then it will also have the initial value of 0, as already discussed. Finally, if a variable is assigned a value by a dead assignment, then this assignment will not be present in the reduced program and the dead variable will continue to hold its original value past the execution of this code, and this value will have to be 0 because of the previous conditions.

5.4 An example of dead variable reduction

We now show the effect of the code transformation due to dead variable reduction by means of an example. Figure 5.1 shows a NIPS byte-code program before and after the reduction, analogously to Figure 4.5. The program contains several load and store operations on the global variable 0 and the local variables 0 and 1 of the process that begins at label L2. Of these operations, two local store instructions are dead assignments and they are marked with the corresponding comments. Both are contained in a procedure which begins at label L1, and they represent dead assignments because both of their corresponding local variables 0 and 1 are dead at the only calling context of this procedure, which is located at label L2. Thus, these instructions are replaced by POPX instructions in the reduced program. Also, this code contains two death points for the variables 0 and 1 respectively. They are both located after the last access to each corresponding variable in the control-flow path which begins at L3 and which represents a control cycle. Thus, the instruction sequence required to set these variables to the default value of 0 is inserted before each corresponding death point in the reduced program. Notice that this code executes “in between” the execution of the code that represents an arithmetic operation. This is because dead variables are reset to a default value as soon as they become dead in the execution of the byte-code. The use of an operand stack permits this code to be easily inserted without interfering with other ongoing operations.

			GLOBSZ 1
			LDC 0
			STVA G 1u 0
			LRUN 2 0 L2
			POP r0
			STEP T 0
L1:	LDC 0		L1: LDC 0
	STVA L 1u 0	; Dead assign.	POPX
	LDC 0		LDC 0
	STVA L 1u 1	; Dead assign.	POPX
	RET		RET
L2:	LCALL L1		L2: LCALL L1
L3:	LDVA G 1u 0		L3: LDVA G 1u 0
	TRUNC 8		TRUNC 8
	STVA L 1u 0		STVA L 1u 0
	STEP N 0		STEP N 0
	LDVA L 1u 0		LDVA L 1u 0
	LDVA L 1u 0		LDVA L 1u 0
	ADD	; Death pt. (0)	LDC 0
	TRUNC 8		TRUNC 8
	STVA L 1u 1		STVA L 1u 0
	STEP N 0		ADD
	LDVA L 1u 1		TRUNC 8
	LDVA L 1u 1		STVA L 1u 1
	MUL	; Death pt. (1)	STEP N 0
	TRUNC 8		LDVA L 1u 1
	STVA G 1u 0		LDC 0
	STEP N 0		TRUNC 8
	LJMP L3		STVA L 1u 1
			MUL
			TRUNC 8
			STVA G 1u 0
			STEP N 0
			LJMP L3

Figure 5.1: An example of dead variable reduction.

Chapter 6

Step Confluence Reduction

This chapter presents the *step confluence reduction*, which is a technique for reducing the size of the reachable state space of a model by reducing the quantity of program locations that may appear in a state.

The reduction works by identifying sets of equivalent program locations which may appear as enabled in a state. For each of these sets, a representative is chosen and all other locations in the set are replaced by this representative. This causes that two states in the original system which differ only in enabled program locations which are equivalent, will be represented by just one state in the reduced system. In order to preserve the validity of temporal logic specifications, this reduction must be restricted to those program locations which are not contained in the temporal logic formula. Otherwise, a location which is referred to by the specification could be removed from the program or be equated with a different location.

Step confluence reduction is closely related to both path reduction and to dead variable reduction. Like path reduction, it reduces the quantity of program locations which may appear as enabled in a state. However, whereas path reduction detects equivalent consecutive locations in a single control path, step confluence reduction works with arbitrary *sets* of locations, identifying equivalent locations from possibly different paths and merging them into one location. And like dead variable reduction, it identifies classes of states which differ only in values which are equivalent and which therefore have equivalent “futures”, and chooses one representative from each equivalence class which replaces every state in its class. This may be generalized to the notion of *equivalent values* and its corresponding reduction [Pel05], of which both dead variable reduction and step confluence reduction are specific cases. However, while dead variable reduction reduces according to the values of dead variables, step confluence reduction reduces with respect to program locations and thus with respect to the value of the program counter of each process.

6.1 Equivalent steps

The only program locations that may appear as enabled in a given state correspond to the initial locations of every step in the model. Thus, two different program locations are equivalent if the steps that may be executed starting from them are really the same. This may happen when two steps start at different locations but then “become” the same step via jumps. Consider the NIPS program fragment depicted at the left side of Figure

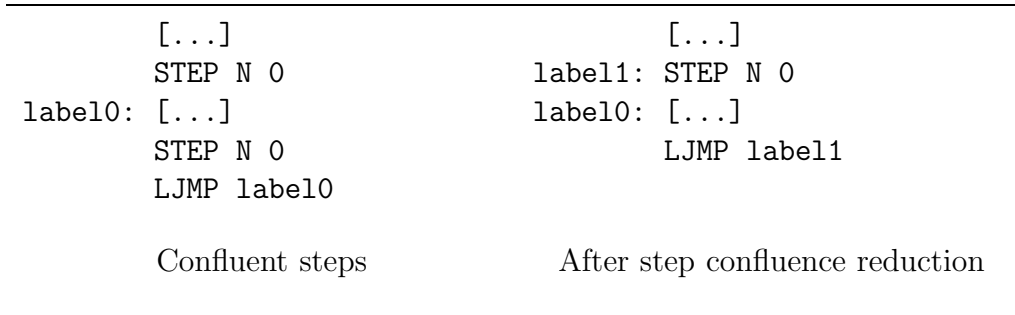


Figure 6.1: Equivalent steps in NIPS.

6.1. A `STEP` instruction is followed by a label `label0`. Later in the code, another step instruction is followed by a jump to this label. Therefore, the code that follows both `STEP` instructions is really the same, and the corresponding steps differ only in an `LJMP` instruction. Thus, the location of both `STEP` instructions is equivalent. It is easy to see that these two locations are interchangeable during execution without any change to the program’s semantics. Moreover, when the program has executed the first `STEP`, the instruction which follows and which corresponds to `label0` is enabled at the next state. When the program has executed the second `STEP`, it is the location of the `LJMP` instruction which appears as enabled in the next state. Since these two locations are different, they will induce different states. Step confluence reduction avoids this by performing the code transformation depicted at the right side of Figure 6.1.

The reduction detects that both `STEP` instructions are equivalent and chooses the first as a representative. Then, it replaces the second one by a jump to the first one, which is now labeled with `label1`. Thus, when the execution reaches this jump, it will go to the first `STEP` and finish the execution of the current step at `label1`, causing `label0` to appear as enabled in the produced state. This is the same behavior of the execution of the code that precedes the first `STEP` instruction. Therefore, both steps will finish their execution with the same `STEP` instruction, and cause the same location to be enabled in the produced state. Because of this, these steps are said to be *confluent*, and the task of step confluence reduction is to detect steps that can be made confluent and cause them to share the corresponding `STEP` instruction and end at the same code location.

Figure 6.1 shows a scenario not at all uncommon in the compilation of high-level languages to NIPS. If a language presents the usual cyclic control structures like `for` or `while` loops, and even jumping statements like `break` or `goto`, then a compiler for this language will most likely emit NIPS code that contains confluent steps. For instance, the Promela compiler provided with NIPS [WS05] generates steps like these in the compilation of `do` loops, `break` jumps and `goto` jumps. The detection and optimization of these confluent steps is simpler to achieve at the byte-code level than at the level of PROMELA code.

6.2 Step confluence analysis

We begin the presentation of step confluence reduction for NIPS by analyzing the characteristics of confluent steps. We note that two steps are confluent if they have different

starting points but have the same *effective* instructions, that is, they share all the instructions which perform operations on the stack, registers, local and global variables, channels, etc. Therefore, confluent steps have a common suffix of effective instructions, and differ in a prefix of *displacement* instructions. In this analysis, we will consider as displacement instructions all those instructions whose execution does nothing more than to change the value of the program counter. In the case of NIPS these instructions are JMP, LJMP and NOP. The first two are unconditional jumps to a given address, and the third is the instruction that does nothing and then jumps to its next instruction.

6.2.1 Confluence points

The way to determine if two steps are confluent is by removing the longest possible prefix of displacement instructions from each step, and then testing if the remaining instructions are equal. Here, equality will be used in its strongest form, that is, by considering instructions to be equal if they are the same, in other words, if they have the same location in the program. One could think about a weaker form of equality where two instructions are equal if they have the same operation, modifiers and parameters, even if their locations differ. This would permit the *factorization* of duplicated code in the program. However, by restricting the notion of equality among instructions we may implement a simpler analysis and code transformation which works well for the case of code that results from the compilation of control-flow structures in high-level languages.

Thus, it suffices to identify the *first* effective instruction of each step, and then compare these instructions among steps. If two steps share such an instruction, then it means that the steps are confluent, because they start their execution at the same location after possibly executing some displacement instructions, and will clearly execute in the same manner past this initial location. These initial locations are referred to as *confluence points*, and they are defined in the following.

Definition 6.1 (Confluence points) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the partial function $CP^p : N \rightarrow N$ of p is defined such that $CP^p(n) = n^i$ iff there exists a control path $\pi = nn^{i_1} \dots n^{i_k} n^i$ in G^p such that $k \geq 0$, $o^{i_j} \in \{\text{JMP, LJMP, NOP}\}$ for every $1 \leq j \leq k$ and $o^i \notin \{\text{JMP, LJMP, NOP}\}$.*

The function CP^p maps a control node n to its confluence point, which is the control node of an effective instruction that is reachable from n by a control path that only traverses displacement instructions, and is undefined for those nodes which have no successors. Notice that all displacement instructions considered here have only one successor in the control-flow graph, and therefore, the path π makes no forks. However, n could itself have two successors, in which case $CP^p(n)$ is ambiguous. We will avoid this situation by restricting the domain of CP^p to those instructions which have *exactly* one successor in the control-flow graph. This does not represent a problem because we are interested in $CP^p(n)$ only for the case when n represents a STEP instruction.

6.2.2 Confluent instructions

Simply stated, two instructions are confluent if they share a confluence point. Thus, after executing any two confluent instructions, the same effective instruction will be executed

next, as well as the code that follows it. This is stated in the following definition.

Definition 6.2 (Confluent instructions) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, two instructions $i, i' \in I^p$ are confluent, denoted as $i \bowtie i'$, iff $i \neq i'$ and $CP^p(i) = CP^p(i')$.*

The confluence relation is symmetric and transitive, but not reflexive. It is defined in this way because any instruction would be trivially confluent with itself otherwise. Now that we have a way to identify instructions which are confluent, we will restrict our attention to those instructions which represent the last instruction of each step, namely, **STEP** instructions.

6.2.3 Confluent steps

We know that two confluent instructions are followed by the same effective code in the control-flow graph. Additionally, if two confluent instructions are identical, that is, they have the same operation, modifiers and parameters, then it is clear that the locations of these two instructions are interchangeable during the execution of the program without any change to its outcome: when the program is about to execute one of these instructions, it could *instantly* jump to the other instruction and execute that one instead. Therefore, it is unnecessary to have two confluent and identical instructions in a program, and a way to avoid this situation is precisely to replace one such instruction by a jump to its confluent counterpart.

Consider again the example shown in Figure 6.1. The two **STEP** instructions are confluent because they have the same confluence point at the start of the code labeled by `label10`. Additionally, both instructions are identical, and therefore, the code transformation shown on the right side of Figure 6.1 is valid, because it replaces one of the **STEP** instructions by a jump to the other one.

Since we have the goal of reducing the size of the reachable state space of the program, we focus our attention on applying this code reduction to those locations which may appear as enabled in a given state. As discussed earlier, this would consist of identifying steps which only differ in a possibly empty prefix of displacement instructions, and forcing them to be same step. From the analysis of elementary steps of Chapter 4, we know that every non-terminating **STEP** instruction marks the end of a step and the beginning of the next step. Thus, the code locations which may appear as enabled in a state are those which follow a **STEP** instruction in the code. If two **STEP** instructions are confluent, as in the case depicted in Figure 6.1, then it must mean that the steps which follow each of these instructions are confluent as well, because they have the same initial effective location, which corresponds precisely with the confluence point of the **STEP** instructions. We may then force these two steps to be the same step by removing the code which makes them different, in this case, the instruction `LJMP label10`, and equating the **STEP** instructions which precede these steps by replacing one of them by a jump to the other. As already mentioned, this entire reduction is only valid if both **STEP** instructions are identical. The resulting code has no confluent steps, and as a result, less **STEP** instructions which cause less locations to appear as enabled in the state-transition system.

Visible locations

As in the case of the state space reductions presented in the previous chapters, step confluence reduction should preserve the equivalence of the induced transition system with respect to temporal logic specifications. In this sense, we need to treat any program locations which are contained in the given specification in a special way. If a given program location which is referred to by the specification is removed from the program, or is chosen as a representative for other locations, then this will certainly alter the validity of the temporal logic formula with respect to the produced transition system. Thus, we need to exclude from this reduction any confluent step which starts at a visible location, and therefore, any **STEP** instruction which is directly followed by such a location. In this manner, we guarantee that any visible location which is present in the original system will also be present in the reduced system, and that it will not be aliased with any other location in the system. This is the reason behind the following definition.

Definition 6.3 (Visible steps) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of visible steps $VS_S^p \subseteq I^p$ is defined such that $i \in VS_S^p$ iff $o^i = \mathbf{STEP}$ and there exists $(n^i, n^{i'}) \in X$ such that $Loc_p^{i'}$ is visible by S .*

Thus, visible steps are all those **STEP** instructions which are followed by a visible location, and must therefore be excluded from the reduction.

Confluence sets and confluent representative

From the discussion of the previous sections it follows that the way to implement step confluence reduction is to replace every **STEP** instruction by a jump to another **STEP** instruction which is confluent and identical to the first. However, there may exist the **STEP** instructions i_1, \dots, i_n such that *all* are confluent and identical to each other. These instructions conform a *confluence set*. Thus, we need to choose one unique representative from each confluence set. Furthermore, we need to exclude every visible **STEP** instruction from this analysis in order to preserve the model's temporal logic properties. This is expressed by the following definitions.

Definition 6.4 (Confluence sets) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the confluence set $CS_S^p(i) \subseteq I^p$ of $i \in I^p$ is defined such that $i' \in CS_S^p(i)$ iff $o^i = o^{i'} = \mathbf{STEP}$, $M^i = M^{i'}$, $A^i = A^{i'}$, $i \notin VS_S^p$, $i' \notin VS_S^p$ and $i \bowtie i'$.*

Definition 6.5 (Confluent representative) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the confluent representative partial function $CR_S^p : I^p \rightarrow I^p$ is defined such that $CR_S^p(i) = i'$ iff $i' \in CS_S^p(i)$ and $Loc_p^{i'} = \min\{ Loc_p^{i''} \mid i'' \in CS_S^p(i) \}$.*

Therefore, the confluence set of a non-visible **STEP** instruction i contains every non-visible **STEP** instruction i' such that i and i' are confluent and identical, and we choose as a confluent representative i' which is contained in the confluence set of i and which first appears in the program's byte-code. This means that if $CR_S^p(i) = i'$, then i

may be safely replaced by a jump to i' under consideration of the program's semantics and the validity of the specification S . It also means that if $CR_S^p(i_1) = i'_1$, $CR_S^p(i_2) = i'_2$ and $i_1 \bowtie i_2$, then $i'_1 = i'_2$ and both i_1 and i_2 will be replaced by jumps to the same representative. This is guaranteed by the constraint in Definition 6.5 which chooses that confluent instruction with the smallest address as a representative.

6.3 Step confluence transformation

The result of the analysis given by Definition 6.5 may be directly used in the code transformation associated with step confluence reduction. As seen in Figure 6.1, the transformation consists of replacing every **STEP** instruction i by a jump to i' if $i' = CR_S^p(i)$. However, we also need to remove the prefix of displacement instructions from the corresponding confluent step. For this we may note that once the replacement of the **STEP** instruction has been applied, this prefix becomes *unreachable code*, that is, it consists of instructions whose control nodes are not reachable from an initial node through any control path. Therefore, we may apply this transformation followed by another which removes any unreachable instructions from the program.

Definition 6.6 (Unreachable instructions) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, an instruction $i \in I^p$ is unreachable iff there exists no control path $\pi = n^{i'} \dots n^i$ in G^p where $i' \in Prs^p \cup Prd^p$.*

We may now give the definition for the code transformation of step confluence reduction.

Definition 6.7 (Step confluence reduction for NIPS) *Given a NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the step confluence reduction of p produces a program $p' \in P$ which is equal to p with each instruction i replaced with **LJMP** a iff $i' = CR_S^p(i)$ and $a \equiv Loc_p^{i'}$, and with every instruction i'' removed iff i'' is unreachable.*

After applying this code transformation, the resulting byte-code program will have no confluent steps unless they are visible by the specification. As already discussed, less program locations can appear as enabled in the state-transition system, which may result in less reachable states.

Chapter 7

Reduction of Sequential NIPS Models

Thus far, we have only considered models which consist of multiple processes running concurrently, which is the most common scenario for model checking. However, single-process models may also be encoded in NIPS byte-code. Although they lack concurrency and therefore do not suffer the state explosion problem caused by the process interleaving model, they may however use local non-determinism to express complex behaviors and also produce large state spaces in this way.

The reductions that have been presented so far may also be applied to these sequential models, with the additional benefit that their conditions for the reduction are relaxed. For instance, path reduction may now consider both internal and external operations to be non-breaking, as there exists no other process to consider for interaction. Likewise, dead variable reduction may now safely operate on both local and global variables in the same manner, because there exists no other process that may read or write global variables.

Another advantage of applying these reductions based on static analysis to sequential programs is that they are able to achieve important state space reductions even in the absence of concurrency. The same cannot be said for state space reductions based on partial order methods [HP94, CGMP98, CGP99], which work inherently with concurrent models.

In essence, this chapter presents modified versions of the reductions which consider the special case of a sequential NIPS model. Since it is possible to automatically detect such a case, we begin by presenting the algorithm behind this detection. The implementation may therefore decide whether a sequential or a concurrent model has been given to it, and apply the corresponding reduction in each case.

7.1 Sequential NIPS models

A sequential model is one which presents no concurrency, and therefore, one which induces a state-transition system where only one process is active at any state. This includes, but is not limited to, those models where only one process instance is activated during their entire execution, meaning that the *same* process is activated at any state. Since

this case may be detected statically in a precise and efficient manner, we will implement the detection of sequential models in this way.

The execution of every NIPS program begins with the instantiation of the initial process, which in turn begins its execution at the first instruction of the byte-code. Thus, we will consider this process instantiation as the one and only process instantiation of a sequential system in NIPS. Any program whose initial process contains at least one reachable `*RUN` instruction in its code is capable of executing this instruction to create a second process instance. Therefore, we will consider such a program as concurrent. This is stated in the following definition.

Definition 7.1 (Sequential and concurrent NIPS programs) *Given a NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, p is concurrent iff there exists a path $\pi = n^i \dots n^{i'}$ in G^p such that $Loc_p^i = 1$ and $o^{i'} = \text{*RUN}$. Consequently, p is sequential iff p is not concurrent.*

With this simple detection, we are able to automatically distinguish between sequential and concurrent models, and thus, to decide on which version of the reduction to apply to a given model.

7.2 Path reduction for sequential models

The path reduction optimization is based on the analysis of breaking points and breaking steps, as presented in Chapter 4. A breaking step is one that contains at least one breaking point, and a breaking point is in turn an instruction which is either an external operation, an influential operation, the call to a breaking procedure or the first instruction of a process. Additionally, a `STEP` instruction may be forced to be an elementary step if it is contained in a control-flow cycle that contains no other elementary steps. All these conditions continue to be necessary in the case of a sequential NIPS model with the exception of the one that makes any external operation a breaking point.

External operations represent all forms of interaction between processes, and must be considered breaking points in a concurrent system because the interleaving order of two steps which contain these instructions and are executed concurrently can be important for the outcome of the execution. However, since a sequential system has only one process instance, then this requirement is no longer necessary as any ordering of steps is imposed by the control-flow structure of the process. Even in the case of channel operations, the single process may send and receive messages from a channel and the order of these operations cannot be affected by the merging of steps that path reduction performs.

Thus, we may define a more relaxed version of Definition 4.4 for the case of sequential NIPS programs. This definition is given in the following.

Definition 7.2 (Breaking points for sequential programs) *For a sequential NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the set of breaking points $BP^p \subseteq N$ is defined such that $n^i \in BP^p$ iff at least one of the following conditions hold:*

1. $Loc_p^i = 1$.

2. $i \in \text{Inf}_S^p$.
3. $i = \text{*CALL } a$, $a \equiv \text{Loc}_p^{i'}$ and $n^{i'}$ is a breaking procedure.

So now we consider as a breaking point the first instruction of the initial process (whose location is 1), any influential instruction, and any call to a breaking procedure. This definition represents the only modification to the original version of path reduction from Chapter 4 that is needed for the reduction of a sequential system. With this, a better reduction is obtained than if the original version were to be applied to this system.

7.3 Dead variable reduction for sequential models

As in the case of path reduction, the dead variable reduction may perform a better reduction in the case of a sequential system by applying a relaxed version of its corresponding analysis and code transformation. In this case, the main advantage is that the reduction may now operate on global variables as well as local variables. In NIPS, global and local variables are allocated in different variable arrays and thus in two disjunct storage spaces. Therefore, the analysis and reduction of both types of variables are independent, and may in fact be performed in parallel. Thus, in the case of a sequential program, both global and local dead variable analyzers are needed in order to determine every dead variable at every program location.

7.3.1 Global live variable analysis

In Chapter 5 it was explained that the reduction needed to be restricted to local variables because of the difficulties associated with the live variable analysis of variables which are shared among concurrent processes. However, in the case of a sequential system this restriction disappears, and the traditional live variable analysis from [ASU86] may be applied to global variables as well as to local variables. In this sense, the local and global variables of a sequential NIPS program may be treated in the exact same way, differing only in their location within the state of the virtual machine.

Therefore, we present the following modified versions of Definitions 5.1 and 5.2 which work with global variables instead of local variables.

Definition 7.3 (Global storage size analysis) *The global storage size analysis for a sequential NIPS program $p \in P$ with control-flow graph $G^p = \langle N, X, T, E \rangle$ consists of the solution to the data-flow equation*

$$\text{size}_G(n) = \max\{ \text{gen}(n), \text{in}(n) \},$$

with

$$\begin{aligned} \text{in}(n) &= \max_{(n',n) \in E} \text{size}_G(n'), \\ \text{gen}(n^i) &= \begin{cases} s & \text{if } i = \text{GLOBSZ } s, \\ s & \text{if } i = \text{GLOBSZX } s, \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

where $\text{size}_G(n) \in \mathbb{N}_0$ and $n \in N$.

As we may infer from this definition, the global variable array has a size of 0 when the program begins its execution, and this array is resized to a given size s by the execution of the operations `GLOBSZ` or `GLOBSZX`. Because of this, it is usual for such an instruction to be the first instruction of the initial process of a NIPS program.

Definition 7.4 (Global live variable analysis) *The global live variable analysis for a sequential NIPS program $p \in P$ with control-flow graph $G^p = \langle N, X, T, E \rangle$ consists of the solution to the data-flow equation*

$$live_G(n) = gen(n) \cup (in(n) \setminus kill(n)),$$

with

$$\begin{aligned} in(n) &= \bigcup_{(n,n') \in E} live_G(n'), \\ gen(n^i) &= \begin{cases} \{a, \dots, a + sizeof(t) - 1\} & \text{if } i = \text{LDVA } G \ t \ a \\ \{0, \dots, size_G(n) - 1\} & \text{if } i = \text{LDV } G \ t \\ \emptyset & \text{otherwise} \end{cases} \\ kill(n^i) &= \begin{cases} \{a, \dots, a + sizeof(t) - 1\} & \text{if } i = \text{STVA } G \ t \ a \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

where $live_G(n) \subseteq 2^{\mathbb{N}_0}$, $n \in N$ and $sizeof(t)$ is the size in bytes of the data type t .

The global live variable analysis corresponds to its local counterpart, replacing $size(n)$ for $size_G(n)$ and `L` for `G` as the modifier of the variable operations. With this analysis in place, we may apply the dead variable reduction to global variables.

7.3.2 Global dead variable reduction

The reduction of global dead variables is analogous to the reduction of local variables presented in Chapter 5. For this, we need to define the sets of death points and dead assignments which refer to global variables. As before, these are variants of Definitions 5.3 and 5.4.

Definition 7.5 (Death points of global variables) *Given a sequential NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of death points of global variables $DP_G^p \subseteq \mathbb{N}_0 \times N$ of p is defined such that $(v, n) \in DP_G^p$ iff $(n', n) \in E$, $v \in live_G(n')$ and $v \notin live_G(n)$.*

Definition 7.6 (Dead assignments of global variables) *Given a sequential NIPS program $p \in P$ and its control-flow graph $G^p = \langle N, X, T, E \rangle$, the set of dead assignments of global variables $DA_G^p \subseteq N$ is defined such that $n^i \in DA_G^p$ iff for $(n^i, n') \in E$ one of the following conditions hold:*

- $i = \text{STVA } G \ t \ v$ and $live_G(n') \cap \{v, \dots, v + sizeof(t) - 1\} = \emptyset$.
- $i = \text{STV } G \ t$ and $live_G(n') = \emptyset$.

Finally, we may use the results of these definitions along with those of Chapter 5 to define the code transformation associated with the dead variable reduction of a sequential NIPS program. It combines the code transformations needed to reduce according to global and local dead variables.

Definition 7.7 (Dead variable reduction for sequential NIPS programs) *Given a sequential NIPS program $p \in P$, its control-flow graph $G^p = \langle N, X, T, E \rangle$ and a specification S over AP_{NIPS} , the dead variable reduction of p produces a reduced program $p' \in P$ which is equal to p after the following transformations have been applied to it:*

- For every $i \in I^p$ where $(v, n^i) \in DP^p$ and v is not visible by S , insert c before i where

$$c = \text{LDC } 0; \text{TRUNC } 8; \text{STVA L UBYTE } v$$

- For every $i \in I^p$ where $(v, n^i) \in DP_G^p$ and v is not visible by S , insert c before i where

$$c = \text{LDC } 0; \text{TRUNC } 8; \text{STVA G UBYTE } v$$

- For every $n^i \in DA^p$ where $i = \text{STVA L } t \ v$ and v is not visible by S , replace i by c where

$$c = \text{POPX}$$

- For every $n^i \in DA_G^p$ where $i = \text{STVA G } t \ v$ and v is not visible by S , replace i by c where

$$c = \text{POPX}$$

- For every $n^i \in DA^p$ where $i = \text{STV L } t$, $\exists r \in Prs^p$, $n^i \in N_{Prs}^r$ and no variable of r is visible by S , replace i by c where

$$c = \text{POPX}; \text{POPX}$$

- For every $n^i \in DA_G^p$ where $i = \text{STV G } t$ and no global variable is visible by S , replace i by c where

$$c = \text{POPX}; \text{POPX}$$

As a result of applying this code transformation to a sequential NIPS program, every state of its induced transition system will store a default value of 0 for every variable which is statically known to be dead at the current program location of the process. This represents a better state space reduction than that which is achieved by the sole reduction of dead local variables.

7.4 Step confluence reduction and sequential models

As opposed to the two state space reductions already presented in this chapter, step confluence reduction gains no benefit when applied to a sequential program. This is due to the fact that the reduction works entirely at the level of single processes, performing a code transformation based only on the control-flow structure of each individual process, and without any consideration for concurrency. Therefore, the step confluence reduction remains unmodified when applied to a sequential program.

Chapter 8

Correctness of the Reductions

This chapter is devoted to demonstrating the correctness of the state space reductions presented in the previous chapters. Here, correctness will be equated with the preservation of equivalence under a set of specification formulas, i.e. under a temporal logic language. By determining which types of properties are preserved by a given reduction, we may decide on the usefulness of a reduction in given applications.

We assume that the programs under consideration are concurrent. The case of a sequential system is not detailed in this chapter. However, the results presented here also apply to the specific case of sequential systems, as the assumptions made for sequential systems in Chapter 7 agree with those which are made for concurrent systems in the previous chapters.

8.1 Equivalences

Table 8.1 shows the type of equivalences of state-transition systems which are preserved by the reductions, and the temporal logic languages whose semantics are also preserved [Pel05]. Obviously, the application of more than one type of reduction on a model preserves the weakest of the corresponding equivalences and the intersection of the corresponding logic languages.

The following definitions are presented according to [Pel05], [YG04] and [CGMP98].

Definition 8.1 (Simulation relation) *Given two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$, a relation $H \subseteq \Sigma \times \Sigma'$ is a simulation relation iff for every $(\sigma, \sigma') \in H$ the following hold:*

- $L(\sigma) = L'(\sigma')$.
- For every $\sigma' \rightarrow' \sigma''$ there exists $\sigma \rightarrow \sigma''$ such that $(\sigma'', \sigma'') \in H$.

Reduction	Equivalence	Temporal logic
Path reduction	Stuttering bisimulation	CTL*-X
Dead variable reduction	Bisimulation	CTL*
Step confluence reduction	Bisimulation	CTL*

Table 8.1: Equivalences and temporal logic languages preserved by the reductions.

Definition 8.2 (Bisimulation relation) *Given two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$, a relation $H \subseteq \Sigma \times \Sigma'$ is a bisimulation relation iff both H and H^{-1} are simulation relations.*

Definition 8.3 (Bisimulation equivalence) *Two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ are bisimulation equivalent, or bisimilar, denoted as $TS \sim TS'$, iff there exists a bisimulation relation H such that $(\sigma_0, \sigma'_0) \in H$.*

Definition 8.4 (Stuttering simulation relation) *Given two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$, a relation $H \subseteq \Sigma \times \Sigma'$ is a stuttering simulation relation iff for every $(\sigma, \sigma') \in H$ the following hold:*

- $L(\sigma) = L'(\sigma')$.
- For every path $\pi' = \sigma' \dots$ in TS' there exists a path $\pi = \sigma \dots$ in TS , a partition $\pi_{ST} = b_1 b_2 \dots$ and a partition $\pi'_{ST} = b'_1 b'_2 \dots$ such that for every $j \geq 1$, blocks b_j and b'_j are nonempty and finite, and for every state $t \in b_j$ and $t' \in b'_j$, it holds that $(t, t') \in H$.

Definition 8.5 (Stuttering bisimulation relation) *Given two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$, a relation $H \subseteq \Sigma \times \Sigma'$ is a stuttering bisimulation relation iff both H and H^{-1} are stuttering simulation relations.*

Definition 8.6 (Stuttering bisimulation equivalence) *Two state-transition systems $TS = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and $TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ are stuttering bisimulation equivalent, or stuttering bisimilar, denoted as $TS \sim_{ST} TS'$, iff there exists a stuttering bisimulation relation H such that $(\sigma_0, \sigma'_0) \in H$.*

8.2 Correctness of path reduction

Path reduction preserves stuttering bisimulation equivalence, and therefore also preserves the semantics of temporal logic formulas written in CTL*-X. The X (next-step) operator must be omitted because some paths of the state-transition system will be compressed so that intermediate states are eliminated, and therefore reasoning about the next state is no longer valid. However, this limitation is rarely a problem, as this operator is not required for the verification of safety or liveness properties [Lam83]. Thus, path reduction is useful in most situations.

8.2.1 Single step reduction

We begin by identifying the parts of the state-transition system that will be affected by path reduction. From Definition 4.11, we know that the code transformation associated with path reduction does only two things: it replaces some **STEP** instructions by **STEP I**, and it removes some **STEP** instructions. Therefore, when the transition system is produced from the reduced program, its difference with respect to the original transition system will be that some states will not be part of the visible state space anymore, because their

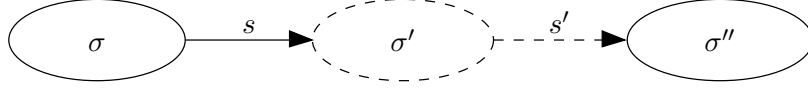


Figure 8.1: Scenario for path reduction

creation was due to at least one of the missing **STEP** instructions, or it is now an invisible state that is removed from the state space by the virtual machine.

In either case, we can identify the scenario in the state-transition system where this reduction occurs, which is depicted in Figure, 8.1. The steps s and s' are executed by the same process, and s' is not a breaking step. Therefore, in accordance to Definition 4.11, the **STEP** instruction of s is not an elementary step and is therefore subject to reduction. This causes s and s' to execute as one step, inhibits the creation of state σ' , and produces a transition $\sigma \xrightarrow{ss'} \sigma''$.

Consider a variant of path reduction which is limited to the suppression of a single **STEP** instruction. That is, one which performs the analysis of breaking points and steps as presented in Chapter 4, but that chooses just one of the non-elementary steps found and reduces it accordingly. We will call this variant *single step reduction*, and associate with it a pair of contiguous steps $\langle s, s' \rangle$ to be merged as depicted in Figure 8.1. We define this as follows:

Definition 8.7 (Single step reduction) *Given a NIPS program $p \in P$ and a specification S over AP_{NIPS} , a single step reduction of p is a tuple $\langle s, s' \rangle$ where $s, s' \in St^p$, $s' \notin BSt^p$, and there exists an elementary path $e \in EP^p$ such that $e = \dots ss' \dots$*

A single step reduction is responsible for a set of reduction scenarios like the one seen in in Figure 8.1. The following definitions formalize this relationship and the resulting transformation on the state-transition system after applying this reduction.

Definition 8.8 (Reduction scenario) *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, a specification S over AP_{NIPS} and a single step reduction $R = \langle s, s' \rangle$ of p , a tuple $\langle \sigma, \sigma', \sigma'' \rangle$ where $\sigma, \sigma', \sigma'' \in \Sigma$ is a reduction scenario of R in TS^p iff $\sigma \neq \sigma'$, $\sigma' \neq \sigma''$, $\sigma \xrightarrow{s}_q \sigma'$ and $\sigma' \xrightarrow{s'}_q \sigma''$.*

Notice that the transitions in a reduction scenario may not be looping transitions, as σ' must be different from both σ and σ'' . However, σ and σ'' could be the same state, in which case the reduction creates a looping transition that starts and ends at this state.

Definition 8.9 (Application of single step reduction) *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, a specification S over AP_{NIPS} and a single step reduction $R = \langle s, s' \rangle$ of p , the application of R to TS^p , denoted as $R(TS^p) = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$, is defined such that $\sigma'_0 = \sigma_0$, $L'(\sigma) = L(\sigma)$ for every $\sigma \in \Sigma'$, and $\Sigma' \subseteq \Sigma$ and \rightarrow' are the sets defined by the following rules:*

- $\sigma'_0 \in \Sigma'$
- If $\sigma \in \Sigma'$ and $\langle \sigma, \sigma', \sigma'' \rangle$ is a reduction scenario of R , then $\sigma \rightarrow' \sigma''$.

- If $\sigma \in \Sigma'$, $\sigma \rightarrow \sigma'$ and there exists no reduction scenario $\langle \sigma, \sigma', \sigma'' \rangle$ of R , then $\sigma \rightarrow' \sigma'$.
- If $\sigma \rightarrow' \sigma'$ then $\sigma' \in \Sigma'$.

This definition models precisely the way that the NIPS virtual machine can be used to construct a state-transition system, that is, by maintaining a set of reachable states, exploring every outgoing transition from every reachable state, and augmenting the set of reachable states with the states resulting from these transitions. However, here we also detect reduction scenarios due to the single step reduction under consideration, and create a single transition from each one found. This leaves the corresponding state σ' of each of these reduction scenarios out of the set of visible states.

Notice that in the case that a process executes the step s followed by s' and s' blocks, then this will *not* cause a reduction scenario as given by Definition 8.8. In this case, s will induce a transition by itself, which correctly models the behavior of NIPS when blocking occurs at an invisible state.

We now present a lemma that will aid us in the proof of correctness, which states that non-breaking steps may always be commuted with any other concurrent step. This is analogous to the concept of *globally independent statements* from [HP94].

Lemma 8.1 *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, a specification S over AP_{NIPS} and a step $s \in St^p$ such that $s \notin BSt^p$, then for every $\sigma, \sigma' \in \Sigma$, $s' \in St^p$ and processes q, r such that $q \neq r$, it holds: if $\sigma \xrightarrow{s}_q \sigma'$ and $\sigma \xrightarrow{s'}_r \sigma_1$, then $\sigma' \xrightarrow{s'}_r \sigma_2$ and $\sigma_1 \xrightarrow{s}_q \sigma_2$, where $\sigma_1, \sigma_2 \in \Sigma$.*

Proof We prove each of the two consequences of the lemma with the help of Figure 8.2 as an illustration. Since $s \notin BSt^p$, then its execution is internal and cannot affect, enable or disable any step which runs concurrently with it, and vice versa.

- Since s' is enabled in σ for process r , it must still be so after the execution of s by q . So it must be that $\sigma' \xrightarrow{s'}_r \sigma_2$.
- Since s is enabled for q in σ , it must also be enabled after the execution of s' by r , and therefore it must be that $\sigma_1 \xrightarrow{s}_q \sigma_3$ for some $\sigma_3 \in \Sigma$.
- From σ the following two step sequences may be executed: $s_q s'_r$ and $s'_r s_q$, which reach the respective states σ_2 and σ_3 . Given that $s \notin BSt^p$, then its execution cannot affect the concurrent execution of s' and the two step sequences must produce the same state. Therefore, $\sigma_2 = \sigma_3$.

□

The following theorem states that the application of a single step reduction preserves stuttering bisimulation equivalence.

Theorem 8.1 *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, a specification S over AP_{NIPS} and a single step reduction R of p , it holds: $TS^p \sim_{ST} R(TS^p)$.*

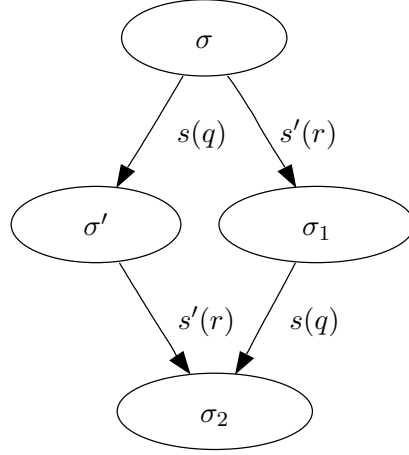


Figure 8.2: Proof of Lemma 8.1, commuting of non-breaking steps.

Proof Let $R(TS^p) = TS' = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ and $R = \langle s, s' \rangle$. We build a stuttering bisimulation relation $H \subseteq \Sigma \times \Sigma'$ as follows:

- If $\sigma \in \Sigma'$ then $(\sigma, \sigma) \in H$.
- If $\sigma_1 \xrightarrow{s'} \sigma_2$ and $\sigma_2 \in \Sigma'$ then $(\sigma_1, \sigma_2) \in H$.

From Definition 8.9 we know that $\sigma_0 = \sigma'_0$, and therefore $(\sigma_0, \sigma'_0) \in H$. We need to show that H is indeed a stuttering bisimulation relation, and for this we analyze each condition of Definition 8.4 separately:

- If $(\sigma_1, \sigma_2) \in H$, then either $\sigma_1 = \sigma_2$ or $\sigma_1 \xrightarrow{s'} \sigma_2$. For the first case it clearly holds that $L(\sigma_1) = L'(\sigma_2)$. For the second case, we know from Definition 8.7 that $s' \notin BSt^p$, and therefore s' is not influential on S and its execution does not affect state labeling, so $L(\sigma_1) = L(\sigma_2) = L'(\sigma_2)$. Thus, $L(\sigma_1) = L'(\sigma_2)$ for every $(\sigma_1, \sigma_2) \in H$, and likewise for every $(\sigma_2, \sigma_1) \in H^{-1}$.
- Let $(\sigma_1, \sigma_2) \in H$ and $\pi' = \sigma_2 \dots$ a path in TS' . Since every path in TS' originates from a path in TS^p , we relate π' to its original path where the transitions due to s and s' have not been merged. Let π'' be a path equal to π' with every occurrence of the sequence $\sigma\sigma''$ replaced by $\sigma\sigma'\sigma''$ for every reduction scenario $\langle \sigma, \sigma', \sigma'' \rangle$ of R . For every such scenario, because $\sigma' \xrightarrow{s'} \sigma''$, it holds that $(\sigma', \sigma'') \in H$ if $\sigma'' \in \Sigma'$.

- If $\sigma_1 = \sigma_2$, then $\pi = \pi''$ is a path in TS^p , and the following stuttering equivalence partition exists:

$$[\pi'|\pi] = [\sigma_2|\sigma_1] [t|t] \dots [\sigma|\sigma] [\sigma''|\sigma'\sigma''] [t|t] \dots$$

- If $\sigma_1 \xrightarrow{s'} \sigma_2$, then $\pi = \sigma_1\pi''$ is also a path in TS^p , and the following stuttering equivalence partition exists:

$$[\pi'|\pi] = [\sigma_2|\sigma_1\sigma_2] [t|t] \dots [\sigma|\sigma] [\sigma''|\sigma'\sigma''] [t|t] \dots$$

Thus, H is a stuttering simulation relation.

- Let $(\sigma_2, \sigma_1) \in H^{-1}$ and $\pi = \sigma_1 \dots$ a path in TS^p . Because of the reduction, π might not have originated a path in TS' , but in this case TS^p must contain a path which is stuttering equivalent to π and which did originate a path π' in TS' . Therefore, we can map π to π' via this intermediate path. Figure 8.3 depicts this (the shaded states correspond to π').

Let π' be a path equal to π after the following transformations have been applied to it:

- Replace every occurrence of the sequence $\sigma'\sigma''$ with σ'' iff $\sigma' \xrightarrow{s'} \sigma''$.
- Replace every occurrence of the state σ' with σ'' iff $\sigma' \xrightarrow{s'} \sigma''$ and the occurrence is not followed by σ'' .

We can verify that π' is a path in TS' . First, every occurrence of a reduction scenario of R in π is replaced by the corresponding merged transition, which agrees with Definition 8.9. However, a path in π could contain the sequence $\sigma\sigma'$ of a given reduction scenario, followed by a state σ'_1 which is not σ'' . However, by repeated applications of Lemma 8.1, we can guarantee that every state σ'_j which is reachable from σ' by a path which executes the steps $s_1 \dots s_j$ has an outgoing s' transition to a state σ''_j if every s_j is executed by a different process than the one that executes s' , and also, every σ''_j is reachable from σ'' by a path which executes the same steps $s_1 \dots s_j$. Therefore, π' may be seen as π with every reduction scenario adequately compressed, and every additional execution of s' “moved back” so that it directly follows the corresponding execution of s , and then compressed in the same way. We analyze each possible (σ_2, σ_1) :

- If $\sigma_1 = \sigma_2$, then the following stuttering equivalence partition exists:

$$[\pi|\pi'] = \begin{cases} [\sigma_1|\sigma_2][t|t] \dots [\sigma|\sigma][\sigma'\sigma''|\sigma''] [t|t] \dots \\ [\sigma_1|\sigma_2][t|t] \dots [\sigma|\sigma][\sigma'|\sigma''] [\sigma'_1|\sigma''_1] \dots [\sigma'_n|\sigma''_n][\sigma''_n][t|t] \dots \end{cases}$$

- Consider $\sigma_1 \xrightarrow{s'} \sigma_2$. If $\pi = \sigma_1\sigma_2 \dots$, then $\pi' = \sigma_2 \dots$ and the stuttering equivalence partition given for the previous case also holds for this case. Otherwise, $\pi = \sigma_1\sigma'_1 \dots$ where $\sigma'_1 \neq \sigma_2$. By Lemma 8.1, it holds that $\sigma'_1 \xrightarrow{s'} \sigma''_1$ and we have that $\pi' = \sigma_2\sigma''_1 \dots$. Thus, the stuttering equivalence partition of the previous case also holds for this case, with the following modification at its beginning:

$$[\pi|\pi'] = [\sigma_1|\sigma_2][\sigma'_1|\sigma''_1] \dots [\sigma'_n|\sigma''_n][\sigma''_n][t|t] \dots$$

Thus, H^{-1} is a stuttering simulation relation.

We may conclude that H is a stuttering bisimulation relation, and therefore, that $TS^p \sim_{ST} R(TS^p)$. \square

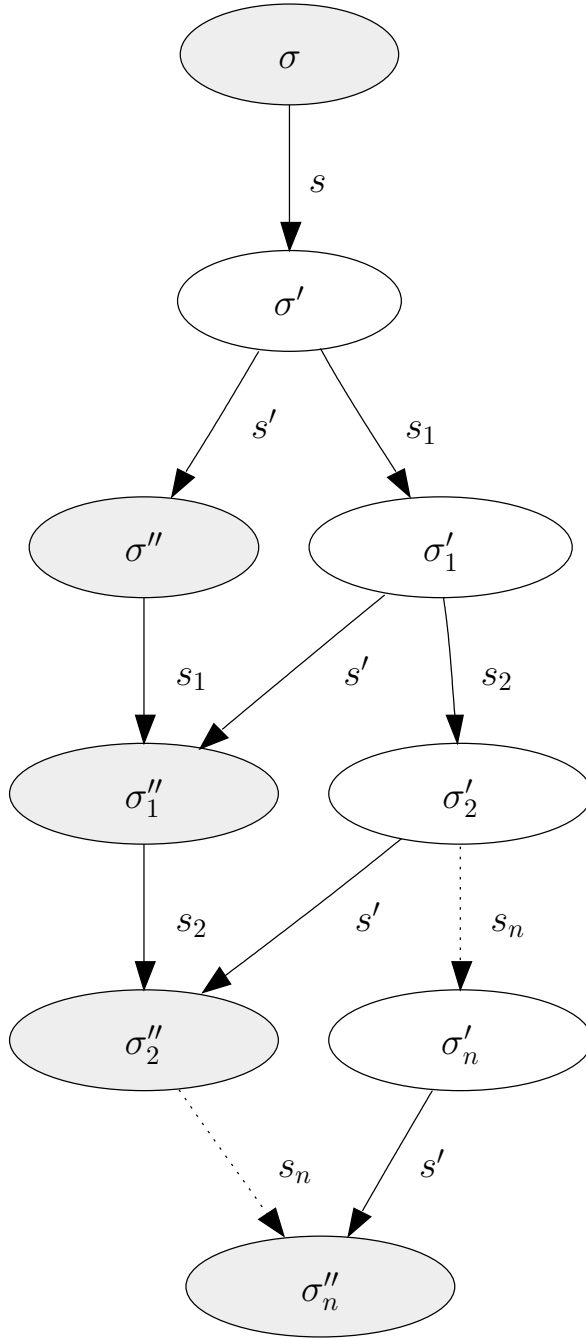


Figure 8.3: Stuttering equivalence between π and π' .

8.2.2 Multiple step reduction

Single step reductions represent the simplest code transformations which use the idea of path reduction in order to reduce the size of the transition system. Therefore, we may model path reduction as a composition of single step reductions, one for every non-elementary step which path reduction suppresses. We define this in the following.

Definition 8.10 (Multiple step reduction) *Given a NIPS program $p \in P$ and a specification S over AP_{NIPS} , the multiple step reduction R^* of p is the set defined such that $R \in R^*$ iff R is a single step reduction of p .*

The application of $R^ = \{R_1, R_2, \dots, R_n\}$ to the induced state-transition system TS^p of p is defined as follows:*

$$R^*(TS^p) = R_1(R_2(\dots R_n(TS^p)\dots)).$$

Lemma 8.2 *Given a NIPS program $p \in P$, its induced state-transition system TS^p , a specification S over AP_{NIPS} and the multiple step reduction R^* of p , then for a program p' which results from the application of path reduction to p , and its corresponding state-transition system $TS^{p'}$, it holds: $TS^{p'} = R^*(TS^p)$.*

Proof Let $h \in HS^p$ a hideable STEP instruction in of p . Then, there exists a single step reduction $R = \langle s, s' \rangle \in R^*$ where $s = \dots h$, and in $R(TS^p)$ every execution of s' has been merged with a previous execution of s by the same process. Also, in p' , h has been replaced with STEP I if $h \notin RS^p$, or has been removed from p' otherwise. Therefore, in $TS^{p'}$ every execution of s' has also been merged with a previous execution of s by the same process. Since every $h \in HS^p$ has a corresponding single step reduction $R = \langle s, s' \rangle \in R^*$, then $R^*(TS^p)$ and $TS^{p'}$ must have the same merged transitions, and therefore, must be the same transition systems. \square

Finally, we may prove that path reduction preserves stuttering bisimulation equivalence, by showing that the multiple step reduction which models path reduction preserves this equivalence.

Theorem 8.2 *Given a NIPS program $p \in P$, its induced state-transition system TS^p , a specification S over AP_{NIPS} and the multiple step reduction R^* of p , it holds: $R^*(TS^p) \sim_{ST} TS^p$.*

Proof Since $R^*(TS^p) = R_1(R_2(\dots R_n(TS^p)\dots))$, then by Theorem 8.1 it must also hold that

$$TS^p \sim_{ST} R_n(TS^p) \sim_{ST} R_{n-1}(R_n(TS^p)) \sim_{ST} \dots \sim_{ST} R_1(R_2(\dots R_n(TS^p)\dots)).$$

\square

8.3 Correctness of dead variable reduction

As seen in Table 8.1, the reduced state-transition system produced by a model with dead variable reduction is bisimilar to the full system. Thus, the reduction preserves the CTL* temporal logic as well as its subset LTL. We will show this with the help of some auxiliary definitions.

Definition 8.11 (Substitution of variable valuations) *For a state $\sigma \in \Sigma_{NIPS}$ where $v \in GVar_\sigma$, $\sigma[v/u] \in \Sigma_{NIPS}$ represents a state such that $GVal_{\sigma[v/u]}(v) = u$ and is identical to σ otherwise. Likewise, if $r \in PID_\sigma$ and $v \in LVar_\sigma^r$, then $\sigma[r, v/u] \in \Sigma_{NIPS}$ represents a state such that $LVal_{\sigma[r, v/u]}^r(v) = u$ and is identical to σ otherwise.*

This definition models the assignment of u to the global or local variable v such that its valuation is the only difference between σ and $\sigma[v/u]$, $\sigma[r, v/u]$. With the help of this notation, we may describe the effect of dead variable reduction as a state transformation. First, we define the sets of live and dead variables of a state σ .

Definition 8.12 (Live and dead variables of a state) *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, then for a state $\sigma \in \Sigma$ and a process identifier $r \in PID_\sigma$, the set $Live_\sigma^r \subset \mathbb{N}_0$ is defined such that*

$$v \in Live_\sigma^r \iff PC_\sigma(r) \equiv Loc_p^i, i \in I^p, v \in LVar_\sigma^r, v \in live(n^i).$$

Likewise, for a state $\sigma \in \Sigma$ and a process identifier $r \in PID_\sigma$, the set $Dead_\sigma^r \subset \mathbb{N}_0$ is defined such that

$$v \in Dead_\sigma^r \iff PC_\sigma(r) \equiv Loc_p^i, i \in I^p, v \in LVar_\sigma^r, v \notin live(n^i).$$

Definition 8.13 *Given a NIPS program $p \in P$ and its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, the set $\Sigma_D \subseteq \Sigma$ is defined such that $\sigma \in \Sigma_D$ iff $Dead_\sigma^r \neq \emptyset$ for some $r \in PID_\sigma$.*

Definition 8.14 (Dead variable reduction of states) *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and a specification S over AP_{NIPS} , the function $DVR_S : \Sigma_D \rightarrow \Sigma_{NIPS}$ is defined as*

$$DVR_S(\sigma) := \sigma[r_1, v_1/0] \dots [r_k, v_k/0]$$

for every $r_j \in PID_\sigma$ and $v_j \in Dead_\sigma^{r_j}$ such that S does not refer to v_j .

Theorem 8.3 *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and a specification S over AP_{NIPS} , then for a program $p' \in P$ and its induced transition-system $TS^{p'} = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ where p' is the result of applying dead variable reduction to p , the following assertions hold:*

1. $\sigma'_0 = \sigma_0$.
2. For every $\sigma \in \Sigma$ it holds:
 - $\sigma \notin \Sigma_D \Rightarrow \sigma \in \Sigma'$.

- $\sigma \in \Sigma_D \Rightarrow DVR_S(\sigma) \in \Sigma'$.

3. For every $\sigma \xrightarrow{s}_r \sigma'$ where $\sigma, \sigma' \in \Sigma$ it holds:

- $\sigma \notin \Sigma_D \wedge \sigma' \notin \Sigma_D \Rightarrow \sigma \xrightarrow{s'}_r \sigma'$.
- $\sigma \notin \Sigma_D \wedge \sigma' \in \Sigma_D \Rightarrow \sigma \xrightarrow{s'}_r DVR_S(\sigma')$.
- $\sigma \in \Sigma_D \wedge \sigma' \notin \Sigma_D \Rightarrow DVR_S(\sigma) \xrightarrow{s'}_r \sigma'$.
- $\sigma \in \Sigma_D \wedge \sigma' \in \Sigma_D \Rightarrow DVR_S(\sigma) \xrightarrow{s'}_r DVR_S(\sigma')$.

Proof

1. Since NIPS always begins execution with the same initial state, it holds that $\sigma_0 = \sigma'_0$ for every NIPS-induced transition systems TS and TS' .
2. We develop the proof by induction on the length of the path $\pi = \sigma_0 \dots \sigma$ in TS^p .

- 1: Consider $\sigma = \sigma_0 \in \Sigma$. It trivially holds that $\sigma_0 \in \Sigma'$ when $\sigma_0 \notin \Sigma_D$. Let $\sigma_0 \in \Sigma_D$. Then $\sigma_0 = DVR_S(\sigma_0)$ because $PID_{\sigma_0} = \{0\}$ and $LVal_{\sigma_0}^0(v) = 0$ for every $v \in LVar_{\sigma_0}^0$. Thus, $DVR_S(\sigma_0) \in \Sigma'$.
- $n \rightarrow n+1$: Let $\pi = \sigma_0 \dots \sigma' \sigma$ where $\sigma' \xrightarrow{s}_r \sigma$. Assume that the theorem holds for σ' . We analyze each case for σ' :

- $\sigma' \notin \Sigma_D$. Then, $\sigma' \in \Sigma'$ and there exists $\sigma' \xrightarrow{s'}_r \sigma''$ where $\sigma'' \in \Sigma'$. If $\sigma \notin \Sigma_D$, then it must be that $[s] = [s']$ because either $s = s'$ or $s \neq s'$ but produce the same valuations for all live variables, and therefore $\sigma = \sigma''$ and $\sigma \in \Sigma'$. If $\sigma \in \Sigma_D$, then s has at least one death point, $s \neq s'$ and $\sigma'' = DVR_S(\sigma)$. Thus, $DVR_S(\sigma) \in \Sigma'$.
- $\sigma' \in \Sigma_D$. Then, there exists $\sigma''' = DVR_S(\sigma')$ where $\sigma''' \in \Sigma'$, as well as $\sigma''' \xrightarrow{s'}_r \sigma''$ where $\sigma'' \in \Sigma'$. If $\sigma \notin \Sigma_D$ then $Dead_{\sigma'}^r \neq \emptyset$ and $Dead_{\sigma'}^q = \emptyset$ for every $q \neq r$ and $[s] = [s']$ for the same reason given above, and since σ' and σ''' differ at most in the valuations of dead variables, then $\sigma = \sigma''$ and therefore $\sigma \in \Sigma'$. Consider the case where $\sigma \in \Sigma_D$. If $v \in Dead_{\sigma}^q$ for some $q \neq r$, then s' does not modify the valuation of v and $LVal_{\sigma''}^q(v) = LVal_{\sigma'''}^q(v) = 0$. If $v \in Dead_{\sigma}^r$, then either $v \notin Dead_{\sigma'}^r$ and s' contains a death point for v , or $v \in Dead_{\sigma'}^r$ and s' passes the value of v from σ''' to σ'' . In either case, $LVal_{\sigma''}^q(v) = 0$ for every $q \in PID_{\sigma''}$ and $v \in Dead_{\sigma}^q$, and consequently, $\sigma'' = DVR_S(\sigma)$ and $DVR_S(\sigma) \in \Sigma'$.

3. It follows from the proof of 2.

□

Theorem 8.4 *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and specification S over AP_{NIPS} , then for a program $p' \in P$ and its induced transition-system $TS^{p'} = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ where p' is the result of applying dead variable reduction to p , it holds: $TS^p \sim TS^{p'}$.*

Proof We build a bisimulation relation $H \subseteq \Sigma \times \Sigma'$ as follows:

- $(\sigma, \sigma) \in H$ if $\sigma \notin \Sigma_D$.
- $(\sigma, \sigma') \in H$ if $\sigma \in \Sigma_D$ and $\sigma' = DVR_S(\sigma)$.

We know that $\sigma_0 = \sigma'_0$. Since the only process identifier active at σ_0 is 0, then if $Dead_{\sigma_0}^0 = \emptyset$ it follows that $\sigma_0 \notin \Sigma_D$ and $(\sigma_0, \sigma'_0) \in H$. Otherwise, $\sigma_0 = DVR_S(\sigma_0)$ because $LVal_{\sigma_0}^0(v) = 0$ for every $v \in LVar_{\sigma_0}^0$. Thus, $(\sigma_0, \sigma'_0) \in H$ as well.

We need to show that H is a bisimulation relation, which we do by showing that H and H^{-1} are simulation relations with the help of Theorem 8.3.

- Let $(\sigma, \sigma') \in H$. If $\sigma \notin \Sigma_D$ then $\sigma = \sigma'$ and it follows that $L(\sigma) = L'(\sigma')$. Otherwise, $\sigma' = DVR_S(\sigma)$ in which case σ and σ' differ at most in the valuation of non-visible dead variables, and therefore it must also hold that $L(\sigma) = L'(\sigma')$. Consequently, this also holds for every $(\sigma', \sigma) \in H^{-1}$.
- Let $(\sigma, \sigma') \in H$. Since σ and σ' differ at most in the valuations of dead variables, then for every $\sigma' \xrightarrow{s'}_r \sigma''$ there exists at least one original $\sigma \xrightarrow{s}_r \sigma''$.
 - Assume $\sigma \notin \Sigma_D$ and thus $\sigma = \sigma'$. If $\sigma'' \notin \Sigma_D$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H$. If $\sigma'' \in \Sigma_D$ then $\sigma'' = DVR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H$.
 - Assume that $\sigma \in \Sigma_D$ and thus $\sigma' = DVR_S(\sigma)$. If $\sigma'' \notin \Sigma_D$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H$. If $\sigma'' \in \Sigma_D$ then $\sigma'' = DVR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H$.

Thus, H is a simulation relation.

- Let $(\sigma', \sigma) \in H^{-1}$. Again since σ and σ' differ at most in the valuations of dead variables, then for every original $\sigma \xrightarrow{s}_r \sigma''$ there exists one $\sigma' \xrightarrow{s'}_r \sigma''$.
 - Assume $\sigma \notin \Sigma_D$ and thus $\sigma = \sigma'$. If $\sigma'' \notin \Sigma_D$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H^{-1}$. If $\sigma'' \in \Sigma_D$ then $\sigma'' = DVR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H^{-1}$.
 - Assume that $\sigma \in \Sigma_D$ and thus $\sigma' = DVR_S(\sigma)$. If $\sigma'' \notin \Sigma_D$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H^{-1}$. If $\sigma'' \in \Sigma_D$ then $\sigma'' = DVR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H^{-1}$.

Thus, H^{-1} is a simulation relation.

We may conclude that H is a bisimulation relation and therefore $TS^p \sim TS^{p'}$. \square

8.4 Correctness of step confluence reduction

As in the case of dead variable reduction, the transformation due to step confluence reduction preserves bisimulation equivalence, and hence the validity of formulas expressed in CTL*. Since program locations are *fields* of a state just like variables are, the proof of this equivalence is very similar to the corresponding proof for dead variable reduction.

Definition 8.15 (Substitution of enabled locations) For a state $\sigma \in \Sigma_{NIPS}$ where $r \in PID_\sigma$ and $l \equiv PC_\sigma(r)$, $\sigma[r, l/l'] \in \Sigma_{NIPS}$ represents a state where $l' \equiv PC_{\sigma[r, l/l']}(r)$ and is identical to σ otherwise.

This definition models the *relocation* of the process r from l to l' , that is, an *instant jump* of r from the program location l to the program location l' , such that no other part of the state is modified. With the help of this notation we may define the effect of step confluence reduction on states.

Definition 8.16 Given a NIPS program $p \in P$ and its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$, the set $\Sigma_C \subseteq \Sigma$ is defined such that $\sigma \in \Sigma_C$ iff $PC_\sigma(r) \equiv Loc_p^i$, $(n^{i'}, n^i) \in E$ and $i'' = CR_S^p(i')$ for some $r \in PID_\sigma$, $i, i', i'' \in IP$.

The set Σ_C contains to all those states in Σ that have at least one enabled program location which is preceded by a confluent STEP instruction that is not a representative for its corresponding confluence set. Thus, this location is affected by the state space reduction and will not appear as enabled in the reduced transition system.

Definition 8.17 (Step confluence reduction of states) Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and a specification S over AP_{NIPS} , the function $SCR_S : \Sigma_C \rightarrow \Sigma \setminus \Sigma_C$ is defined as

$$SCR_S(\sigma) := \sigma[r_1, l_1/l'_1] \dots [r_k, l_k/l'_k]$$

for every $r_j \in PID_\sigma$, $Loc_p^{i_j} = l_j$ and $Loc_p^{i'_j} = l'_j$ such that $(n^{i''_j}, n^{i_j}) \in E$, $(n^{i''_j}, n^{i'_j}) \in E$ and $i''_j = CR_S^p(i'_j)$.

The mapping SCR_S models the state space reduction due to step confluence reduction. Every state $\sigma \in \Sigma_C$ is mapped to an equivalent state where all enabled program locations have been substituted by their confluent representatives, and is undefined for every state $\sigma \in \Sigma \setminus \Sigma_C$, because by Definition 8.16 the state σ has no enabled locations which are confluent.

Theorem 8.5 Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and a specification S over AP_{NIPS} , then for a program $p' \in P$ and its induced transition-system $TS^{p'} = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ where p' is the result of applying step confluence reduction to p , the following assertions hold:

1. $\sigma'_0 = \sigma_0$.
2. For every $\sigma \in \Sigma$ it holds:
 - $\sigma \notin \Sigma_C \Rightarrow \sigma \in \Sigma'$.
 - $\sigma \in \Sigma_C \Rightarrow SCR_S(\sigma) \in \Sigma'$.
3. For every $\sigma \xrightarrow{s}_r \sigma'$ where $\sigma, \sigma' \in \Sigma$ it holds:
 - $\sigma \notin \Sigma_C \wedge \sigma' \notin \Sigma_C \Rightarrow \sigma \xrightarrow{s'}_{r'} \sigma'$.

- $\sigma \notin \Sigma_C \wedge \sigma' \in \Sigma_C \Rightarrow \sigma \xrightarrow{s'}_r SCR_S(\sigma')$.
- $\sigma \in \Sigma_C \wedge \sigma' \notin \Sigma_C \Rightarrow SCR_S(\sigma) \xrightarrow{s'}_r \sigma'$.
- $\sigma \in \Sigma_C \wedge \sigma' \in \Sigma_C \Rightarrow SCR_S(\sigma) \xrightarrow{s'}_r SCR_S(\sigma')$.

Proof

1. Already proved for point 1 of Theorem 8.3.
2. As for Theorem 8.3, we develop the proof by induction on the length of the path $\pi = \sigma_0 \dots \sigma$ in TS^p .
 - 1: Consider $\sigma = \sigma_0 \in \Sigma$. Since the initial location of the initial process is not preceded by any STEP instruction, then it trivially holds that $\sigma_0 \notin \Sigma_C$ as well as $\sigma_0 \in \Sigma'$.
 - $n \rightarrow n+1$: Let $\pi = \sigma_0 \dots \sigma' \sigma$ where $\sigma' \xrightarrow{s}_r \sigma$. Assume that the theorem holds for σ' . We analyze each case for σ' :
 - $\sigma' \notin \Sigma_C$. Then, $\sigma' \in \Sigma'$ and there exists $\sigma'' \xrightarrow{s'}_r \sigma''$ where $\sigma'' \in \Sigma'$. If $\sigma \notin \Sigma_C$, then it must be that $s' = s$ and therefore $\sigma = \sigma''$ and $\sigma \in \Sigma'$. If $\sigma \in \Sigma_C$, then s precedes a confluent step and s' corresponds to s with its STEP instruction modified by a jump to its representative. Also, $PC_\sigma(q) = PC_{\sigma''}(q)$ for every $q \in PID_\sigma$, $q \neq r$. Therefore, $\sigma'' = SCR_S(\sigma)$ and also $SCR_S(\sigma) \in \Sigma'$.
 - $\sigma' \in \Sigma_C$. Then, there exists $\sigma''' = SCR_S(\sigma')$ where $\sigma''' \in \Sigma'$, as well as $\sigma''' \xrightarrow{s'}_r \sigma''$ where $\sigma'' \in \Sigma'$. Also, s and s' are either confluent steps if $PC_{\sigma'}(r) \neq PC_{\sigma'''}(r)$ or equal if $PC_{\sigma'}(r) = PC_{\sigma'''}(r)$. If $\sigma \notin \Sigma_C$ then s and s' end at the same STEP instruction and $PC_\sigma(q) = PC_{\sigma''}(q)$ for every $q \in PID_\sigma$. Therefore, $\sigma = \sigma''$ and $\sigma \in \Sigma'$. Consider the case where $\sigma \in \Sigma_C$. If $PC_\sigma(r) = PC_{\sigma''}(r)$ then s and s' end at the same program location l which was not subject to the reduction, in which case it must be that $PC_\sigma(q) \neq PC_{\sigma''}(q)$ and $PC_{\sigma'}(q) \neq PC_{\sigma'''}(q)$ for some $q \neq r$. If $PC_\sigma(r) \neq PC_{\sigma''}(r)$ then s and s' end at different locations, which means that s precedes a confluent step and s' corresponds to s with its STEP instruction modified by a jump to its representative. In both cases, $\sigma'' = SCR_S(\sigma)$ and thus $SCR_S(\sigma) \in \Sigma'$.
3. It follows from the proof of 2.

□

Theorem 8.6 *Given a NIPS program $p \in P$, its induced state-transition system $TS^p = \langle \Sigma, \sigma_0, \rightarrow, L \rangle$ and specification S over AP_{NIPS} , then for a program $p' \in P$ and its induced transition-system $TS^{p'} = \langle \Sigma', \sigma'_0, \rightarrow', L' \rangle$ where p' is the result of applying step confluence reduction to p , it holds: $TS^p \sim TS^{p'}$.*

Proof We build a bisimulation relation $H \subseteq \Sigma \times \Sigma'$ in the same manner as for Theorem 8.4:

- $(\sigma, \sigma) \in H$ if $\sigma \notin \Sigma_C$.
- $(\sigma, \sigma') \in H$ if $\sigma \in \Sigma_C$ and $\sigma' = SCR_S(\sigma)$.

We know that $\sigma_0 = \sigma'_0$ and that $\sigma_0 \notin \Sigma_C$. Thus it holds that $(\sigma_0, \sigma'_0) \in H$.

We need to show that H is a bisimulation relation, which we do by showing that H and H^{-1} are simulation relations with the help of Theorem 8.5.

- Let $(\sigma, \sigma') \in H$. If $\sigma \notin \Sigma_C$ then $\sigma = \sigma'$ and it follows that $L(\sigma) = L'(\sigma')$. Otherwise, $\sigma' = SCR_S(\sigma)$ in which case σ and σ' differ only in the values of non-visible program locations, and therefore it must also hold that $L(\sigma) = L'(\sigma')$. Consequently, this also holds for every $(\sigma', \sigma) \in H^{-1}$.
- Let $(\sigma, \sigma') \in H$. For every $\sigma' \xrightarrow{s'}_r \sigma''$ there exists at least one original $\sigma \xrightarrow{s}_r \sigma''$ such that s and s' are either confluent or equal.
 - Assume $\sigma \notin \Sigma_C$ and thus $\sigma = \sigma'$. If $\sigma'' \notin \Sigma_C$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H$. If $\sigma'' \in \Sigma_C$ then $\sigma'' = SCR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H$.
 - Assume that $\sigma \in \Sigma_C$ and thus $\sigma' = SCR_S(\sigma)$. If $\sigma'' \notin \Sigma_C$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H$. If $\sigma'' \in \Sigma_C$ then $\sigma'' = SCR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H$.

Thus, H is a simulation relation.

- Let $(\sigma', \sigma) \in H^{-1}$. Again, for every original $\sigma \xrightarrow{s}_r \sigma''$ there exists one $\sigma' \xrightarrow{s'}_r \sigma''$ such that s and s' are either confluent or equal.
 - Assume $\sigma \notin \Sigma_C$ and thus $\sigma = \sigma'$. If $\sigma'' \notin \Sigma_C$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H^{-1}$. If $\sigma'' \in \Sigma_C$ then $\sigma'' = SCR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H^{-1}$.
 - Assume that $\sigma \in \Sigma_C$ and thus $\sigma' = SCR_S(\sigma)$. If $\sigma'' \notin \Sigma_C$ then $\sigma'' = \sigma''$ and $(\sigma'', \sigma'') \in H^{-1}$. If $\sigma'' \in \Sigma_C$ then $\sigma'' = SCR_S(\sigma'')$, and also $(\sigma'', \sigma'') \in H^{-1}$.

Thus, H^{-1} is a simulation relation.

We may conclude that H is a bisimulation relation and therefore $TS^p \sim TS^{p'}$. \square

Chapter 9

SARN: Static Analysis and Reduction for NIPS

SARN, which stands for “Static Analysis and Reduction for NIPS”, is an implementation of the static analyzes and state space reduction optimizations that have been presented in this work. It is also a library which may be used in the development of new state space reductions, and more generally, in any software tool that needs to analyze and manipulate NIPS byte-code. This chapter presents SARN along with its implementation details.

9.1 The SARN class library

SARN is both a library and a set of command-line tools for the application of the state space reductions to NIPS byte-code files. It is written in the JavaTM programming language, and is therefore structured as a class library. We outline its most important classes in the following (all classes reside in the `sarn` package):

- **Bytecode:** Represents the byte-code operations of NIPS. One instance of this class is created for each combination of byte-code operation, valid modifiers and parameter types. It also offers methods to identify and classify byte-code operations.
- **Instruction:** Represents an instruction of the virtual machine, that is, a configuration of a `Bytecode` and values for each of its corresponding parameters.
- **BytecodeFile:** Represents the file format of the NIPS byte-code. It offers an object model for the manipulation of every section of a byte-code file, including the sequence of instructions.
- **BytecodeReader:** Implements the algorithm that reads the contents of a byte-code file and produces the corresponding instance of the `BytecodeFile` class. It also offers a command that displays the contents of a given byte-code file.
- **BytecodeWriter:** Implements the algorithm that takes a `BytecodeFile` instance and writes the corresponding byte-code file.

- **Step**: A high-level representation of an instruction. It uses object references instead of numeric addresses, and additionally maintains source-code-level information and attributes of the instruction.
- **Model**: A high-level model of a byte-code program. It uses instances of the **Step** class and therefore uses object references instead of numeric addresses. It also maintains information about the processes and procedures contained in the program, and implements the conversion between **Model** and **BytecodeFile** instances.
- **ControlFlowGraph**: Represents a control-flow graph of a NIPS program. It creates such graphs from instances of the **Model** class, and provides methods for traversal of the graph and iteration over paths and cycles in the graph, using the *visitor* design pattern. It also offers a command that displays the control-flow graph as a textual or graphical representation.
- **Specification**: Represents a temporal logic specification. It abstracts away from any specific temporal logic language, and only maintains information which is of importance for the state space reductions, such as the sets of visible local and global variables and visible program locations.
- **PathReduction**: Implements the path reduction optimization as a command that receives a byte-code file and produces a reduced version of it. It may also be used as a component within a host application.
- **DataFlowAnalysis**: Implements the iterative solution to the data-flow problem for single processes.
- **VariableAnalysis**: Implements the solution to the storage size analysis and to the live variable analysis, using the **DataFlowAnalysis** class for the computation of the solution.
- **DeadVariableReduction**: Implements the dead variable reduction optimization as a byte-code transformation command. It may also be used as a component within a host application.
- **VariableLocator**: Maintains a mapping of variable names to variable offsets in the variable array. These variable names correspond to the variables of a high-level program that has been compiled to the byte-code, and are stored in the byte-code file as additional information.
- **StepConfluenceReduction**: Implements the step confluence reduction optimization as a byte-code transformation command, which may also be used as a component within a host application.
- **CodeReachability**: Analyzes the reachability of byte-code instructions within a program and removes unreachable code from it.
- **ConcurrencyAnalysis**: Implements the detection of sequential and concurrent processes, and instructs the reductions on which version of the reduction to apply.

Additionally, the following support classes are also included in NIPS:

- **Command**: Represents a command which may be invoked from a command shell, and manages options, parameters and error handling.
- **Util**: Manages output messages and verbosity levels.

The SARN class library is contained in a JAR file called `sarn.jar`. This file is created when SARN is compiled from its sources.

9.2 SARN commands

SARN provides the following commands that apply the state space reductions to given byte-code files, and also display information about these files. Since they are presented in the form of executable JavaTM classes, they may be invoked with the command

```
java -cp sarn.jar sarn.<name>
```

where `<name>` is substituted with the name of the corresponding class.

Common options

All commands in SARN admit the following options:

- **-verbose**: Causes the command to write informational messages to the console. The command is completely *silent* otherwise.
- **-debug**: Causes the command to write debugging messages to the console, and implies **-verbose**.
- **-help**: Shows information about the usage of the command and exits.

BytecodeReader

This command displays the contents of a given byte-code file. It lists all instructions in the file along with their numerical addresses, as well as other parts of the file like the strings table, the source locations table and the structure information table. It takes the name of the byte-code file as its only argument.

ControlFlowGraph

This command creates the control-flow graph for the given byte-code file, and displays this graph in one or more ways, depending on the options that are given. The options that may be given to this command are:

- **-print**: Shows each node of the control-flow graph, followed by each of its successor nodes.

- **-paths**: Prints every control-flow path of the program as a list of instructions. Each path ends either at a **STEP T** instruction or when the path closes a loop. A path may also end at a **RET** instruction when it belongs to a procedure.
- **-dot=<value>**: Produces a graphical representation of the control-flow graph of the program in Graphviz *dot* format. The name of the file where the graph is to be saved is given in place of **<value>**.

ControlFlowGraph takes the name of the byte-code file as its only argument.

PathReduction

This command applies the path reduction optimization to a given byte-code file. It takes two arguments: the name of the byte-code file to reduce, and the name of the reduced byte-code file. It also admits the following options in order to guide the reduction:

- **-no-remove**: Causes the command to apply path reduction without step removal, therefore replacing non-elementary steps by **STEP I**.
- **-no-loop-check**: Disables the execution of the safety loop check, causing the reduction to ignore cycles and to avoid the introduction of forced elementary steps.
- **-visible=<value>**: Causes the reduction to assume that the given variables are visible by the specification. The string **<value>** corresponds to a list of variable names, separated by commas. Global variables correspond to their source code names, and local variables use the notation **process.variable**, where **process** is the source code name of the corresponding process type and **variable** is the source code name of the local variable.
- **-visible-channels**: Causes the reduction to assume that the contents of all channels is visible by the specification.
- **-visible-globals**: Causes the reduction to assume that every global variable is visible by the specification.
- **-visible-locals**: Causes the reduction to assume that every local variable of every process is visible by the specification.

NIPS additionally supports LTL model checking by the use of a *monitor* process, which is a special process that runs in parallel with the system and represents the LTL formula in the form of a Büchi automaton. As such, any local variable or process location that is visible by this monitor process must be regarded as visible by the specification. This command detects this and honors the visibility of variables and locations which are referred to by the monitor, without the need to supply any option.

This implementation of path reduction does not remove non-elementary steps, but rather replaces these steps by a **NOP** instruction. Although both code transformations are equivalent, the latter produces code where the original location of every removed **STEP** instruction is still visible when inspecting the byte-code, and has been very convenient during the development phase of this work.

DeadVariableReduction

The dead variable reduction optimization is applied by this command. Like the previous command, it takes two arguments: the name of the original byte-code file and the name of the reduced byte-code file. Additionally, it admits the options `-visible=<value>`, `-visible-globals` and `-visible-locals`, which behave in exactly the same manner as for `PathReduction`. The last two options are useful in the case of a sequential system, because otherwise, assuming all global variables as visible has no effect on a purely local reduction, and assuming all local variables as visible completely inhibits this reduction. This command also considers the variables referred to by an existing monitor process as visible variables.

StepConfluenceReduction

This command applies the step confluence reduction to a given byte-code file. Like the two previous commands, it also takes two arguments: the name of the original byte-code file and the name of the reduced byte-code file. This command admits no options other than the default options, and considers any location visible by an existing monitor process as a visible location.

Chapter 10

Experimental Results and Evaluation

The state space reduction techniques that have been presented in this work achieve significant results in practice, and this chapter demonstrates this by presenting some experimental results that have been obtained from the application of these reductions, as well as by evaluating these results.

10.1 Experimental procedure

The experiments that have been carried out consist of using the NIPS virtual machine to generate the complete state-transition system of a given model using a breadth-first search of the state space. For this, the sample application `nips_vm` included with the NIPS distribution is used. Upon conclusion, this application displays some statistics from the generated state-transition system and the state space search that was performed, such as the total number of states and transitions encountered, as well as the total amount of memory that was required for the storage of states. We therefore extract this information for our results. Also, the execution time of the application is measured. By using SARN to apply one or more of the reductions to a given byte-code file, we may compare and contrast the results of running the unreduced and reduced models in order to quantify the effects of the reductions.

If the NIPS model that is being used for experimentation has been compiled from a model written in some other language, we may also compare the results obtained by other implementations of the language and those obtained by NIPS and its reductions. Since NIPS includes a compiler from the Promela language to NIPS, we may therefore use models written in Promela for our experiments, and compare the results of NIPS with the results of performing a similar state space search using Spin. This is accomplished by compiling the model with the `spin` command and producing a C program called a *protocol analyzer*. This program may then be compiled and run in order to perform the state space exploration. The statistics which are given by this tool are also similar to those given by NIPS, so we are able to extract them in the same way. Additionally, a compiler from the low-level assembly language of the Atmel ATmega16 embedded micro controller to the NIPS byte-code language is currently being developed [Roh06]. By using this compiler to translate embedded programs for this micro controller to the byte-code

of the NIPS virtual machine, it is possible to explore their state spaces using NIPS. Thus, we may also use the models of these embedded programs to evaluate the effects of the reductions.

10.2 Spin examples

The Spin model checker is distributed with a set of examples which show several features of Spin and the Promela language. A subset of these examples have been used to evaluate the state space reductions presented in this work. Each example has been compiled to NIPS first. Then, each of the state space reductions has been applied separately in order to measure its individual effect. Finally, all three reductions have been applied in a sequence in order to measure their combined effect.

The examples have also been evaluated by running them under Spin. In this case, two different configurations have been used: one in which partial order reduction and additional data-flow optimizations have been deactivated, and one in which all of these reductions are applied.

To run these experiments, a machine with an AMD Athlon™ 64 Processor 3500+ and with 2 GB of memory was used. The results of these experiments are shown in Table 10.1 and Table 10.2. They present the measured values for the number of states and transitions created, the amount of memory used for the storage of states, and the total elapsed time taken by the execution of the application. Also, the same results are shown expressed as percentages with respect to the results of the corresponding unreduced NIPS model. In two cases, these percentages could not be calculated because NIPS reported 0 MB of memory usage when a small number of total states was produced. The configuration of each model is encoded in its file name as follows: first, the name of the model is used, followed by the names of every reduction technique applied, and then followed by the ending `.b` when the model corresponds to a NIPS byte-code file, and `.spin` when it corresponds to a protocol analyzer produced by Spin. We designate path reduction with `pr`, dead variable reduction with `dvr`, step confluence reduction with `scr`, partial order reduction with `por` and the data-flow optimizations of Spin with `opt`.

The data that has been gathered measures two different but related aspects: the size of the produced state-transition system, and the time required for its construction. We will study both aspects in detail. We may first notice that there exists a difference in the results that both implementations produce. On average, Spin generated state spaces which are 20% smaller than the state spaces generated by NIPS, when no reductions are applied in both cases. This difference is due to the way in which Spin counts and stores the generated states. The Promela language provides a way to designate a block of program statements as *atomic*, which basically means that the corresponding process executes this block without allowing other processes to execute until the last statement in the block has been executed. Spin creates the intermediate states but does not count or store them; NIPS provides the same functionality but counts these states as normal states, and thus obtains a higher state count. Also, the memory usage of Spin was approximately 42% less than that of NIPS, and it also required 58% less computation time. The first is due to the fact that Spin performs state compression whereas NIPS stores the actual states in memory. The fact that Spin is faster than NIPS is not surprising, considering that NIPS

Model	States		Transitions		State Memory		Elapsed Time	
	Total	%	Total	%	MB	%	sec.	%
eratosthenes.b	56744	100.00	217013	100.00	12.15	100.00	1.34	100.00
eratosthenes+pr.b	979	1.73	2344	1.08	0.22	1.81	0.08	5.97
eratosthenes+dvr.b	56744	100.00	217013	100.00	12.15	100.00	1.58	117.91
eratosthenes+scr.b	56744	100.00	217013	100.00	12.15	100.00	1.34	100.00
eratosthenes+pr+dvr+scr.b	979	1.73	2344	1.08	0.22	1.81	0.09	6.72
eratosthenes.spin	47669	84.01	177716	81.89	5.84	48.02	0.52	38.81
eratosthenes+por+opt.spin	2093	3.69	2571	1.18	0.51	4.17	0.02	1.49
leader2.b	5950945	100.00	23856364	100.00	1378.46	100.00	368.84	100.00
leader2+pr.b	202639	3.41	632237	2.65	46.85	3.40	4.95	1.34
leader2+dvr.b	5937736	99.78	23817984	99.84	1375.77	99.80	324.15	87.88
leader2+scr.b	5950945	100.00	23856364	100.00	1378.46	100.00	376.26	102.01
leader2+pr+dvr+scr.b	202519	3.40	632117	2.65	46.83	3.40	5.49	1.49
leader2.spin	5934070	99.72	23809900	99.81	1094.13	79.37	90.24	24.47
leader2+por+opt.spin	14122	0.24	14241	0.06	2.86	0.21	0.06	0.02
leader.b	46091	100.00	185523	100.00	10.63	100.00	0.61	100.00
leader+pr.b	1505	3.27	4799	2.59	0.35	3.29	0.07	11.48
leader+dvr.b	46091	100.00	185523	100.00	10.63	100.00	0.66	108.20
leader+scr.b	46091	100.00	185523	100.00	10.63	100.00	0.61	100.00
leader+pr+dvr+scr.b	1505	3.27	4799	2.59	0.35	3.29	0.07	11.48
leader.spin	45919	99.63	185066	99.75	8.40	79.03	0.59	96.72
leader+por+opt.spin	97	0.21	97	0.05	4.88	45.90	0.01	1.64
loops.b	28	100.00	34	100.00	0.00	-	0.04	100.00
loops+pr.b	12	42.86	18	52.94	0.00	-	0.04	100.00
loops+dvr.b	28	100.00	34	100.00	0.00	-	0.04	100.00
loops+scr.b	28	100.00	34	100.00	0.00	-	0.04	100.00
loops+pr+dvr+scr.b	12	42.86	18	52.94	0.00	-	0.04	100.00
loops.spin	27	96.43	34	100.00	4.88	-	0.01	25.00
loops+por+opt.spin	25	89.29	31	91.18	4.88	-	0.01	25.00
mobile1.b	38597	100.00	117957	100.00	5.89	100.00	0.81	100.00
mobile1+pr.b	11183	28.97	27549	23.36	1.70	28.86	0.31	38.27
mobile1+dvr.b	23003	59.60	72429	61.40	3.51	59.59	0.54	66.67
mobile1+scr.b	38597	100.00	117957	100.00	5.89	100.00	0.81	100.00
mobile1+pr+dvr+scr.b	5267	13.65	13875	11.76	0.80	13.58	0.18	22.22
mobile1.spin	32668	84.64	98268	83.31	2.86	48.51	0.29	35.80
mobile1+por+opt.spin	9971	25.83	20246	17.16	1.01	17.22	0.08	9.88
mobile2.b	12818	100.00	39227	100.00	1.96	100.00	0.31	100.00
mobile2+pr.b	3861	30.12	9441	24.07	0.59	30.10	0.13	41.94
mobile2+dvr.b	7620	59.45	24051	61.31	1.16	59.18	0.21	67.74
mobile2+scr.b	12818	100.00	39227	100.00	1.96	100.00	0.31	100.00
mobile2+pr+dvr+scr.b	1841	14.36	4787	12.20	0.28	14.29	0.09	29.03
mobile2.spin	10865	84.76	32688	83.33	0.91	46.53	0.09	29.03
mobile2+por+opt.spin	3301	25.75	6531	16.65	0.50	25.66	0.03	9.68

Table 10.1: Results of running the Spin examples (1).

Model	States		Transitions		State Memory		Elapsed Time	
	Total	%	Total	%	MB	%	sec.	%
peterson_N.b	51118	100.00	143231	100.00	2.10	100.00	0.27	100.00
peterson_N+pr.b	2273	4.45	5042	3.52	0.09	4.29	0.06	22.22
peterson_N+dvr.b	37436	73.23	103676	72.38	1.54	73.33	0.22	81.48
peterson_N+scr.b	51118	100.00	143231	100.00	2.10	100.00	0.27	100.00
peterson_N+pr+dvr+scr.b	1945	3.80	4314	3.01	0.08	3.81	0.06	22.22
peterson_N.spin	45916	89.82	128655	89.82	2.45	116.86	0.15	55.56
peterson_N+por+opt.spin	3000	5.87	3806	2.66	4.98	237.24	0.02	7.41
peterson.b	75	100.00	135	100.00	0.00	-	0.04	100.00
peterson+pr.b	73	97.33	133	98.52	0.00	-	0.04	100.00
peterson+dvr.b	75	100.00	135	100.00	0.00	-	0.04	100.00
peterson+scr.b	58	77.33	101	74.81	0.00	-	0.04	100.00
peterson+pr+dvr+scr.b	56	74.67	99	73.33	0.00	-	0.04	100.00
peterson.spin	56	74.67	100	74.07	4.88	-	0.01	25.00
peterson+por+opt.spin	41	54.67	68	50.37	4.88	-	0.01	25.00
pftp.b	1378184	100.00	2365684	100.00	249.72	100.00	9.39	100.00
pftp+pr.b	301623	21.89	684474	28.93	54.65	21.88	4.30	45.79
pftp+dvr.b	1354561	98.29	2342061	99.00	245.44	98.29	9.91	105.54
pftp+scr.b	1376973	99.91	2364206	99.94	249.50	99.91	9.39	100.00
pftp+pr+dvr+scr.b	290207	21.06	672791	28.44	52.58	21.06	4.56	48.56
pftp.spin	439895	31.92	1301620	55.02	56.61	22.67	3.44	36.63
pftp+por+opt.spin	47356	3.44	64970	2.75	6.33	2.54	0.22	2.34
snoopy.b	124434	100.00	414047	100.00	24.45	100.00	4.11	100.00
snoopy+pr.b	68658	55.18	230483	55.67	13.49	55.17	2.95	71.78
snoopy+dvr.b	94364	75.83	319895	77.26	18.54	75.83	3.88	94.40
snoopy+scr.b	109960	88.37	370045	89.37	21.60	88.34	3.77	91.73
snoopy+pr+dvr+scr.b	46630	37.47	160215	38.69	9.16	37.46	2.38	57.91
snoopy.spin	91920	73.87	305460	73.77	9.81	40.11	0.81	19.71
snoopy+por+opt.spin	9707	7.80	13888	3.35	1.20	4.90	0.04	0.97
sort.b	1078855	100.00	5853039	100.00	248.75	100.00	14.56	100.00
sort+pr.b	7170	0.66	29622	0.51	1.66	0.67	0.18	1.24
sort+dvr.b	1078855	100.00	5853039	100.00	248.75	100.00	15.86	108.93
sort+scr.b	1078855	100.00	5853039	100.00	248.75	100.00	14.56	100.00
sort+pr+dvr+scr.b	7170	0.66	29622	0.51	1.66	0.67	0.19	1.30
sort.spin	659683	61.15	3454990	59.03	111.19	44.70	10.29	70.67
sort+por+opt.spin	135	0.01	135	0.00	4.88	1.96	0.01	0.07

Table 10.2: Results of running the Spin examples (2).

is a virtual machine that interprets byte-code while Spin creates a C program which runs natively. Therefore, we should keep these differences in mind when evaluating the effects of the reductions.

The main objective of the state space reduction techniques is the reduction of the size of the corresponding transition system, which is in direct relation to the number of reachable states. The number of transitions and the amount of memory used are usually proportional to this measure, so we will concentrate on analyzing just the number of reachable states for each experiment.

In the case of path reduction, the state space was reduced by 74% on average, yielding state spaces which range between 0.66% and 97.33% of the original. Excluding the case of the `peterson` model, roughly half of the state space or more is discarded when path reduction is applied in every case. This great reduction supports what has already been mentioned in Chapter 4 about the exponential effect on the state space caused by variations in the step granularity of concurrent processes that are modeled by interleaving semantics. The `peterson` model is a very small and simple model which does not gain much from path reduction as its step granularity is almost fully “reduced” already in its original form. The rest of the models have a significant gain when path reduction is applied to them.

In the case of dead variable reduction, the state space was reduced by 12% on average, yielding state spaces which range between 59.45% and 100% of the original. The effect of dead variable reduction is notably less than that of path reduction, but nevertheless significant in models like `mobile1` and `mobile2` where it reduces the state space by nearly 40%. The actual effect of dead variable reduction depends on the use of local variables by the processes of the concurrent system, and therefore causes no reduction whatsoever in some models.

The step confluence reduction achieved a state space reduction of 3% on average while yielding states spaces which range between 77.33% and 100% of the original. This technique is in general less powerful than the two other techniques considered. However, notice the case of the `peterson` model where step confluence reduction obtains the best individual reduction. Also, its reduction with respect to the `pftp` model is very close to that of dead variable reduction.

When all reductions were combined, the state space was reduced by 80% on average, yielding state spaces which range between 0.66% and 74.67% of the original. This amounts to a minimum reduction of 25% on any tested model, and if we again exclude the `peterson` model, nearly 60% of the state space or more is discarded on each model when all the reductions are applied together.

In relation to the results obtained from the protocol analyzers generated by Spin, the combined reductions on NIPS generate absolute smaller state spaces than those of Spin with partial order reduction and data-flow optimizations in 5 out of 11 models. This is a very positive result, considering what was mentioned earlier about the base difference between the state space sizes produced by NIPS and Spin. It also shows that the great reduction achieved by path reduction is in some cases comparable and even superior to the reduction obtained by partial order reduction. The smaller state space size produced by Spin in the remaining models is due to this base difference, together with the additional reduction performed by partial order reduction with respect to the interleaving of independent actions.

Model	States		Transitions		State Memory		Elapsed Time	
	Total	%	Total	%	MB	%	sec.	%
test_loop.b	330009	100.00	330008	100.00	356.89	100.00	4.01	100.00
test_loop+pr.b	60002	18.18	60001	18.18	64.89	18.18	2.08	51.87
test_loop+dvr.b	330009	100.00	330008	100.00	356.89	100.00	4.20	104.74
test_loop+scr.b	330009	100.00	330008	100.00	356.89	100.00	4.01	100.00
test_loop+pr+dvr+scr.b	60002	18.18	60001	18.18	64.89	18.18	2.09	52.12

Table 10.3: Results of running a sequential model of an embedded program.

Considering computational time, we may see that it is more or less proportional to the size of the generated state space, with an added constant factor due to the interpretation of the byte-code, whose size is modified only minimally by the reductions. We may also note that dead variable reduction sometimes causes the time to be *increased* with respect to the original model. This is also not surprising, as the code transformation associated with dead variable reduction adds a total of 3 byte-code instructions for every death point encountered, and this should cause an increase in the computational time due to the execution of the modified steps. When an actual state space reduction is obtained, this normally compensates the extra time due to execution and causes a balance in favor of the reduction. However, when no reduction is obtained, the result is unfavorable because the same transition system is created in a somewhat slower manner. As before, Spin obtains shorter execution times than NIPS, even when producing larger state spaces.

10.3 An embedded program

The second performance test that we will analyze corresponds to an embedded program for the Atmel ATmega16 embedded micro controller. As mentioned earlier, we may use the compiler from [Roh06] to translate this program, which is in the form of the assembly language of the micro controller, to the NIPS byte-code language. This test is interesting because we may evaluate the effects of the reductions on a model which was not hand written, and which corresponds to an actual program. Also, this particular program consists of a single process which models the micro controller, and thus, we are also able to evaluate the reductions when applied to a sequential model.

As in the case of the Spin examples, we test the unreduced model, each reduction individually, and the combination of all the reductions. The machine which was used for these tests is the same machine used for running the Spin examples, and the corresponding results are shown in Table 10.3. We use the same naming conventions as in Table 10.1 and Table 10.2.

First of all, we notice that the only reduction technique which causes an effect on this model is path reduction. The execution of the model when reduced with respect to the other two reductions is exactly the same as the execution of the original unreduced model, with the exception of dead variable reduction which causes a slight increase in the execution time because of the reasons already discussed in the previous section.

However, the effect that path reduction achieves in this example is particularly good: it discards almost 82% of the state space of the original model. The computation time is also reduced by one half, which accounts for the creation of less states but while executing

Model	States		Transitions		State Memory		Elapsed Time	
	Total	%	Total	%	MB	%	sec.	%
lunar_4nodes_1down.b	226598	100.00	246664	100.00	126.90	100.00	3.29	100.00
lunar_4nodes_1down+pr.b	65813	29.04	76169	30.88	36.87	29.05	1.87	56.84
lunar_4nodes_1down+dvr.b	224916	99.26	244982	99.32	125.96	99.26	3.62	110.03
lunar_4nodes_1down+scr.b	226507	99.96	246534	99.95	126.85	99.96	3.31	100.61
lunar_4nodes_1down+pr+dvr+scr.b	65714	29.00	76031	30.82	36.82	29.01	2.09	63.53
lunar_4nodes_1down.spin	21606	9.53	66029	26.77	10.56	8.32	1.03	31.31
lunar_4nodes_1down+por+opt.spin	6839	3.02	14678	5.95	3.46	2.73	0.26	7.90
lunar_4nodes_1up.b	8119319	100.00	8723018	100.00	4610.68	100.00	93.14	100.00
lunar_4nodes_1up+pr.b	3265954	40.22	3853245	44.17	1854.89	40.23	66.59	71.49
lunar_4nodes_1up+dvr.b	7872574	96.96	8476273	97.17	4470.52	96.96	107.13	115.02
lunar_4nodes_1up+scr.b	8119298	100.00	8722988	100.00	4610.66	100.00	92.12	98.90
lunar_4nodes_1up+pr+dvr+scr.b	3265933	40.22	3853215	44.17	1854.88	40.23	81.72	87.74
lunar_4nodes_1up.spin	612915	7.55	2117030	24.27	292.06	6.33	37.08	39.81
lunar_4nodes_1up+por+opt.spin	303469	3.74	892781	10.23	144.78	3.14	15.35	16.48
lunar_4nodes_simul1.b	1668408	100.00	1779254	100.00	1020.22	100.00	19.43	100.00
lunar_4nodes_simul1+pr.b	637818	38.23	734444	41.28	390.36	38.26	13.72	70.61
lunar_4nodes_simul1+dvr.b	1611913	96.61	1722759	96.82	985.63	96.61	21.94	112.92
lunar_4nodes_simul1+scr.b	1668408	100.00	1779254	100.00	1020.22	100.00	19.53	100.51
lunar_4nodes_simul1+pr+dvr+scr.b	637786	38.23	734412	41.28	390.34	38.26	16.27	83.74
lunar_4nodes_simul1.spin	129366	7.75	409334	23.01	68.18	6.68	7.54	38.81
lunar_4nodes_simul1+por+opt.spin	58487	3.51	151143	8.49	30.93	3.03	2.81	14.46

Table 10.4: Results of running the LUNAR models.

approximately the same amount of instructions. The actual code transformation applied by path reduction in this case removed 17 out of a total of 19 **STEP** instructions, leaving just one **STEP** N instruction plus a terminating **STEP** T instruction. Thus, every state produced for this model is due to the same **STEP** N instruction, with the exception of the initial state and possibly a final state created by the execution of the **STEP** T instruction. This shows how drastic the reduction obtained by path reduction can be, both in the number of **STEP** instructions of the byte-code program and in the size of its corresponding state-transition system. This is true even in the case of a sequential program like this example, whose state space is not created by the interleaving of concurrent processes and is therefore not affected by the state-explosion problem.

Finally, we may point out that since this model corresponds to a sequential program, we can therefore expect no reduction to occur when partial order reduction is applied to it, in accordance to what was mentioned in Chapter 7. Although we may not verify this experimentally, it does highlight the importance of purely static state space reductions like path reduction when performing state exploration on sequential programs.

10.4 LUNAR models

The last performance test that we will consider corresponds to a set of models for the LUNAR protocol [TGRW04, WPP04]. LUNAR is a routing protocol for wireless ad hoc networks, and has been verified using Spin by modeling several possible scenarios of the topology of the network. This offers an opportunity to test the effects of the reductions on models which have been created to model check a real protocol and its implementation. Table 10.4 shows the results of running three of these models under NIPS and Spin, in the same way that was described in section 10.2. The machine used for these tests has four AMD Opteron™ 840 processors and 16 GB of memory.

In these tests, Spin outperforms NIPS in the size of the generated state-transition

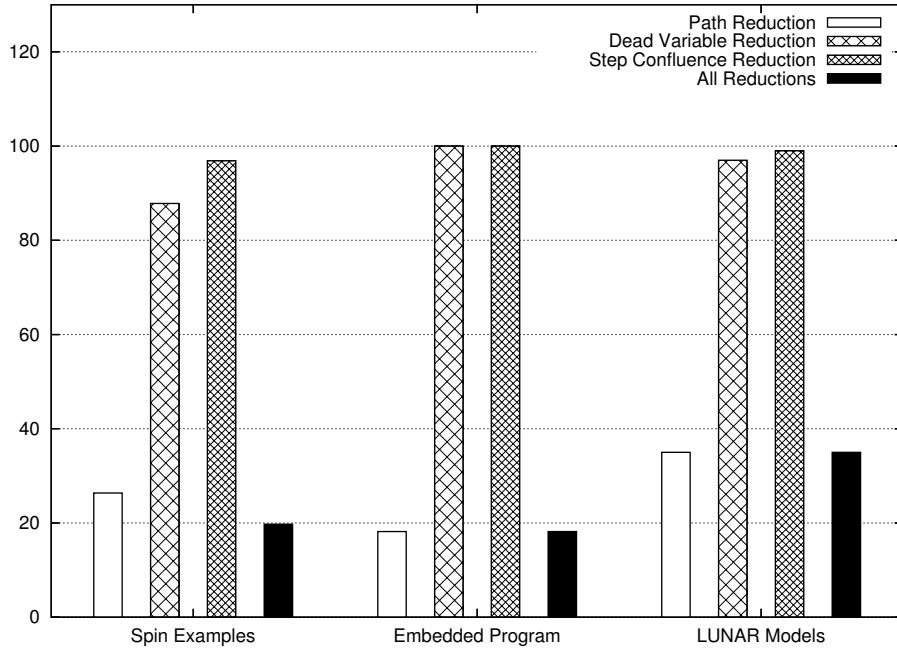


Figure 10.1: Average percentages of generated states.

system by an order of magnitude when no reductions are applied. This difference is far greater than that which was observed when running the Spin examples, and was not overcome when applying the state space reductions. However, in relation to the size of the state space generated by the unreduced NIPS model, the combined reductions create state spaces which are 64% smaller on average. Practically all of this reduction is due to path reduction, with small contributions from dead variable reduction and to a lesser extent from step confluence reduction. With respect to computational time, the reductions achieve a decrease of 22% on average with respect to the original model. In general, this is an important reduction and shows that the effectiveness of these state space reduction techniques may also be experienced with real models used in the verification of actual communication protocols.

10.5 Summary

In order to obtain a general view of the results of these performance tests, we may compare the relative state space reduction achieved on average by each reduction technique in each of the three classes of tests that were conducted. Figure 10.1 shows a graph which represents the average size of the state space that was obtained by each reduction in each of the test categories with respect to the unreduced original model, as well as the corresponding size for the case when all three reductions worked together.

The superiority of the effect of path reduction with respect to the effects of the other reductions is clear in this representation, where path reduction consistently reduces the state space by factors of 60% to 80%, many times more than what is achieved by the other reductions.

The effect of dead variable reduction is more dependent on the type of model, and

although it was able to perform reductions of up to 40% in some models, on average its reduction results in a factor of up to 10%. This is nevertheless an important reduction, especially in the case of models with very big state spaces.

The results obtained by step confluence reduction are less impressive. On average we may expect a reduction factor of up to 5% when this reduction is applied.

The clear winner in this case is the combination of all the reduction techniques. Although path reduction accounts for almost all of the reduction obtained by this multiple reduction technique, it still benefits from the application of dead variable reduction and step confluence reduction, as may be seen in the case of the Spin examples and to a lesser extent in the case of the LUNAR models.

In general, the reductions are less effective when applied to a real protocol model, as in the case of the LUNAR models. This agrees to what is argued in [Pel05] about the real effectiveness of state space reductions when applied to actual case studies and not just tailored examples. Still, the reduction obtained by the combination of the static reduction techniques on these models is very positive indeed.

Chapter 11

Conclusions

This work has presented a set of techniques for reducing the size of the reachable state space of models encoded in the NIPS byte-code language. These techniques are based on the static analysis of the byte-code of these models, and operate exclusively by performing a transformation of this byte-code. The reductions obtained by these techniques have been shown to preserve the equivalence of the original and reduced state-transition systems with respect to temporal logic specifications, and their effectiveness has been demonstrated experimentally. This chapter presents the concluding remarks of the present work.

11.1 Related work

Path reduction and other similar reductions have been presented in [Hol99, Yor00, KLY02, YG04]; likewise, dead variable reduction has been presented in [BFG99, Hol99, Yor00, YG04]. Arguably, these reductions are simpler to describe and implement for the byte-code language of NIPS than in their original presentations, thanks to the simple structure of the byte-code.

In the case of path reduction, the analysis and code transformation need only be done at the corresponding locations of each individual breaking point and `STEP` instruction when applying it to NIPS, and without any further regard for the remaining instructions or the structure of the control-flow graph, other than that the consideration of cycle safety. Also, the corresponding code transformation is extremely simple. The path reduction optimization as presented in [Yor00, YG04] requires a more complex manipulation of the syntactic structures of the high-level language used therein, and it relies on some duplication of code when statements are shared among elementary paths. The analysis and algorithms presented in [KLY02] are also notably more complex than the ones that have been presented for NIPS. Finally, the statement compression technique described in [Hol99] is limited to the merging of statements with strictly deterministic behavior, and is therefore less powerful than the version of path reduction presented in this work. Also, it must perform its analysis directly on the syntax of Promela code and keep track of which statements may be compressed when producing the state-transition system. The analysis of the NIPS byte-code and the interpretation of the modified code by the virtual machine may be seen as a simpler way to achieve this.

Regarding dead variable reduction, the analysis presented in [Yor00, YG04] consists of a modified live variable analysis for the high-level WHILE language which computes

the sets of dead variables by applying inductive rules on the syntactic structure of the program. The analysis of death points and dead assignments presented in Chapter 5, and which is based on the result of a standard live variable analysis of the control-flow graph, represents a simpler form of analysis. The reduction presented in [BFG99] is of a somewhat different nature than the dead variable reduction presented in this work. The language considered there is a process algebra whose control graph corresponds to the parallel composition of a fixed number of processes. Thus, they are able to compute an accurate live variable analysis for shared variables. Also, they are able to extend the analysis to the values contained in communication queues. This does not apply to the static reduction applied to NIPS, which must handle each of the process types individually and prior to their concurrent execution. In [Hol99], the resetting of temporarily dead variables to a value of zero is analogous to the optimization due to death points in NIPS; the handling of dead assignments is not mentioned in this paper.

The idea behind step confluence reduction came while observing the byte-code programs produced by the Promela compiler provided with NIPS. The compilation of `goto` and `break` statements caused the creation of confluent `STEP` instructions. A similar optimization is performed by the Spin compiler, which inhibits the execution of these statements and uses them to determine the ending location of the immediately preceding statement [Hol03]. It was not possible to find any additional approaches to reduce the size of the reachable state space by removing differences among states which consist of equivalent program locations.

11.2 Implementation

As mentioned in Chapter 2, the nature of the NIPS virtual machine and its byte-code language greatly facilitates the implementation of state space reductions based on static analysis of the model's code. The byte-code has a simple structure which is easy to represent, read, analyze and transform within a software tool. Also, it is easy to visualize the results of the reductions when they consist of a transformation of the code, and it is equally easy to evaluate these results by just executing the reduced byte-code within the virtual machine, and without the need to modify or reconfigure this machine in any way. Finally, any number of transformations may be combined in a simple manner. All these characteristics make the NIPS virtual machine a most convenient platform for the development of state space reduction techniques, as well as for the model checking applications which may benefit from these techniques.

The fact that these reductions are applied off-line and prior to the execution of the byte-code alleviates their implementations from any strict efficiency constraints. In any case, static analyzes are not inherently complex and thus do not usually have high requirements for memory and computational time. For instance, the application of each of the state space reductions implemented in SARN normally takes no more than a few seconds to complete, even in the case of big models. Their state space exploration, on the contrary, can take several minutes. Thus, it is specially good that the algorithms for the reductions do not cause a runtime overhead for the virtual machine.

Because of these considerations, the JavaTM language was chosen for the implementation of SARN. It allowed a very clear modeling of the different elements of the NIPS

byte-code, such as instructions and their modifiers, parameters, control-flow structure, etc. Also, it was very convenient to use the visitor pattern to implement a generic traversal algorithm of the control-flow graph and later implement visitor objects which make specific uses of the traversal. Most of the analyzes of SARN, such as the analysis of breaking points, blocking steps, elementary steps, death points, dead assignments, confluence points and the detection of unsafe loops, are all implemented in this way. Also, the generic container classes included in the standard API such as `Set`, `List` and `Map` were extensively used throughout the implementation of SARN, as well as the enhanced `for` loop which may be used to iterate through the elements of these containers in a very convenient and type-safe manner.

11.3 Experimental results

Chapter 10 presented the results of applying the state space reductions to three different types of tests: the examples of the Spin model checker, a sequential model generated from an embedded program, and the models of a wireless routing protocol. These three tests are each of a different nature and with different characteristics. Some are hand written, while the model of the embedded program is automatically generated; some correspond to relatively small models while others are particularly big and complex. Finally, some are examples written to show different aspects of the Spin model checker, while others like the LUNAR models represent the actual implementation of a protocol.

The results of the reductions on these models are in general very positive, as they are able to obtain combined reductions of up to 80% in the number of reachable states. Path reduction accounts for most of this reduction, but in general it is advisable to apply all reductions together in order to obtain the best possible reduction.

These reduction techniques enjoy all the advantages and benefits which were outlined in Chapter 2, namely, having a simple implementation as a code transformation, incurring in no runtime overhead, constructing the reduced transition system directly from the model, permitting the simple reuse of reduced byte-code files, working independently of any high-level modeling language and thus being reusable with any new language which is implemented using NIPS, and making it possible to combine these reductions in a simple and straightforward way. Also, they are effective even when being applied to models which lack concurrency.

11.4 Future work

A number of possible enhancements to the reduction techniques presented here have been identified throughout the development of this work. They have not been made part of this work because of time constraints. However, are presented in this section in order to show some the possible extensions to these static state space reduction techniques, and which are still unexplored.

- *Abstract interpretation.* Throughout the definitions of the static analyzes presented in this work, it has been mentioned that no abstract interpretation of the byte-code is performed, thus limiting the scope of the analysis to a purely static approach.

Indeed, an abstract interpretation of the byte-code, including constant propagation and constant folding, may permit more accurate results in these analyzes and correspondingly, better state space reductions. For this, it would be useful to create a basic block representation of the control-flow graph as described in [ASU86]. Also, it would be required to apply this analysis to the values of the variables, registers and the operand stack.

- *Determination of exclusive access of global variables.* Consider a global variable which is used exclusively by one process type. Furthermore, assume that this process is instantiated only once during the execution of the system. Then, the global variable is really accessed as if it were a local variable of this process. An analysis that is able to detect this situation would allow dead variable reduction to be applied to this variable even in the case of a concurrent system. Likewise, any accesses to this variable need not be considered as external operations when applying path reduction. Thus, both reductions would benefit from this analysis, which would need to determine which global variables are used by which process types, and also, if a given process type may be instantiated multiple times or not.
- *Channel analysis.* In [MT99], an analysis of Promela is described which determines statically which channel variables may be aliased among each other in any point of the execution of the program. Although this analysis is intended for the slicing of Promela models, the analyzes presented in this work may also benefit from such an analysis, as it offers more precise information about the relationship between channels and variables.
- *Elimination of write-only and unused variables.* In [Hol99], it is described how Spin detects write-only global variables as well as unused variables and completely removes them from the representation of each state. Whereas dead variable reduction causes every variable which is *temporarily* dead to have a default valuation, this reduction completely *eliminates* variables which are *always* dead. Since NIPS is likely to be used as a back-end for compilers, this task will commonly be done by the compiler itself. However, it is possible to implement this optimization directly in NIPS byte-code and thus permit its reuse among all tools which target NIPS.
- *Program slicing.* A program slicing technique for reducing the size of a Promela model with respect to a given temporal logic specification has been presented in [MT98, MT99]. It is therefore feasible to adapt this same technique to NIPS, which is straightforwardly implementable as a code transformation like the ones presented in this work.
- *Partial order reduction.* Partial order reduction is the best known state space reduction technique used in software model checking. A mostly-static version of partial order reduction like the one presented in [KLM⁺98] can be applied to NIPS. This requires, however, an adaptation of the virtual machine so that it allows the exploration of successor states to be guided by a component that computes the ample sets dynamically, based on statically gathered information.

Thus far, the possibilities for state space reduction in NIPS are very promising, considering the empirical results of this work. The development of future and enhanced

reductions based on the static analysis of NIPS byte-code will support the creation of novel model checking applications which make use of the NIPS virtual machine for the generation of state-transition systems.

Bibliography

- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.
- [BFG99] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. State space reduction based on live variables analysis. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 1999.
- [CGMP98] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *Software Tools for Technology Transfer*, 2, 1998.
- [CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
- [Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
- [Hol99] Gerard J. Holzmann. The engineering of a model checker: The Gnu i-Protocol case study revisited. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pages 232–244, London, UK, 1999. Springer-Verlag.
- [Hol03] Gerard J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
- [HP94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
- [IZ93] M. Robert Ito and A. Zaafrani. Data flow analysis for parallel programs. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 318–325, New York, NY, USA, 1993. ACM Press.
- [KLM⁺98] Robert P. Kurshan, Vladdimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static partial order reduction. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 345–357, London, UK, 1998. Springer-Verlag.

- [KLY02] Robert P. Kurshan, Vladimir Levin, and Hüsnü Yenigün. Compressing transitions for model checking. In *CAV*, pages 569–581, 2002.
- [Lam83] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Proceedings of the IFIP Congress on Information Processing*, pages 657–667, Amsterdam, 1983. North-Holland.
- [LC91] Douglas Long and Lori A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 21–35, New York, NY, USA, 1991. ACM Press.
- [Mil99] Robin Milner. *Communicating and mobile systems: the π -calculus*. Cambridge University Press, New York, NY, USA, 1999.
- [MT98] Lynette Millett and Tim Teitelbaum. Slicing promela and its applications to protocol understanding and analysis. In *SPIN '98: 4th International SPIN Workshop*, 1998.
- [MT99] Lynette I. Millett and Tim Teitelbaum. Channel dependence analysis for slicing promela. In *PDSE*, pages 52–61, 1999.
- [Pel05] Radek Pelánek. On-the-fly state space reductions. Technical Report FIMU-RS-2005-03, Masaryk University Brno, 2005.
- [Roh06] Michael Rohrbach. Model checking embedded systems software. Diploma thesis, Lehrstuhl für Informatik IXI, RWTH Aachen, Germany, 2006.
- [Sch05] Stefan Schürmans. Ein Compiler und eine Virtuelle Maschine zur Zustandsraumgenerierung. Diploma thesis, Lehrstuhl für Informatik II, RWTH Aachen, Germany, 2005.
- [TGRW04] Christian Tschudin, Richard Gold, Olof Rensfelt, and Oskar Wibling. LUNAR: a lightweight underlay network ad-hoc routing protocol and implementation. In *NEW2AN '04: Proceedings of Next Generation Teletraffic and Wired/Wireless Advanced Networking*, 2004.
- [WPP04] Oskar Wibling, Joachim Parrow, and Arnold Pears. Automatized verification of ad hoc routing protocols. In *FORTE 2004: 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, volume 3235 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2004.
- [WS05] Michael Weber and Stefan Schürmans. A virtual machine for state space generation. Lehrstuhl für Informatik II, RWTH Aachen, Germany, 2005.
- [YG04] Karen Yorav and Orna Grumberg. Static analysis for state-space reductions preserving temporal logics. *Formal Methods in System Design*, 25(1):67–96, 2004.

- [Yor00] Karen Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion, Israel Institute of Technology, Haifa, Israel, 2000.