



# A Debugging Framework for NIPS

*Master thesis*

by Riemer van Rozen

October 2007

Committee

dr.ir. Theo Ruys
dr.rer.nat. Michael Weber
dr.ir. Arend Rensink

Research Group

University of Twente
Department of Computer Science
Formal Methods and Tools Group

## Acknowledgements

I would like to thank Theo Ruys and Michael Weber for being my first and second advisor, for their ideas and feedback and the long meetings at the UT and the CWI,
Arend Rensink for being third advisor,
Patricia Dockhorn for some general pointers about the thesis structure, my colleagues Frank van Es and Paul Zandbergen for providing feedback on spelling mistakes and sentence structure, Ismenia Galvao for being patient and sweet and my parents for supporting me during my education and for raising me.

## Intended Audience

The intended audience of this thesis report consists of experienced professionals in the field of Formal Methods and Software Engineering, as well as students with interest in debugging information.

SDI
**S**imple, **S**tatic, **S**murf
Elf → Dwarf → Smurf

# Contents

# Chapter 1

# Introduction

Software Engineering is the field in Computer Science concerned with the process of software development and all its intricacies. The system design life cycle can be modeled using the *waterfall model*, in which the sequential development phases are conceptually linked [34] as is depicted in Figure 1.1. The phases are often repeated iteratively, resulting in a new version of the software every iteration.

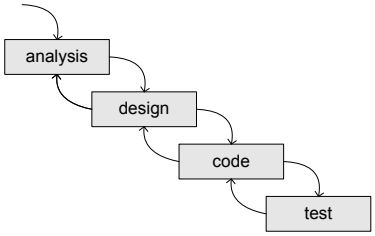



Figure 1.1: Waterfall Model

A major goal of software engineering is to enable developers to construct systems that operate reliably despite their scale and complexity. To this end a lot of time and attention is spent on testing whether the software system meets its requirements.

Formal methods are mathematically based languages, techniques, and tools that can be used to specify and verify large and complex systems [7]. Tools have been derived from these formal methods that offer functionality supporting activities in the design, coding and testing development phases. The software industry is motivated to use these tools because they can play an important role in developing reliable and quality software. They can help in the early identification of software (design) errors which become more expensive and time-consuming to find and repair later in the system design cycle. The tools that we are concerned with in this thesis report are *explicit state model checkers* and *debuggers*.

## 1.1 Background

Model checkers, compilers and debuggers are tools that provide specific contributions to the formal design of software systems. This section introduces basic ideas concerning them.

### 1.1.1 Model Checking

**Model.** A model formally describes the behaviour of a system whilst abstracting from details which are not relevant for its use. The representation of a model may be abstract, but models may also be textually represented in a modeling language. Design models can be used to formally specify systems before they are implemented or they can be extracted from system implementations.

**Simulation.** A simulation is a step-by-step execution of a model. Simulations can be used to show the behaviour of a system. In particular, it can be used to show a *counter-example*, which is sequence of steps leading to an undesired situation.

**Property.** A property specification formally describes a requirement about the system which can be an invariant or related to safety, fairness, liveness, etc. Properties are typically described using formal specifications which are expressed as logic formulas, e.g. Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) and may be associated with an automaton [23].

**Model Checking.** Model checking is a formal method used to automatically verify the correctness of finite-state systems with respect to specification properties. Verification algorithms are used to traverse every possible behaviour of the model, also referred to as the *state space*, to check whether a property holds (is true) or not. If the property holds then the model satisfies a specification. If the property does not hold, a counter-example is produced.

In [13] the two fundamental approaches to model checking are described: In *symbolic model checking* a symbolic representation for the state set is used, usually based on binary decision diagrams. Validating a property in symbolic model checking amounts to performing a symbolic fixpoint computation. Symbolic model checking works especially well for hardware verification. In *explicit state model checking* an explicit representation of the system's global state graph is used, usually given by a state transition function. The validity of LTL properties over a model are evaluated by interpreting its global state transition graph as a Kripke structure. For every LTL formula there exists a Büchi automaton that accepts precisely those runs that satisfy the formula. Verifying that a model $M$ satisfies a property $\Phi$: $M \models \Phi$ entails performing a partial or a complete exploration of the state space. A comprehensive foundation to model checking is given in [6].

**Model Checker.** A *model checker* is a tool that is concerned with automating the search for errors in software (designs) by providing model checking as a push-the-button functionality. Usually a model checker also supports one or more forms of simulation.

One model checker that has been successfully applied in many software design projects is Spin [20]. The input language of Spin is Promela (Process Meta-Language) which is a high level language used to model concurrent systems. Spin will be a benchmark tool reference in this report. Indeed much of the ideas in this report, the context and the reasoning have been derived from Spin or are strongly related to it as will be made apparent in subsequent chapters.

### 1.1.2 Compiling and Debugging

Compiling and debugging code are strongly related activities. We will give an introduction to both compilers and debuggers and explain what they have to do with each-other.

Compilers can be used by programmers to translate high level languages to low level runnable machine code. The process of compiling a program from human-readable form into the machine code that a processor can execute is described in [10] as: *"successively recasting the source programs into simpler and simpler forms, discarding information at each step until, eventually, the result is a sequence of simple operations, registers and memory addresses and binary values that consist of zeros and ones."*

A *multi-pass compilation scheme* can be described as a sequence of steps (or passes). Each step performs a specific operation on the same structures in order to perform a translation. In contrast, a *single-pass compilation scheme* performs the translation in one step. Figure 1.2 schematically depicts a multi-pass compilation scheme at compiler time and shows that we can either run or debug a program at run-time. Multi-pass compilers are also explained in [49].

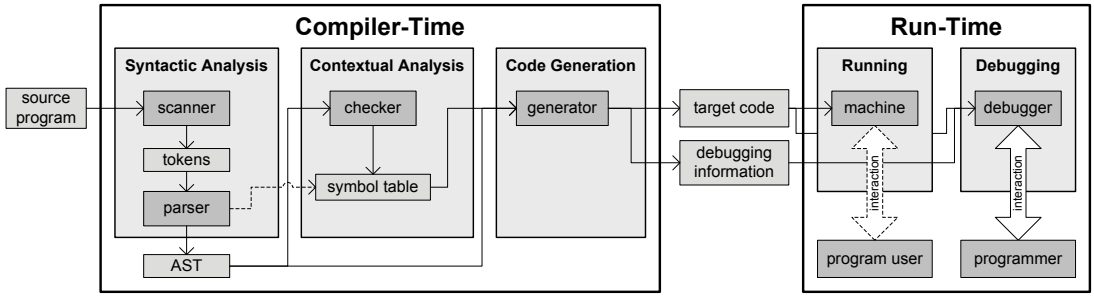



Figure 1.2: Multipass Compiler

**Multi-pass compilation.** The first phase of the compilation process is the *syntactic analysis* phase in which the source program is scanned by a *scanner* (also called lexer) and represented as a stream of *tokens*, the basic textual building blocks of the language. They are the input for the *parser* which is based on a *grammar* and creates an Abstract Syntax Tree (AST), an intermediate representation of the program syntax. It is practical to describe a grammar in EBNF (Extended Backus-Naur Form) which is a standard form to describe the structure of programming languages. After the syntactic analysis phase, the compiler performs a *contextual analysis* of the AST which means it is analyzed for type correctness and contextual constraints. A *symbol table* is used to store and retrieve information about variable declarations in order to facilitate scope and type checking and code generation. The information about variables that may be retrievable are its *type*, the *source line* and *column* number of the corresponding token and the *memory location* at which the variable will be saved within the machine at run-time. The final phase of the compilation is the *code generation* in which the AST of a program is translated to a lower level language called the *target language*. Sometimes it is necessary for an *assembler* to transpose the generated code before it can be executed. Assemblers may create *object files*, binary files that contain object code which consists of zeros and ones. In order to create an executable, object files have to be linked to other object files by a *linker*. Compilers can be automatically generated from compiler generator tools such as Lex and Yacc [24, 22], ANTLR [30] and SableCC [16].

**Run-time.** A compiled program can be run by a program user who may interact with the program and view the results on the screen. The compiled program consists of code that is either run directly on the Central Processing Unit (CPU) or it is a *byte code* that is interpreted by a Virtual Machine (VM). A VM differs only from physical machines in that it is not represented by a hardware component directly but is a program itself running on another machine.

**Debugging.** In case of a program error at run-time the programmer can either examine the source code or the generated code. In order to be able to examine binary target code it must be analyzable in terms of the source code. A *source-level debugger* is a tool that depicts the progress of a program in terms of its program source and allows control over the program control flow in order to find software bugs.

*Debugging* is the process in which software developers use a debugger to prove or disprove hypotheses about the source program. Debugging is related to model checking, simulation and testing. In testing, hypothesis about the program are described as test-cases which consist of a predefined expected result and the observed result. When the two correspond the test passes, otherwise it fails. Aside from being used for *complete validation* to certify the quality of the product or design model by establishing its absolute correctness, a model checker can be used as a *debugging aid* to find residual design and code faults using partial state space exploration methods [13]. The counter-examples produced by model checkers provide a means to simulate the source program and direct the behaviour to the error state. Testing, debugging and model checking are complementary activities because any verification is only as good as the validity of the system model.

Ryu et. al describe two fundamental approaches to source-level debugging of compiled code in [39]: The first approach is *reverse engineering* where the compiler generates code and additional information that enables the debugger to analyze the object code and report information at the source level, e.g. `ldb` [35], GDB and ACID [52]. The second approach is *instrumentation* where the compiler modifies the program code and inserts extra instructions that are used for debugging, e.g. `smld` [45]. Although instrumentation can provide debugging support with a modest effort it is also slow at run-time [39].

**Debugging Information.** We define *debugging information* as the extra, optional information generated by the compiler which is (usually) not necessary for programs to run, but which is necessary for debuggers to make source-level debugging possible.

**Reflection.** Debugging information is related to an object oriented design pattern called *reflection*. Bobrow et al. define reflection in [4] as a mechanism for observation and modification: *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution"*. There are two aspects of such manipulation: *introspection* and *intercession*. The first aspect is the ability of a program to observe and reason about its own state. The second is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require *reification*, which is a mechanism for encoding execution state as data.

Parson et al. pose in [31] that: *"Reflection provides a powerful means for a target software system to expose aspects of its implementation to a tool such as a debugger, so that the tool can configure its user interface and command set to adapt to the special requirements and capabilities of the target system."*

### 1.1.3 The Nips VM

The Nips VM is targeted to be a fast, reusable, *Embeddable Virtual Machine for State Space Generation* [50]. The Nips VM and the Nips byte code it runs are designed for operational models of high-level languages for use with verification tools. The VM can play the role of the state space generation back-end in an *explicit state model checking framework*. Using the Nips VM as tool back-end saves the tool engineer the often complex and time-consuming task of having to design and implement a model checking engine. Furthermore, the design allows the reuse of modeling languages and common (byte code) analyses.

The Nips VM runs Nips byte code supplied by a compiler, executes its semantics and generates *states vectors*, low level snapshots of the system behavior, based on the byte code and an input state vector. The Nips VM Application Programming Interface (API) is schematically depicted in Figure 1.3.

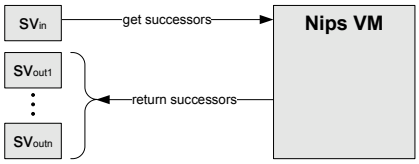



Figure 1.3: Nips VM API

The Nips Promela Compiler translates Promela models, created by the tool user into Nips byte code. Together with the Nips VM and a scheduler component, it forms an explicit state model checker that offers comparable functionality to the Spin model checker.

The Nips VM is mainly distinguished from other explicit state model checkers by its modular design and its small implementation. Consequently, it can be embedded in host-tools as a model checker engine. The host-tools determine the strategy for the search of the state space and may use any high level language, provided that it can be translated to Nips byte code. Furthermore, the Nips byte code can be optimized as a means for state space reduction and the Promela compiler allows the reuse of a large amount of case-studies.

## 1.2 Problem Statement

This thesis is concerned with working towards fulfilling the promise of the Nips VM as a reusable component for state space generation that is part of a extensible tool set of explicit state model checker components. The following problems related to model checking and debugging regarding the Nips VM can be identified.

**State BLOBs.** The main problem with the Nips VM is that it cannot be properly used since the output consists of state vectors, i.e. arrays of unnamed, untyped bytes which are displayed as Binary Large Objects (BLOBs).

**Debugging.** Consequently, the Nips VM misses debugging functionality. Users of Nips VM tools cannot analyse the behaviour of design models compiled to Nips byte code. The need exists for a *debugger* that enables users of Nips VM based tools to *analyse the results* the tool

generates.

**Debugging Information.** We lack the language to describe source level information that can be used by a debugger for Nips to display states at run-time. The question is what kinds of debugging information there are and what they can be used for. What should debugging information tailored towards the Nips VM look like, and how can this information be provided by compilers targeting it?

**Embeddable Nips VM.** The Nips VM is designed to be an embeddable component for state space generation, but it is unclear how it can be embedded in host applications because it consists of undocumented C code. For tool engineers that wish to embed the Nips VM as a explicit state model checker back-end, it needs to be clear what the Nips VM API is in terms of functions and procedures and their arguments, such that they can design host applications that can gain access to the VM. How can the Nips VM be embedded? Does the VM offer all the services host applications need to make use of it? If not, then in what way does the API need to be extended or altered?

## 1.3 Objectives

The primary goal of the research work elaborated in this thesis is to allow users to make use of Nips VM based tools and to make it more attractive to tool engineers to embed the Nips VM as a tool back-end. The objectives are sub-devised as follows.

- **Readable States.** States and counter-examples should be unparsed to their source-level equivalent allowing program debugging, simulation and viewing model checking results.
- **Embeddable Nips VM.** The Nips VM should be more easily embeddable in host applications by giving access to state components and facilitating state introspection, paving the way for state-of-the-art state space reduction techniques such as *state collapsing* [19] and *Partial Order Reductions* (POR) [6, 36].

The research must be applicable to the field of explicit state model checking but the specific goal of the research is to extend the application field of the Nips VM.

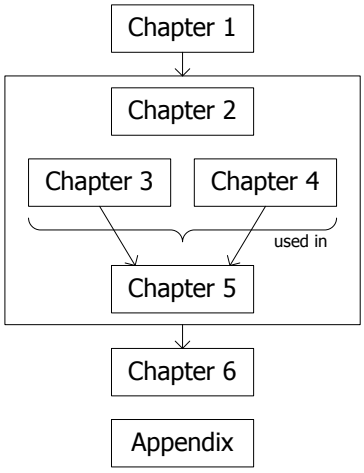
## 1.4 Approach

We discussed the problems regarding the Nips VM related to model checking and debugging in Section 1.2 and set specific goals to achieve a subset of these problems in Section 1.3. Here we give an outline of the approach on how to achieve the objectives and a chapter structure of the report. Figure 1.4 shows the chapter overview.

Chapter 2 places the thesis research in context. It elaborates and explains references used in the introduction background. Existing definitions of debugging information formats are discussed to see if there is a likely candidate to be used with the Nips VM. Existing implementations of explicit state model checkers similar to the Nips VM are compared with the Nips VM.

The study into related work in Chapter 2 yielded no immediate solution that could easily be adapted to support a debugger for the Nips VM. Therefore, a new debugging information lan-

guage must be defined which is both general and reusable, but which can be specifically used to support a debugger for the Nips VM.




(a) Chapter Structure

| | |
|---|---|
| Chapter 1: | Introduction |
| Chapter 2: | Related Work |
| Chapter 3: | The SDI Language |
| Chapter 4: | The SDI Framework |
| Chapter 5: | Case Study: The Nips Debugger |
| Chapter 6: | Conclusions & Future Work |
| Appendix A: Enhancing the Nips VM | |

(b) Chapters

Figure 1.4: Chapter Overview

In this thesis the reverse engineering approach to debugging is applied. We believe that debugging information provides a means to define an API that gives access to the memory model of a running program, which can be used for debugging[1].

A *debugging API* provides the means for a debugger to offer debugging functionality such as displaying states at run-time to the user in an understandable way and allowing users to edit state values. Displaying states can be seen as a form of introspection, editing state values as a form of intercession and debugging information as a means for reification.

The principal contributions of this thesis for reaching the objectives defined in Section 1.3 are presented in Chapters 3, 4 and 5. This thesis introduces a simple multi-use readable format for Static Debugging Information (SDI) in Chapter 3 and the SDI Framework for state manipulation, which is based on the SDI language, in Chapter 4. SDI is a high level modeling notation for debugging information that is meant to describe the source-level elements of modeling languages used with explicit state model checkers. The SDI Framework facilitates a reflective debugging API that consists of function calls that enable debuggers to display and modify the information in state vectors associated with running programs for which memory models have been defined using SDI.

The generic results of the research are applied to the Nips VM in particular. The Nips VM Tool Set is extended with a source-level command-line debugger that allows users to simulate the behaviour of Nips byte code in Chapter 5. The design of the debugger based on the SDI framework is treated as a case study, an in-depth examination of the application in order to gain understanding about the investigated approach.

[1] though its applications are not limited to debugging

The thesis is concluded in Chapter 6 with considerations about the results of the research, motivations and the design and implementation of the SDI Framework and its application to the Nips VM.

Additionally, the Nips API is documented and it is described how to embed it as a tool backend in model checker host applications in Appendix A. This appendix also describes design suggestions for enhanced components for future versions of the Nips VM. Appendix B describes the extensions to the Nips Promela Compiler. Appendix C details an example used throughout this thesis to illustrate our approach. In Appendix D a user manual for the Nips Debugger is presented.

# Chapter 2

# Related Work

This chapter discusses work related to our approach. The Nips Virtual Machine (Nips VM) is more thoroughly introduced than thus far in Section 2.1. After this introduction we discuss debugging information formats as candidates for use with the Nips VM in Section 2.2. Next, explicit state model checkers related to the Nips VM are discussed in Section 2.3, particularly the debugging solutions employed in the tools. How are counter-examples related to the source and how are they presented to the tool user? The debugging solutions offered by the discussed formats and tool designs and languages are compared in Section 2.4 to finally decide about the approach for the debugger for Nips in Section 2.5.

## 2.1 The Nips VM

The Nips VM is described in [50, 51] as a Virtual Machine for state space generation that is designed as a modular, efficient, reusable, embeddable explicit state model checker tool engine (or back-end). It can execute Nips byte code instructions that are translations for high level modeling languages. Executing a Nips byte code program yields a state space that can be used with model checkers simulators and testing tools. The Nips acronym is the reverse of Spin and has (at least two) different meanings: *New Implementation of* Promela *Semantics* and *Never Implement* Promela *Semantics (again).*

### 2.1.1 Motivation

The design of the Nips VM and Nips byte code for implementing an operational model of high-level languages for use in verification tools is motivated by four main arguments [50]. Firstly, it is highly desirable to reuse an already existing modeling language like Promela and reuse existing case studies instead of having to resort to artificial examples. Secondly, the tool developer can focus on the design and implementation of algorithms when using a reusable (or re-implementable) component for state space generation that can easily be interfaced with the tool infrastructure. Thirdly, tool users can switch to other tools with the same input language without having to reimplement the model in another formalism. Lastly, using the Nips VM as a tool back-end allows the implementation and reuse of *common analyses* such as dead variable

reduction and statement merging irrespective of the high level modeling language being used.

### 2.1.2 Language

The NIPS VM runs NIPS *byte code*, an intermediate language, that serves as a means to describe the operational semantics of modeling languages. NIPS byte code works on three types of run-time components: the *global component*, *processes* and *channels* which can be used with all compilers targeting the NIPS VM.

NIPS byte code supports *non-determinism*, *concurrency*[1] of run-time created processes, *rendezvous and asynchronous communication* between processes via channels, sending a *channel as a variable* message via another channel, *priority* control of byte code execution and *speculative execution*. Speculative execution entails that the changes to states caused by byte code of which the execution cannot complete are undone (rolled-back). The byte code can be used to encode LTL properties in the model itself into a *monitor* process. LTL properties are described in PROMELA as `never` claims. A full description of the NIPS byte code can be found in [50].

The NIPS byte code contains incomplete debugging information for PROMELA programs in the form of debugging information strings. These strings are limited in their expressiveness and their meaning depends on the relative placement in the code. Source location markers consist of *line* and *column*. Name markers consist of a *begin* or *end* tag, followed by a *keyword* and possibly a *name*. They do not provide a means for a debugging API and they seem to be designed only with PROMELA in mind.

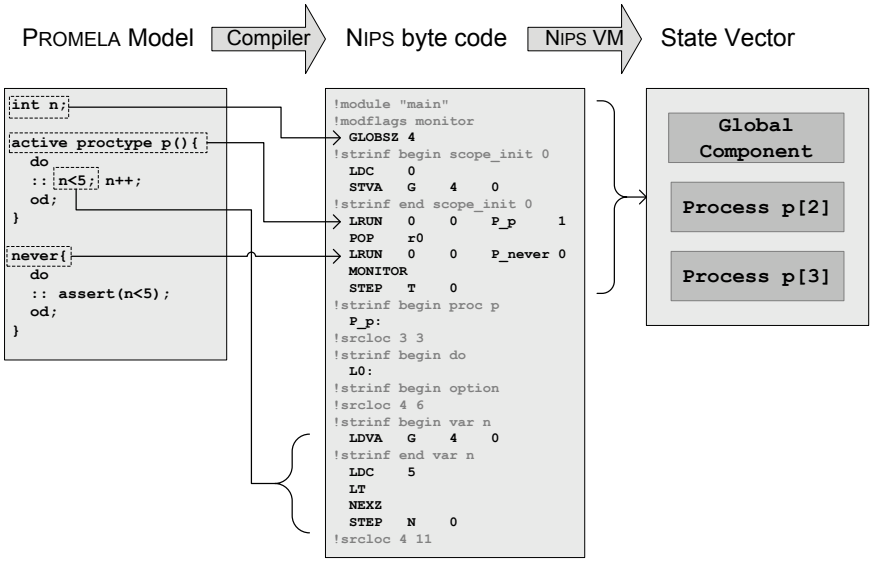



Figure 2.1: NIPS Byte Code Example

Figure 2.1 shows an example byte code snippet compiled from the PROMELA source program on the left. Its source statements can be deduced from the byte code, but the location of variable

[1] Modeled by interleaving semantics

$n$ in the memory is not saved with its source name. Therefore, "variable $n$" is unknown in the resulting state vector on the right.

The *Static Code Optimization* for NIPS byte code described in [2], that works for all compilers targeting the NIPS VM, can in some cases improve the performance substantially. In such cases the amount of byte code and the state space can be statically reduced.

### 2.1.3 Design

The NIPS VM was designed using pragmatic design solutions. First, a formal semantics was written that completely describes the model behavior for each of the PROMELA language features [42]. This formal description was then used to derive the NIPS byte code instructions that are as generic as possible, in order to be able to reuse them for describing the operational semantics of different languages [41]. The NIPS VM was designed to create a model that is simple, efficient and embeddable as a component into host applications [43]. Conceptually the design of NIPS VM based tools is split in the NIPS VM *back-end* for state successor calculation, a *scheduler algorithm* that determines the next state to evaluate and a *compiler* that targets the NIPS VM. The VM makes use of a stack-based architecture for expression evaluation. It has registers for the translation of counting loops. The RISC-like instruction set is motivated by the need for fast decoding inside the instruction dispatcher, the VM's most executed routine. As a design principle the NIPS VM executable remains the same for each model and is not recompiled for specific models as happens with SPIN.

**States.** NIPS VM states are memory BLOBS, untyped sequences of bytes called state vectors. States contains all the information the VM needs to continue its execution. During the execution of a process *step*, the process contains execution stacks and registers but these are removed before the state vector is returned. The NIPS state vector starts with a *global component* that contains global variables, followed by *processes* that contain local variables and *channels* that may contain messages up to their predefined capacity. All compilers for the NIPS VM that support components or objects that do not fit global variables, processes or channels precisely must encode these objects as global or local variables. The order in which the components are placed in the state vector is referred to as the *state format*. Figure 2.2 shows the NIPS VM state format.

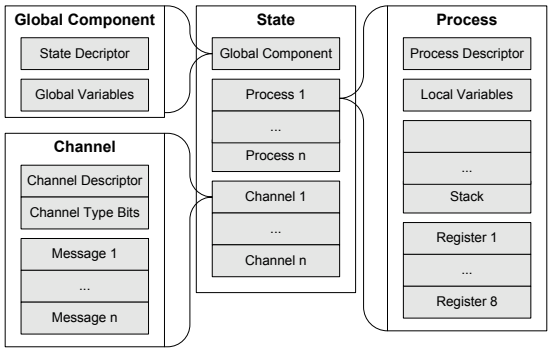



Figure 2.2: NIPS State Format

| variable | describes |
|---|---|
| gvar_size | size of the global variables in bytes |
| process_count | number of processes in the state vector |
| exclusive_pid | pid of the exclusively executing process |
| monitor_pid | pid of the process used for monitoring |
| channel_count | number of channels in the state vector |

(a) Global Component Descriptor

| variable | describes |
|---|---|
| pid | process identifier |
| flags | process execution mode |
| lvar_size | size of the process' local variables |
| pc | program counter |

(b) Process Descriptor

| variable | describes |
|---|---|
| cid | creating process identifier and channel identifier |
| max_length | maximum number of messages in the channel |
| cur_length | current number of messages in the channel |
| msg_length | message length |
| type_length | type preamble length |

(c) Channel Descriptor

Figure 2.3: NIPS VM State Component Descriptors

Each component starts with a *component descriptor* that describes component state information relevant to the execution of the VM. The information in the descriptors is relevant to the placement of the component in the state vector and the retrieval of the component from the state vector. The NIPS component descriptors are depicted in Figure 2.3. Processes are ordered inside the state vector by increasing value of the *pid* and channels are ordered inside the state vector by increasing value of the *cid*. Even though processes are not explicitly typed, depending on the type of the process the *pc* stays within the range of the section of the byte code.

Channel identifiers contain the creating process identifier as means for a simple symmetry reduction. The order in which channels are created by different processes does not lead to different states. Furthermore, a NIPS channel component is always the same size, it is padded with zeros.

For each instruction, the component subject to the instruction must be retrieved from the state vector. This is done by an algorithm reminiscent of a scanner. It uses a component descriptor look-ahead to identify the next component in the state vector. The type of the component and thus the type of the descriptor is determined by the state format. Algorithm 1 describes the retrieval of NIPS VM state components from the state vector.

**Algorithm 1** (NIPS Component Retrieval)**.** *retrieve*$(sv, comp, id)$ is a function to retrieve component *comp* with component identifier *id* from state vector *sv* that starts at **Global** where

- Argument *sv* is a state vector, argument *comp* is a NIPS component where comp $\in \{global, process, channel\}$ and argument *id* is the process identifier if $comp = process$, the channel identifier when $comp = channel$ or not defined if $comp = global$.
- Local variable *process_count* is the number of processes in the *sv*, Local variable *channel_count* is the number of channels in the *sv*, Local variable *curp* is the current number of visited processes and Local variable *curc* is the current number of visited channels.
- And let functions $size(global\ g) = size(global\_descriptor) + g.descriptor.gvar\_size$, $size(process\ p) = size(process\_descriptor) + p.descriptor.lvar\_size$ and $size(channel\ c) = size(channel\_descriptor) + c.descriptor.type\_length + c.descriptor.cur\_length * c.descriptor.msg\_length$ be help functions.

1. **Global**: The global component *g* is at offset zero.
   (a) If the object of the search is the global component, i.e. $comp = global$ return *g*.
   (b) Otherwise read the global component descriptor. Save the number of processes and channels in *process_count* and *channel_count*. Set the number of visited processes and channels to zero: $curp = 0$ and $curc = 0$. Goto process at offset $size(g)$.
2. **Process**(**int** *o*): Process *p* is a component at offset *o*. Increment the number of visited processes: *curp*++. Read the process component descriptor.
   (a) If the process was found then $comp = process$ and $id = p.descriptor.id$ then return process *p*.
   (b) Else if there are more processes i.e. $curp < process\_count$ then goto **Process** at offset $size(p) + o$.
   (c) Else if there are more channels i.e. $curc < channel\_count$ then goto **Channel** at offset $size(p) + o$.
   (d) Else terminate, component not found.
3. **Channel**(**int** *o*): Channel *c* is a component at offset *o*. Increment the number of visited processes: *curc*++. Read the channel component descriptor.
   (a) If the channel was found then $comp = channel$ and $id = c.descriptor.id$ return channel *c*.
   (b) Else if there are more channels i.e. $curc < channel\_count$ then goto **Channel** at offset $size(c) + o$.
   (c) Else terminate, component not found.

### 2.1.4 Applications

**Promela Semantics.** The Nips VM is particularly well-suited for Promela models because Nips byte code has been developed to express the formal semantics of Promela models [42, 41, 43]. The efforts are implemented in the Nips Promela Compiler. Together with the Nips VM and a scheduler component, it provides functionality comparable to that of the Spin Model Checker that fast enough for practical use, although a debugger is missing[2]. The goal is to reuse Promela in order to be able to reuse case studies and be interchangeable with tools that use Promela.

**Schedulers.** The Nips VM distribution contains built-in DFS and BFS schedulers. The algorithm for nested DFS described in [44] has been implemented in Java to gain insight into the algorithm.

**Adaptive Model Checker.** The Nips VM has been used as a state-space generation component in an adaptive external-memory model checking tool [17]. The tools scheduler uses not only the main memory but also the hard drive to store the state space, making it possible to model check Promela models with larger state spaces.

**Nips and DiVinE.** The Nips VM has been used with DiVinE [3] in *distributed algorithms* for verification, in which multiple scheduler algorithms run on different PCs. By letting each scheduler perform a BFS the state space is partitioned and stored distributively. Like the external-memory model checker it makes it possible to use more memory for model checking Promela models.

**Model Checking Embedded System Software.** The Nips VM has been used to model check correctness of assembly code for ATMEL ATmega family of micro-controllers [40].

**Tapir.** Tapir is a programming language designed for *systems programming*. It is a minimalistic object oriented language which has no automatic memory management, no exception handling, no inheritance and no type-casts. Its domain, systems programming, includes networking protocols, operating systems, middlewares, DSM systems, etc. The services provided by such systems are critical for the stability of the programs that rely on them. Therefore the semantics of Tapir is modeled using Nips byte code. A model checker has been implemented that uses the Nips VM which provides a means to check the correctness of the system [46].

## 2.2 Debugging Information Formats

Over the years many debugging information formats for programming languages have been used such as *stabs* [28] , *COFF*, *IEEE-965* [47] (a withdrawn standard) and Dwarf [15]. Debugging information format standards are either combined with object file formats (COFF, IEEE-695) or separately described (stabs, Dwarf) to be used in combination with an object file format. An important example of such a format is the Executable and Linking Format (ELF) which is a standard Unix object file format. ELF largely replaced the Common Object File Format (COFF). The Java programming language uses its own format, called the Java `class` file format, to store both byte code and debugging information. Java virtual machines require part of the debugging information to run Java programs whereas the rest only serves for debugging. In this section we discuss the stabs, Dwarf and the Java `class` File debugging information formats as

[2] See Chapter 5 for the Nips Debugger.

candidates for use with the Nips VM.

### 2.2.1 Stabs

The *stabs* (symbol table strings) debugging information format was originally used with Unix's `a.out` (assembler output) object format for executables, but has been extended over the years for use with Cobol, C, C++, Pascal and other languages. Problematic with the stabs format is its standardization, with some exceptions [28, 29] stabs have not been properly documented. Compilers that support stabs, such as the GNU Compiler Collection (GCC), can generate the debugging information encapsulated in so called *assembler directives* known as stabs, formatted information strings, which are interspersed with the generated code. The assembler adds the information from stabs to the symbol information it places in the *symbol table* and the *string table* of the object file it builds. The linker combines the object files into an executable such that it contains one symbol table and one string table. The resulting linked object or executable can be parsed by a debugger on the same platform, as a source of debugging information about the running program.

**Language.** A documented version of stabs used with the GNU Debugger [28], describes the language that consists of three differently formatted stab assembler directives called *string* (`.stabs`), *number* (`.stabn`) and *dot* (`.stabd`).

```
.stabs "string", type, other, desc, value
.stabn type, other, desc, value
.stabd type, other, desc
```

The *type* field is a number which uniquely determines the stabs type. The stabs type defines the exact interpretation of, and possible values for, any remaining *string*, *desc*, or *value* fields present in the stab. The overall format of the string field for most stab types is:

```
"name:symbol-descriptor type-information"
```

The field describes the *names* of *symbols* and their *type*. Stabs symbols include the: (stack) variable, constant, nested name, (nested) function or procedure, reference or register parameter, module, an enumeration or an array. Stabs type supports includes: built-in (base-), method-, pointer-, reference-, array-, function-, structure-, set- and union- types. Stabs may also describe unnamed entities.

### 2.2.2 Dwarf

The Dwarf[3] debugging information language acronym is said to mean Debugging With Attributed Record Formats [10]. There are three documented versions, the first of which was first used with the *sdb debugger* in Unix System V Release 3 (SVR3) developed by Bell Labs in the mid 1980's. It was first documented by the Programming Languages Special Interest Group (PLSIG), part of Unix International (UI), in 1989 as the Dwarf 1 standard [32] and is still used for debugging small embedded systems processors. Dwarf 2 was introduced as a draft standard [33] in 1990 but a final version was never released. It addressed issues related to the amount of generated data and introduced support for C++. Dwarf was revived in 1999 in order to provide better debugging support for the HP/Intel IA-64 Architecture as well as better documentation of the Application Binary Interface (ABI) used by C++ programs[4]. The Dwarf 3 standard

[3]The name Dwarf is a funny reference to ELF.

[4]An ABI allows compiled object code to function without changes on any system that supports the ABI.

was released in January 2006 [15, 1]. It is backwards compatible with DWARF 2 and therefore resembles it closely. It adds support for Java, C name spaces, C99 base types, cross module entry reference, discontinuous scopes, stack structures and stack unwinding.

**Language.** DWARF 3 is designed to be extensible to support procedural programming languages on any machine architecture. DWARF uses a series of debugging information entries to define a low-level representation of a source program. It is commonly used with ELF but it can be used with other object file formats as well. It does not duplicate information located in the object file.

DWARF is block structured, like many programming languages. The DWARF description of a program is a tree structure which resembles an AST. DWARF tree nodes represent *types*, *variables* or *functions*. The basic descriptive entity in DWARF is the Debugging Information Entry (DIE). Each entity, except the top-most entity, is contained in a *parent* entity and may contain *child* entities. Entities may contain multiple entities called *siblings*. Each DIE has a tag, which specifies what the DIE describes and a list of attributes which fill in details and further describes the entity.

There are a vast amount of DWARF entries, e.g. used for describing: functions, procedures, lexical blocks, labels, statements, error handling, sets, built-in (base-) types, pointer-types, array-types, structure-types, union-types, class-types, interface-types, and member-function types. Furthermore, DWARF contains *instructions* to describe call frame information and to provide a mapping between target code and the program source.

### 2.2.3 GDB

Arguably the most popular debugger for UNIX systems is GDB, the GNU debugger. It is a source-level debugger that can be run on most UNIX variants and Microsoft Windows that allows debugging of programs written in C, C++, Objective-C, Pascal, Java, Fortran and Modula-2, etc. GDB can display variable values and can be used to determine where in the execution errors occurred. It can be used to set *break-points*, which entails selecting a source line where the execution op the program should halt and it can be used to *step* through the code line by line or instruction by instruction. GDB can read various debugging information formats that are output by the GNU Compiler Collection (GCC), including stabs and a modified undocumented modified version of DWARF 2. It is however difficult to extend GDB with support for new languages, because the requirements are not clearly described and it requires an extensive amount of programming [39].

### 2.2.4 Java `class` File Format

The Java virtual machine `class` file format describes the Java byte code structure. Each `class` file consists of a tree structure. Its nodes are described as *tables* that consist of zero or more variable-sized items. The Java class format is extensible because all tree nodes may have any number of *attributes*, general information items, associated with them. Compilers are permitted to define and emit `class` files containing new attributes in the attributes tables of `class` file structures. Java virtual machine implementations are permitted to recognize and use new attributes, e.g. to support vendor-specific debugging, provided that these attributes do not affect the semantics of `class` or `interface` types. Unrecognized attributes must be silently ignored,

allowing byte code with additional attributes to run on different implementations of the Java VM also [25].

The root of `class` tree, which is represented by the `ClassFile` structure, contains the `Constants`, `Fields`, `Methods` and `Attributes` tables. The class format tree structure is schematically depicted in Figure 2.4. We discuss the tree nodes one-by-one.

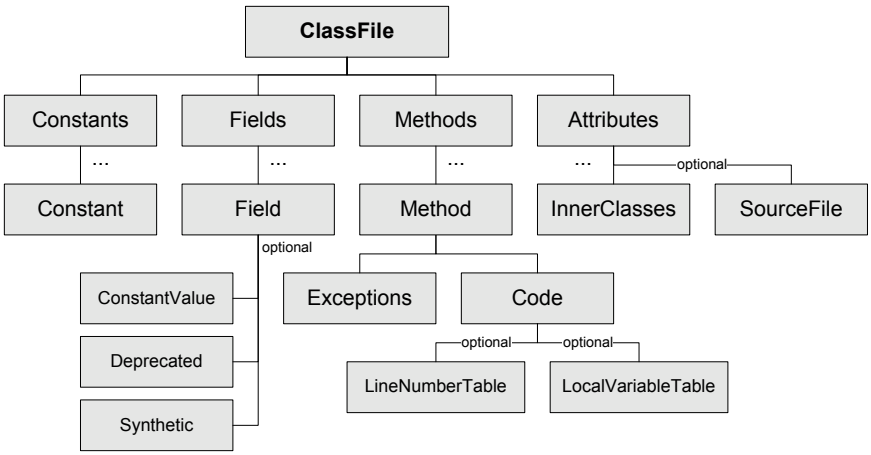



Figure 2.4: Java `class` Format

**Constants.** The `ConstantPool` is a table of entries that among other things, represents a flat representation of the compiler symbol table. Constants in the `ConstantPool` are referred to by other tree nodes by array index number. The format of each entry is indicated by the *tag*, the first byte in the constant entry. Table 2.1 depicts the possible constant types and their tag value. A full description of constant types is given in [25].

| constant type | tag value |
|---|---|
| `Class` | 7 |
| `Fieldref` | 9 |
| `Methodref` | 10 |
| `InterFaceMethodRef` | 11 |
| `String` | 8 |
| `Integer` | 3 |
| `Float` | 4 |
| `Long` | 5 |
| `Double` | 6 |
| `NameAndType` | 12 |
| `Utf8` | 1 |

Table 2.1: `class` format constants

**Fields.** The `Fields` node is a table of entries that represent the fields declared for this `class` or `interface`. A *field* entry contains the *name*, *type* and *flags* that indicate if the field is *public, private* or *static*. For fields that are declared as constants the `ConstantValue` attribute is used to store the constant value. The `Deprecated` attribute is used to indicate that a field is deprecated. The `Synthetic` attribute is used to indicate that a field has been generated by the compiler.

**Methods.** The `MethodInfo` node describes the methods declared by this class or interface type, including instance methods, `class` (static) methods, instance initialization methods, and any class or interface initialization method, but no inherited methods from superclasses or superinterfaces. A *method* entry contains the *name* of the method, the *type* of its parameters and of its return value, *flags* that indicate if the method is *public, private* or *static*. The `Exceptions` attribute contains the names of the exceptions that can be thrown by the method. The method field `Code` attribute contains the code of the non abstract methods.

The `LocalVariableTable` attribute and the `LineNumberTable` are optional attributes of the `Code` attribute. The `LocalVariableTable` attribute may be used by debuggers to determine the value of a given local variable during the execution of a method. The `LineNumberTable` attribute may be used by debuggers to determine which part of the Java virtual machine code array corresponds to a given line number in the original source file.

**Attributes.** The `Attributes` node is a table of *class* attributes. The optional `SourceFile` attribute refers to the name of the source file. The `InnerClasses` attribute describes nested classes and interfaces. This concludes the description of the Java `class` file format.

**Java VM.** The `Code`, `ConstantValue`, and `Exceptions` attributes must be recognized and correctly read by a class file reader for correct interpretation of the class file by a Java virtual machine implementation. Furthermore, the `InnerClasses` and `Synthetic` attributes must be read correctly in order to support the Java and Java 2 platform class libraries. These libraries include support for reflection, loading and creation of classes and interfaces (e.g. the `ClassLoader`), linking and initialization of a class or interface, security and multi-threading. Additionally, the `LineNumberTable`, `LocalVariableTable` and `SourceFile` attributes of a class file and its methods are needed to support source-level debugging.

## 2.3 Model Checkers

In this section, explicit state model checkers that bear resemblance to the NIPS VM are discussed with respect to the motivation, the technologies and the modeling languages. The focus is on the debugging solutions in particular, since counter-examples produced by the tools must be returned to the user at a source level and this has yet to be achieved with the NIPS VM.

### 2.3.1 Spin

SPIN is an explicit state model checker that can be used for the formal verification of distributed software systems. The SPIN acronym, introduced with the first version of the tool, means Simple PROMELA Interpreter. Nowadays SPIN is neither simple nor merely an interpreter, it is a mature model checker that has been successfully applied in many software development projects. SPIN is thoroughly described in [20].

**Motivation.** The motivation for SPIN is to verify the correctness of the design models of concurrent and distributed systems which are difficult to debug and test with traditional means.

**Language.** PROMELA means Process Meta-Language. It is a high level modeling language intended to find good abstractions of system designs (i.e. models) of concurrent software systems. Its commands are guarded which means statements are blocking (will not execute) until they evaluate to true. It features asynchronous *processes* to model threads, buffered and unbuffered

*channels* for communication between processes with message passing, synchronizing statements, non-deterministic control structures, conditional branching statements, loops, atomic sequences, deterministic steps, structured data, etc. The full language features and their syntax are too numerous to explain here, they are described in detail in [20].

**Tool.** SPIN can perform a depth first search (DFS) or a breadth first search (BFS) to validate LTL properties, which are translated to `never claims` in the PROMELA model. To check invariants, assertions can be placed in the model at strategic locations in the model. During a DFS SPIN uses a stack to keep track of visited states. Once a counter-example has been found the stack is simply printed. It is returned to the user as execution trace that consists of a sequence of states and transitions, which are described using source level variables and values and PROMELA source statements. SPIN keeps track of the source statements and variable types with internal look-up tables. It does not have to resort to the use of separately defined debugging information to explain counter-examples. However, the symbol table information that is collected by SPIN while parsing a PROMELA model can be emitted to a file in a machine readable format.

**Debugging Information Format.** The SPIN debugging information format uses one line to describe each symbol table object. Similar to stabs, each line is a string that consists of information *fields*. All objects are described by type and name.

```
type name ...
```

Depending on the type of the object additional information is specified.

For a variable of type *bit, byte, short, int, mtype* and *proctype* the additional information fields are the initial value, the *scope* and its *role*.

```
type name initial_value scope role
```

The scope of variable is global, part of a struct field or local to a process. A variable either plays the role of a variable or that of an argument. The values that the type, scope and role field can range over are depicted in Table 2.2. If the variable is declared as an *array* the name field also contains the array length: `name[length]` where the length is a positive integer. If the variable is a structure field then its name is composed of the type name and the field name separated by a dot: `type_name.field_name`.

| **type** | **scope** | **role** |
|---|---|---|
| `bit` | `<:global:>` | `<variable>` |
| `byte` | `<:struct-field:>` | `<argument>` |
| `int` | `<process name>` | |
| `mtype` | | |
| `proctype` | | |
| `struct` | | |
| `channel` | | |
| ... | | |

Table 2.2: PROMELA Debugging Information Field Values

For structure variables, where the type is `struct`, the third field defines the name of its structure type declaration.

```
struct name type_name scope role
```

Structures can be structure-fields themselves in which case its name is composed of the type name and the field name separated by a dot: `type_name.field_name` analogous to variables that are structure fields.

For a `channel`, the additional information fields are the size, the scope, the role, the number of message parts and the message part types. The size is an integer value greater than or equal to zero, the number of of message parts is an integer value greater than zero, and the message type parts are not struct types but single-value types.

```
channel name size scope role msg_parts type_1...type_n
```

**Optimization Strategies.** Many optimizations have been made to SPIN to reduce the state space, such as *state collapsing*, *state compressions* [19] and the *minimized automaton* [21]. The state space explosion is curbed by means of Partial Order Reduction [20, 6]. Spin also supports approximate model checking where *bit state hashing* or *hash compaction* are used store the state space.

### 2.3.2 JPF

Java Path Finder (JPF) is a second generation explicit state model checker tool environment for the verification, analysis and testing of Java programs, or Java models as they are referred to in this context. The original version of JPF translated Java programs to PROMELA. This approach was abandoned because of the difficulties of mapping the language features of Java to PROMELA. The current version uses a virtual machine based approach and runs Java byte code.

**Motivation.** The design of JPF is motivated by practical language and tool design arguments [48]. Programs often contain fatal errors, such as critical section and deadlock errors, despite the existence of careful designs. Therefore there should also be focus on model checking real programs written in modern programming languages. Because modern programming languages are well designed, they may be good design modeling languages as well. Using programming languages as modeling languages may increase the access to real examples and case studies. It may also enlarge the number of potential users of the tool.

**Tool.** JPF is designed as an extensible tool environment for model checking and debugging, which is as modular and easily understandable to developers as possible at the cost of model checking speed. The tool, which his implemented in Java, simulates non-determinism by using *backtracking*, such that JPF can restore previous execution states, which are heap and thread-stack snapshots, in order to explore different previously unexplored execution paths. The JPF VM runs on a Java VM. It intercepts method invocations and delegates property-irrelevant executions to dedicated classes running the host VM. JPF can only exhaustively explore the state space for small programs, which should be no greater than approximately 10000 lines and may not contain platform specific *native* code, because the JPF VM cannot execute it. Aside from model checking, JPF offers debugging functionality in the form of *run-time analyses* such as data *race condition detection* and *lock order analysis*, that can be used to pinpoint potentially problematic code fragments, deadlocks and unhandled exceptions.

**Explaining Counter-Examples.** To understand how JPF counter-examples can be related to the Java source it is necessary to explain what JPF states are. States are represented in a complex Java data structure which consists of three components: **1)** information for each thread in the Java program, **2)** the static variables (in classes) **3)** and the objects in the system. The information for each thread consists of a stack of frames, one for each method called, whereas the static and dynamic information consists of information about the locks for the classes/objects and the fields in the classes/objects [48].

The state space size is reduced by using a generalized *collapse compression* where state components are stored in separately indexed spaces (hash-tables) and states are represented by indexes referring to component hash-table positions. The states are compressed losslessly and stored in a single integer, making it very fast to compare states.

Explaining a counter-example in JPF entails uncollapsing the integer state vectors and using reflection, a Java language feature, to view run-time models and relate the states to source level information. The java byte code contains debugging information which is used to support reflection at run-time, as is explained in Section 2.2.4. JPF is integrated with BANDERA for its error-displaying capabilities which allow users to step through the code line by line, forwards and backwards, while observing the objects in the memory.

**Optimization Strategies.** To improve its scalability and allow use with larger programs JPF supports *directed model checking* using a heuristics search. The set of follow-on states is filtered according to a property related relevance directs the JPF VM, which is either supplied by the user or gained from a *run-time analysis*. With this inexhaustive model checking the tool does not provide a proof of correctness with respect to a system property, but rather serves as a *debugging tool* (that may also yield false positives).

JPF uses *symmetry reductions* for *class loading* and the *heap*, where the ordering of the classes being loaded and the order in which dynamic objects are created in the heap are canonicalized, such that different orderings are recognized as equivalent. A mark-and-sweep garbage collection algorithm prevents states from growing indefinitely and facilitates heap symmetry reduction.

The *static analyses*: *slicing* and *partial evaluation* are used to generate smaller functionally equivalent programs that result in a smaller state space. A different static analysis called *partial order computation* does not change the program size but reduces the state space by eliminating unnecessary interleavings of the program behaviour.

### 2.3.3 Bogor

*Bogor* is a modular model checker framework with an extensible input language for defining domain specific constructs. Its modular interface is designed to ease the optimization of domain-specific state-space encodings, reductions and search algorithms [38, 8, 37, 18].

**Motivation** Bogor was developed as an answer to difficulties with the application of existing model checker frameworks to particular domains [38]. The authors observed that it is problematic to apply existing model checker frameworks to domain specific model checker problems because it is difficult to: **1)** efficiently map domain constructs to model checker input languages; **2)** change the encoding of system data to achieve state space reductions; **3)** configure the search mode of the model checker based on the reasoning with the particular model checking problem; **4)** alter or combine state space reduction strategies with collections of reduction strategies in order to target related model checking problems.

**Language.** The input language of Bogor is an extended version of the *Bandera Intermediate Representation* (BIR). BIR features language constructs for types, expressions and actions. Unlike most model checker languages, BIR features include dynamic thread creation, object creation, exception handling, virtual functions and recursive functions. BIR can express all features of the Java Programming Language. Bogor can be extended with new *semantics primitives*. An extension declaration consists of a signature declaration which specifies new symbols and associated number of arguments to the name-spaces for types, expressions and actions, and the

name of the Java package that implements the semantics of the extension. Extensions do not alter the BIR grammar, but add names to the set of built-in expressions, actions, etc [38].

**Tool.** Bogor is designed to make it easy to *extend* the model checker with *new semantic primitives* and optimize the *state-space-storage* and *exploration strategies* in order to tailor it towards different application *domains*. Bogor is developed as a plug-in for the Eclipse platform. The type of properties supported by the tool are invariants and LTL properties. The state space can be explored with a DFS, BFS or directed search. The tool offers and editor with syntax highlighting, a well-formedness checker, a user simulation mode and counter-example display. Just like Spin, Bogor keeps track of the source statements, line numbers and variable types in an internal symbol table. It does not have to resort to the use of separately defined debugging information in order to depict counter-examples. In line with Bogor's modular design, the symbol table may be extended to support new actions and expressions.

**Uses.** Bogor is used for *education* in a graduate-level model checking course at (among others) Kansas State University [9]. Because its design aspects are functionally separated and it is programmed in Java, students and professional developers alike can easily add functionality to Bogor.
Bogor has been used to model check Java programs [9]. The prototype Bytecode-to-BIR Translator (BB) translates Java source to BIR. The BB translator makes use of the ASM Framework [14], a Java class manipulation tool designed to dynamically generate and manipulate Java classes, which are useful techniques to implement adaptable systems. ASM is not an acronym but refers to C assembly code functions. The debugging information is saved inside the Java byte code as discussed in Section 2.2.4.

## 2.4 Evaluation

In Section 1.1.2 we stated that debugging information is the extra, optional information generated by the compiler which is usually not necessary for programs to run, but which is necessary for debuggers to make source-level debugging possible.

In this section we evaluate the debugging approaches used with Spin, Bogor and JPF in Section 2.4.1. In the design of each of the tools, except the Nips VM, the problem of relating counter-examples back to the source has been solved. To what extent do approaches used with the explicit state model checkers related to the Nips VM, discussed in Section 2.3, make source-level debugging possible for the Nips VM? Furthermore, we evaluate the debugging formats, discussed in Section 2.2, in Section 2.4.2 in order to reach a conclusion about an debugging approach for Nips.

### 2.4.1 Model Checkers

**Modularity & Extensibility.** What Bogor, JPF and Nips have in common is that they all aim to be easily extensible with new algorithms and standardize model checking solutions by offering an extensible explicit state model checking tool set. This is done by modular design, separating the tools into easily replaceable components. JPF and the Nips VM both employ schedulers to direct the state space evaluation. Components can be reused without the need for a lengthy design process.

**Source Language.** In contrast to SPIN, Bogor and JPF which use high level modeling languages which contain variable and type information, the NIPS VM uses the low level NIPS byte code as its input language which may be derived from a PROMELA model but which contains almost no source-level elements. The debugging information strings the NIPS byte code contains will not be extended because they are tangled with the byte code and they do not support a debugging API. SPIN parses high level PROMELA models itself and keeps track of the source statements and types using internal look-up tables. It has no need for separately defined debugging information to show counter-examples to the user.

Although NIPS byte code and BIR are both intermediate representations, in the design of Bogor there is no need for separate debugging information. Like PROMELA, BIR contains the source level elements that NIPS byte code lacks. The debugging approaches used with SPIN and Bogor cannot be used because the source level elements are not described separately from their respective modeling languages. Consequently, if we wish to either of the approaches, the modeling language associated with it must be reused as well.

The design of JPF is similar to that of the NIPS VM in that both are virtual machines that run byte code and both can be used to model check programming languages. JPF uses Java as its modeling language, is written in Java itself and runs on a Java virtual machine. The JPF approach to show counter-examples to the user is to use the debugging information stored in the `class` files generated by a Java Compiler from a Java model.

A comparison of the discussed modeling languages is shown in Table 2.3. Each of the tools use different languages, some of which are source level modeling languages and some are intermediate representations of models. What the tools have in common is that debugging information is never separately described at a high level.

| **tool** | **modeling language** | **low level language** | **debugging information** |
|---|---|---|---|
| SPIN | PROMELA | none | PROMELA |
| JPF | Java | `class` file format | `class` file format |
| Bogor | BIR | none | BIR |
| Java + BB compilers | Java | BIR | `class` file format |
| NIPS VM based tools | PROMELA, TAPIR, etc. | NIPS byte code | *to be decided* |

Table 2.3: Modeling Languages

**Debugger Design.** Bogor makes use of the Eclipse platform to show counter-examples to the user. The NIPS VM would benefit from a user interface that can embed the NIPS VM as an Eclipse plug-in. JPF makes use of the reflection language feature of Java. The NIPS VM would benefit from a reflective layer that can be used for meta-programming and supports introspection and intercession not only for immediate debugging purposes, but also for dynamic partial order reduction in the long run.

### 2.4.2 Debugging Information

Is there a debugging information format, discussed in Section 2.2, that is suitable to be used with the NIPS VM in order to provide general, reusable debugging support? If not, then what can be learned from the approaches and formats?

Since NIPS byte code designed is not just designed to describe the semantics of PROMELA,

but also different modeling languages, the debugging information should be tailored to support common modeling structures and data types. More generally, it should focus on the abstract meta-modeling of the memory encoded by state vectors that are produced by explicit state model checkers and the source level elements of the used modeling languages. This includes mainly the types of static structures of Nips run-time components and source information. We prefer a high level debugging information language over a low level description because the goal of the language is to be used for modeling the memory of running programs. Therefore the design of the language should be separate from the low level implementation or object file format. The debugging information language must be clearly defined and not a vague standard of which the specification is adjusted for various projects without documentation. It should not be the case that implementing a debugging information format costs an excessive amount of effort in terms of programming. The different debugging information formats are discussed with respect to these requirements.

**Stabs.** The stabs format consists of formatted Strings of type information. Though stabs is extensible to support modeling languages a problem with stabs is that it is not standardized and can be complex and occasionally cryptic [10]. Therefore stabs will not be used with Nips.

**Spin.** Like stabs the Spin debugging information consists of formatted Strings of type information, but it is not extensible. It can only define symbol table information that can be extracted from Promela models. Although it is simple and could easily be documented in Section 2.3.1 by reverse engineering the format, the Spin debugging information is not sufficiently general to describe run-time memory models of programs in languages other than Promela and will therefore not be used.

**Dwarf.** In contrast to stabs and Spin debugging information, Dwarf debugging information is block structured, such that each entry (except the top level entry) is contained within another entry and trees of information are easily represented at run-time. The Dwarf documentation [15] describes in detail how to support different facets of debugging such as describing data types, but also memory location expressions and source location mappings. We favor Dwarf over stabs as because of its uniform standard, its block structure, its clear specification and its expressiveness. However providing complete support for Dwarf for the Nips VM is difficult and unnecessary, because Dwarf is tailored towards procedural programming languages and not towards modeling languages, such as Promela, that are used with the Nips VM.

**GDB** An implementation of a debugger for Nips in GDB is deemed impractical because of the features it supports and the large amount of programming required to add a new language to it [39]. We do not wish to restrict Nips to Promela and we do not wish to have to extend GDB's feature set with non-deterministic languages.

**Java `class` file format.** The Java `class` file format is used by JPF and by the Bogor Bytecode-to-BIR compiler as an intermediate representation of models and to store debugging information. The format is well documented and extensible via the using new attributes [25]. However its description is strongly tied to the implementation of Java virtual machines and the Java programming language and therefore its design is not high level or easily reusable. The format is not block structured as Dwarf.

## 2.5 Concluding Remarks

In this chapter we have presented debugging approaches that are related to this thesis work. The Nips VM, debugging information formats and debugging approaches for model checkers were introduced and evaluated. The comparison of debugging approaches used in Spin, Bogor and JPF yielded no solution for Nips because of architectural differences and differences in the modeling languages. The study of the debugging information formats also did not show a precision fit candidate. Therefore a new debugging information languages is necessary which is similar in structure to Dwarf, using a block structure and extensible attributes, but which is more high level and can be used for explicit state model checking. The language introduced in this report is called Static Debugging Information (SDI). It is described in Chapter 3.

# Chapter 3

# The SDI Language

In this chapter, a description of the Static Debugging Information (SDI) language is given. It facilitates a debugging API through which the access to source program variables, their types and values, and location information is realized. The SDI language is part of the SDI Framework, which will be described in Chapter 4.

The structure of this chapter is as follows. Firstly, the creation of the SDI language is motivated in Section 3.1. The SDI memory model is described in Section 3.2 and the list notation of the language is described in Section 3.3. Next, the SDI notation for variable and scope declarations is described in Section 3.4, types are described in Section 3.5 and locations are described in Section 3.6.

## 3.1 Introduction

Debugging information is commonly not described at a high level, but rather it is a low level byte format integrated with object files, such that its representation is tied to the implementation of a storage format. In explicit state model checking, debugging information is usually stored internally in a symbol table specific to a modeling language, such that the information cannot easily be exchanged with other tools. When source models are represented by an intermediate language for an operational semantics, source level debugging information may be lacking. The SDI language is designed to remedy these problems. The main motivation for its creation is that it can represent information known at compile-time that can be used for debugging at run-time.

SDI is used to build a *memory model* at run-time which relates otherwise unreadable binary state vectors back to source-level variables, names and types. This memory model provides a means for source level debugging. SDI is used to describe a flattened symbol table, run-time component information and source language constants.

The SDI language is described with entries and attributes that are written down using a list notation. Entries can describe symbol table entries, run-time component types and locations. They contain attributes which are a constant name and constant value. Some attributes have a special meaning in an entry and must always be defined whereas other attributes are optional. Attributes that do not have a predefined meaning may be added to static debugging information

to be queried at a later time. Debugging information of which the value is only known at run-time is referred to as run-time debugging information. We avoid the term dynamic information because SDI does not support dynamically typed variables, variables of which the type is generically declared and is only precisely known at run-time. This chapter describes the different SDI entries and for each of them describes the required attributes and their meaning.

## 3.2 Memory Model

This section describes what a program memory model is, how it can be described by SDI variables and types and where the SDI definitions used to construct the model come from.

A run-time state of a program consists of *components*. Each of these run-time component has a *descriptor*, a part of the component that contains variables that describe what is in the component or meta-information that describes the run-time state of the component. Component descriptors are described in more detail in Section 4.4. In order to understand how a memory model describes run-time components and what a memory model expresses, we first explain the different sources of information for a memory model of a running program.

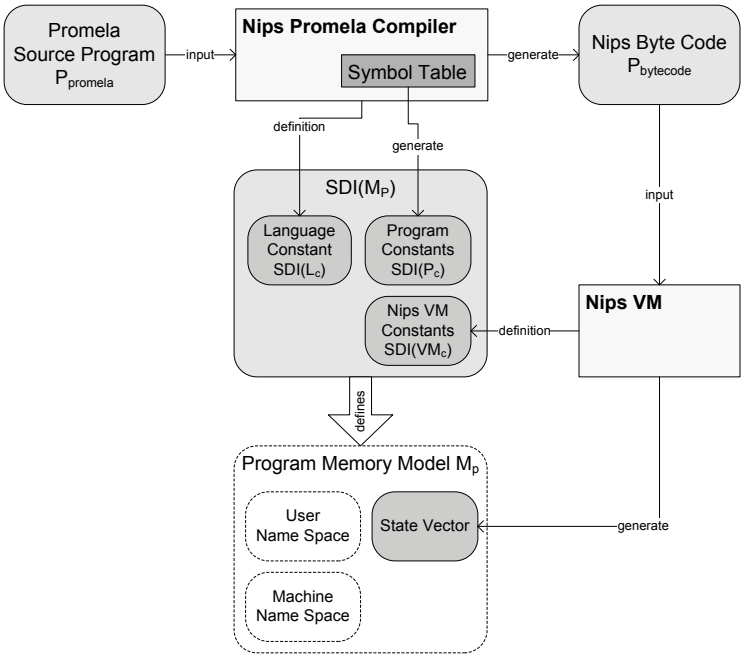



Figure 3.1: Nips Memory Model

In the context in which we are using SDI we describe three information sources. The first source is the *program* for which the memory model is constructed. The second source is the *language* the program is written in. The third source is the *machine* the program will run on and the ultimate supplier of the memory model. Figure 3.1 describes the sources of information for the Nips memory model. Definition 1 describes the memory model of a program that runs on a

virtual machine.

**Definition 1** (SDI definition of a memory model $M_P$ for a program $P$)**.** We assume there exists a compiler $C$, a high level language $L$, a program $P$ written in $L$, a low level byte code representation $P_B$ generated by $C$ from $P$ and a virtual machine $VM$ which runs $P_B$. We assume it is possible to get snapshots of the memory of the running program we call state vectors $sv$. The $VM$ and manages the memory of the running program $sv$ as a sequence of bytes. The memory model $M_P$ is a model of the memory of the running program $sv$, which consists of named and typed variables and their values.

1. For each compiler $C$ that takes language $L$ as input there exist language constant types $L_c$ which are predefined in SDI that are the same for each program.

2. For each program $P$ there exists program constant debugging information $P_c$, which can be written down as SDI definition, that describes the variables in the memory model $M_p$ of the running program. This information is described in the symbol table of the compiler $C$ that translates $P$. The program SDI $P_c$ is a flattened representation of the symbol table.

3. For each virtual machine $VM$ there exist virtual machine constants $VM_c$, that describe machine related information in the memory model $M_p$. The VM environment variables describe the system state of *run-time components*, parts of the state vector that represent managed objects. The environment variables are described using system types we call *component descriptors*. This information is defined according to the design of the machine and how the machine manages the memory model $M_p$.

The complete SDI description of a memory model $M_P$ of a program $P$ is the union of the SDI definitions for the compiler language constants $L_c$, the compiler program constants $P_c$ and the machine constants $M_c$ thus $SDI(M_p) = SDI(L_c) \cup SDI(P_c) \cup SDI(VM_c)$. The memory model $M_P$ describes a *name space*, which consists of a *user name space* and a *system name space*. The user name space is composed of the scopes, variables and type fields defined by the user in the source program. The system name space is composed of the user name space possibly with extra scope levels and component descriptors.

**Memory model.** A memory model of a program described by SDI defines the types of components and named variables inside a program's memory. For each component in the program memory there exists an SDI *component type* that describes the memory range of the component. A *name tree* is associated with each *component type* that defines the names and types of variables inside a run-time component associated with information constant to the program, the compiler and the VM. A *memory model* is the collection of all the name trees associated with run-time components in a program's memory and forms a representation of both user and system name spaces.

The user name space and the system name space are merely different views of the same information described with the SDI language. The information is separated because it can be hidden or shown to whom the information concerns. Programmers are familiar with the user name space which they themselves defined in the source program. System engineers may also be interested to view information from the system name space. The memory may contain more than just program variables. A compiler may introduce extra scope levels in the symbol table of which the programmer is unaware. The VM may use environment variables that define the state of the system. The names of program variables, compiler constructs and VM environment variables

together form the system name space. The system name space contains more information than the user name space.

### 3.2.1 Modeling Notation

Memory models are depicted using a modeling notation for SDI entries and types. First we explain the modeling notation, then an example memory model is described graphically using the modeling notation.

**Shapes.** The basic modeling shapes are depicted in Figure 3.2. Rectangles with rounded edges are types (type *t*). Rectangles are scope and variable entries (scope *a*, variable *b*). Text inside rectangles displayed in bold is included in the name space (variable ***b***). Text inside rectangles displayed in italics is removed from the name space (scope *a*). The long light gray rectangle denotes the component memory. Variables inside the component memory are ordered by increasing addresses from left to right.

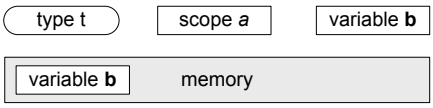



Figure 3.2: Shapes Legend

**Connectors.** The modeling shape connectors are depicted in Figure 3.3. Line connectors describe the entry sub-entry relation. The sub-entries appear below entries. When an entry has a sub-entry we say the entry *contains* the sub-entry. Type *t* contains scope *a*. Scope *a* contains variable *b*. Arrows describe type completions for variable entries. Variable *b* is of type *t2*.

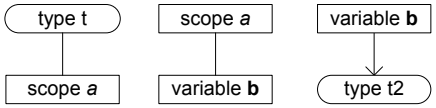



Figure 3.3: Connectors Legend

**Colors.** The modeling colors are depicted in Figure 3.4. Light gray signifies *program constants*, light gray variable and scope entry names appear in the user name space as well as the system name space. Gray signifies *VM constants*, gray variable and scope entries appear in the system name space. Dark gray signifies language *type constants*, dark gray variable and scope entry names appear in both the system and the user name space.

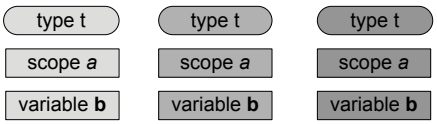



Figure 3.4: Colors Legend

### 3.2.2 Furniture Factory Example

To illustrate Definition 1 we use a running example throughout this chapter. Our example VM is the NIPS VM, which was described in Section 2.1. Our example compiler is the NIPS PROMELA Compiler which was described in Section 2.1.4. We extended the NIPS PROMELA Compiler to generate SDI for its symbol table for any source program and we have defined the PROMELA language constant types and the NIPS VM constant types in SDI. The NIPS VM state vectors are the memory models which we describe using SDI. Figure 3.1 shows the sources of the SDI with respect to PROMELA models. It makes clear that the combined SDI for thePROMELA source program defines a memory model for the running program. The example PROMELA program we will be using is a simple model of furniture factory which is described as follows:

**Furniture Factory Example:** *A furniture factory produces chairs and tables. It can hold up to 50 chairs and 30 tables in the factory. There are two trucks that are used to ship the furniture to a storage facility. The trucks can be used as soon as 10 chairs or 7 tables are completed. A truck is loaded with either chairs or tables. Sometimes a table or one or two chairs break and they are not loaded onto the truck. The material of broken furniture is reused in the factory. We assume the storage facility can hold all the furniture that is unloaded there. When a chair or a table is sold, it is removed from the storage. For the example we abstract from the delivery of the furniture to the customer.*

```
1  mtype = {CHAIR, TABLE};
2  chan trucks = [2] of {mtype, short};
3  int sold_tables;
4  int sold_chairs;
5
6  active proctype producer()
7  {
8    short c; /* chairs */
9    short t; /* tables */
10   do
11   :: c >= 10; trucks!CHAIR,10; c=c-10;
12   :: c >= 10; trucks!CHAIR,9; c=c-10;
13   :: c >= 10; trucks!CHAIR,8; c=c-10;
14   :: t >= 7; trucks!TABLE,7; t=t-7;
15   :: t >= 7; trucks!TABLE,6; t=t-7;
16   :: c < 50; c++; /* produce a chair */
17   :: t < 30; t++; /* produce a table */
18   od;
19 }
20
21 active proctype storage()
22 {
23   int c;   /* chairs in stock */
24   int t;   /* tables in stock */
25   short a; /* amount received */
26   do
27   :: trucks?CHAIR,a; c=c+a; /* rcv chairs */
28   :: trucks?TABLE,a; t=t+a; /* rcv tables */
29   :: c--; sold_chairs++; /* sell a chair */
30   :: t--; sold_tables++; /* sell a table */
31   od;
32 }
```

(a) PROMELA Model

```
1     GLOBSZ  10
2  !strinf begin scope_init 0
3     LDC     0
4     STVA    G    4    6
5     LDC     0
6     STVA    G    4    2
7     LDC     8
8     LDC     -15
9     CHNEW   2    2    0
10    STVA    G    2u   0
11 !strinf end scope_init 0
12    LRUN    4    0    P_producer  1
13    POP     r0
14    LRUN    10   0    P_storage   2
15    POP     r0
16    STEP    T    0
17 I_2:
18 !strinf begin scope_init 2
19    LDC     0
20    STVA    L    2s   0
21    LDC     0
22    STVA    L    2s   2
23 !strinf end scope_init 2
24    RET
25 E_2:
26 !strinf begin scope_exit 2
27    LDC     0
28    STVA    L    2s   0
29    LDC     0
30    STVA    L    2s   2
31 !strinf end scope_exit 2
32    RET
33
34 ... byte code continues ...
```

(b) NIPS Byte Code Snippet

Figure 3.5: Furniture Factory Example

The example is modeled in PROMELA, the resulting model is shown in Figure 3.5(a). A channel with a capacity of two messages is used to model the two trucks. Two global integers are used to keep track of the amount of sold chairs and tables. The process named *producer* models the factory that produces the chairs and the tables. Line 11 models that no chair breaks during transport, line 12 models that one chair breaks and line 13 models that two chairs break. Line 14 models that no table breaks during transport and line 15 models that one table breaks. Lines 16 and 17 model that chairs and tables are produced if they can be stored within the factory. The process named *storage* models the storage facility where the furniture is stored. The local variables named $c$ and $t$ are used to keep track of the number of stored chairs $c$ and the number of stored tables $t$. Lines 27 and 28 model the delivery of furniture. Lines 29 and 30 model the sale of a chair or a table without going into details. The code is compiled with the NIPS PROMELA Compiler to NIPS byte code. The first part of the generated byte code is shown in Figure 3.5(b). The initialisation of the global scope and the two processes is depicted in NIPS byte code. The full byte code can be found in Appendix C.

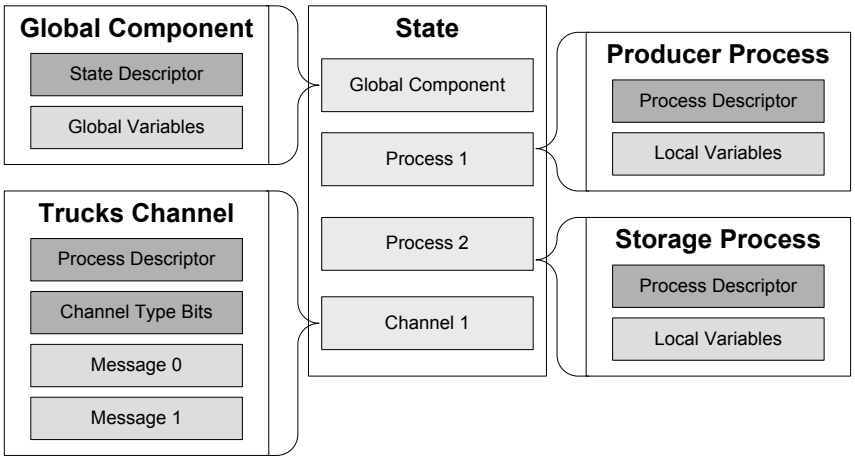



Figure 3.6: Furniture Factory - Example NIPS State

The furniture factory program memory model has four run-time components. The components are stored in a state vector depicted in Figure 3.6. The global component consists of the state descriptor and the global scope. The factory process and the storage process each consists of a process descriptor and a local scope. The trucks channel consists of a channel descriptor and up to two messages. The descriptors describe the state of the component for the VM and do not appear in the user name space. Instead these are part of the system name space.

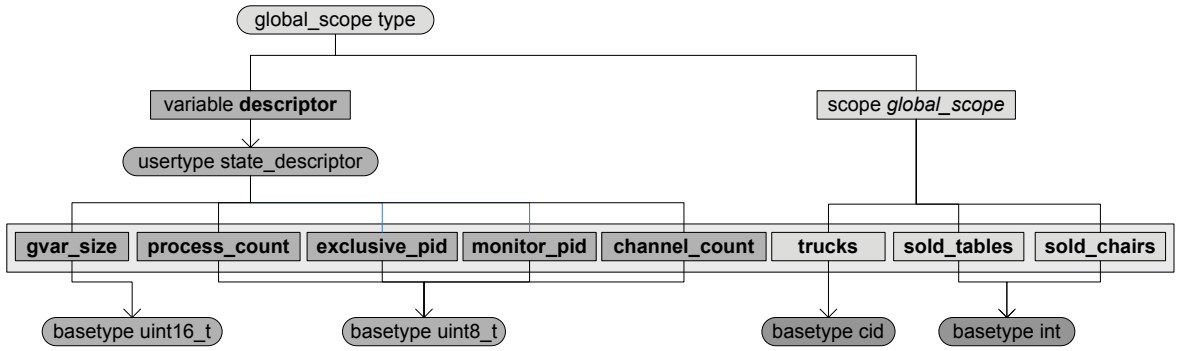



Figure 3.7: Furniture Factory - Global Component Type Name Tree

Figure 3.7 shows the name tree of the global component type. This is the type of the component in the NIPS state vector that contains the state descriptor and the globally declared variables. The variable name descriptor is included in the system name space. System variables are named *descriptor.gvar_size*, *descriptor.process_count*, *descriptor.exclusive_pid*, *descriptor.monitor_pid* and *descriptor.channel_count*. The scope name *global_scope* is excluded from the name space. Global variable declarations are named *trucks*, *sold_chairs* and *sold_tables* respectively just as they were declared on lines 2, 3 and 4 of Figure 3.5(a).

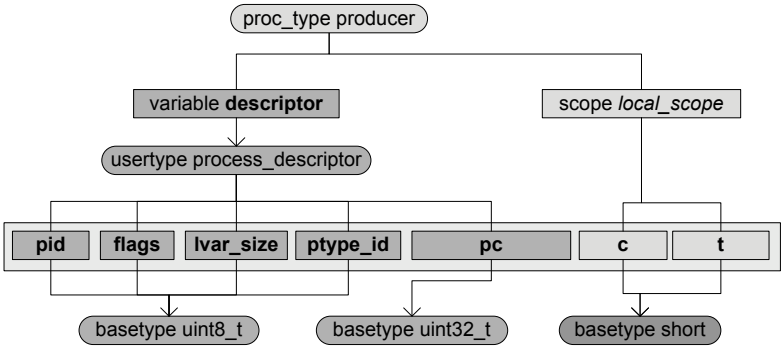



Figure 3.8: Furniture Factory - *Producer* Type Name Tree

Figure 3.8 shows the name tree of the producer process type. This is the type of the component in the NIPS state vector that contains the state descriptor and the locally declared variables. System variables are named: *descriptor.pid*, *descriptor.flags*, *descriptor.lvar_size*, *descriptor.ptype_id* and *descriptor.pc*. Each of the variables is defined to be of the *uint8_t* base type except for *descriptor.pc* which is of the *uint32_t* base type. The scope name *local_scope* is excluded from the name space. Local variable declarations of lines 8 and 9 of Figure 3.5(a) of the producer process type are named *c* and *t* respectively. Both variables are declared to be a *short* which is a PROMELA base type. The variable name *descriptor* is included in the system name space.

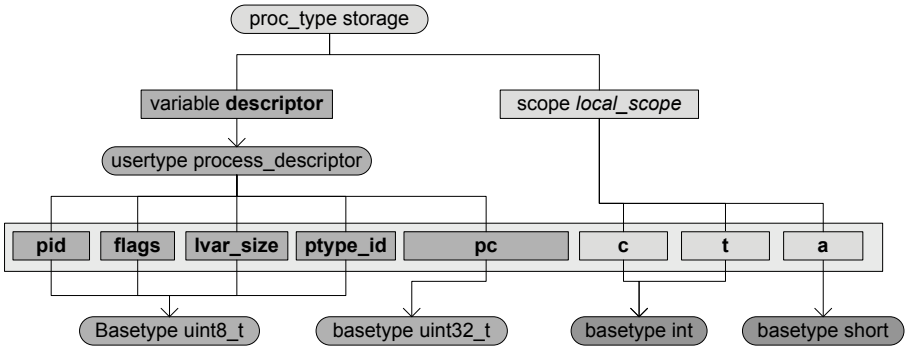



Figure 3.9: Furniture Factory - *Storage* Type Name Tree

Figure 3.9 shows the name tree of the *storage* process type. This is the type of the component in the NIPS state vector that contains the state descriptor and the variables locally declared. The variable name *descriptor* is included in the system name space. System variables are the same as in the type tree of the producer type tree. The scope name *local_scope* is excluded from the name space. Local variable declarations on lines 23, 34 and 25 of Figure 3.5(a) of the *storage* process type are named *c*, *t* and *a* respectively. The variables *c* and *t* are declared to be of the *int* type and the variable *a* is declared to be of the *short*. Both the *int* and *short* types are PROMELA base types.

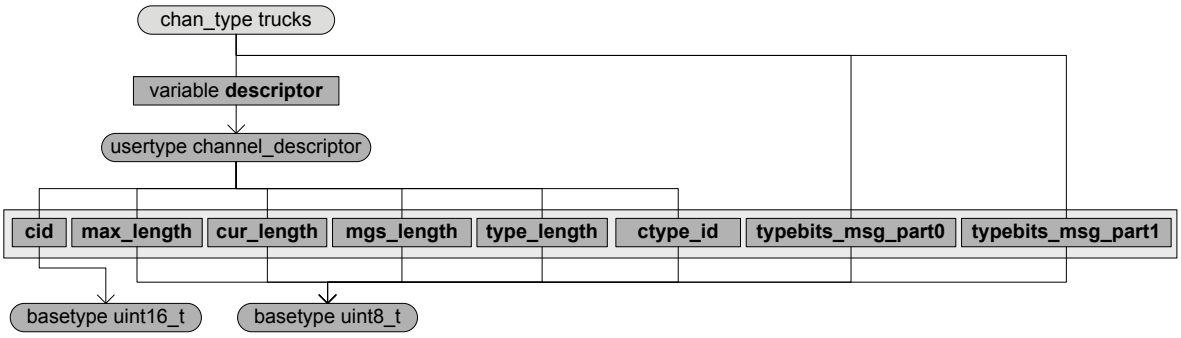



Figure 3.10: Furniture Factory - *Trucks* Type Name Tree

Figure 3.10 shows the name tree of the *trucks* channel type. Note that it only contains system name space variables. The user name space variables are in the messages exchanged via the channel. The message type of the trucks channel type is displayed in Figure 3.11. Note that both parts of the message are in italics which means the names do not appear in the name space. This makes sense because programmers cannot directly access message parts in PROMELA but rather read the parts directly from the channel into program declared variables.

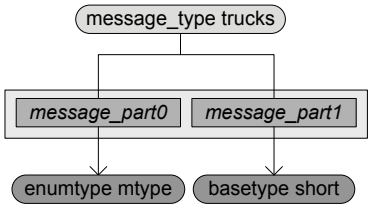



Figure 3.11: Furniture Factory - *Message* Type Name Tree

## 3.3 List Language

Up until now we have described SDI entries in an abstract graphical way. It was explained that SDI definitions describe memory models. A modeling notation was used to depict SDI name trees and the user and system name spaces. Here we explain the textual notation of the SDI language.

The syntax of the SDI language is derived from Lisp [26]. SDI is comprised of a modular list structure which is represented using parenthesis. The choice for a Lisp-like notation is made to design SDI as a concise modeling language. SDI definitions can also be written down in an XML format if it is preferred, e.g. to store in a database. The SDI entries and attributes are represented as list elements. Parenthesis denote the scope of the list element. The element starts with a left parenthesis and ends with a right parenthesis. A list can contain list elements.

```
list       := LPAREN element* RPAREN;
element    := entry | attribute | list;
entry      := LPAREN IDENTIFIER element* RPAREN;
attribute  := LPAREN IDENTIFIER value RPAREN;
value      := IDENTIFIER | INT;
LPAREN     := '('
RPAREN     := ')'
IDENTIFIER := (a..z|A..Z)(0..9|a..z|A..Z)*
INT        := ('-')?(0..9)+
```

Figure 3.12: SDI EBNF grammar

Figure 3.12 shows the SDI list grammar in the Extended Backus-Naur Form (EBNF) notation. The predefined entry and attribute keywords are listed in Table 3.1. Entries may contain sub-entries whereas attributes just consist of a name and a value. Attributes must appear within an entry and cannot exist separately by themselves. The modular structure allows parse trees of entries to be built up.

**Short-hand notation.** The list notation has a head-tail structure, each list begins with an element and the tail is a list. The SDI notation is written down in a so-called short-hand notation. This means that the left and right parenthesis that are not needed to identify the scope of an entry are left out of the notation. For instance, $(a\ (b\ (c\ (d\ ()))))$ can be written as

| subject | entries |
|---|---|
| variables | scope<br>variable<br>field<br>state |
| types | base_type<br>user_type<br>array_type<br>enum_type<br>enum_field<br>global_scope<br>component_type<br>proc_type<br>chan_type<br>msg_type<br>pointer_type |
| locations | t2s |

(a) Entry Keywords

| attributes |
|---|
| name |
| offset |
| size |
| type |
| length |
| marked |
| value |
| format |
| id |
| src_* |
| tgt_* |
| *etc.* |

(b) Attribute Keywords

Table 3.1: Predefined Entry and Attribute Keywords

(*a b c d*). Since SDI entries and attributes are unordered lists, it does not matter which entry or attribute comes first in a list, it only matters which list they appear in.

The grammar does not depict any context constraints. Possible context constraints could be that certain entries must or must not contain certain attributes or sub-entries. What the predefined entries and attributes from Table 3.1 mean and which attributes are required for the entries is described in Sections 3.4, 3.5 and 3.6.

## 3.4 Variables

In this section we describe scope and variable entries and their required attributes. Among variables we regard the *variable*, *field* and *state* entries.

### 3.4.1 Entries

*Scope entries* represent symbol table scopes and contain variables. Scopes may contain child scopes to form a hierarchy of scope levels. *Variable entries* describe variable declarations from the symbol table or variable definitions from a descriptor. *Variable entries* and *field entries* have identical attributes. Using fields instead of a variable can be preferred if the variable is part of a record type.

*State entries* describe an untyped sequence of bytes or Binary Large Object (BLOB). States are variables that do not have a static type but instead are intentionally left untyped. In fact, the SDI memory model is used to describe memory that without SDI is effectively a BLOB. The state memory area contains variable components that do have types. Component types are described in Section 3.5. Components are not SDI entries since they are dynamic entities that must be

instanced at run-time. Although they not explicitly part of the SDI language, components are part of the SDI Framework and are described in Section 4.4.

Variables may not overlap with other variables, one variable corresponds with one sequence of bytes in the memory. This means that a memory location may be occupied by one variable only. It can be determined if this constraint is observed by completing type references and checking *offset* attributes of the variables and *size* attributes of the types and scopes. The *offset* attributes of nested entries, and the *size* attribute of the types determine the memory range of the variable. Although we demand that no variable overlap may occur we do not demand that every memory location must be defined, i.e. there may be gaps in the description of the memory about which it is unknown what it contains. This is in accordance with the notion of strictly optional debugging information.

| | name | offset | size | type | marked |
|---|---|---|---|---|---|
| scope | × | × | × | | o |
| variable | × | × | | × | o |
| field | × | × | | × | o |
| state | × | × | | | |

(a) Required Attributes

| | sub entries | | | |
|---|---|---|---|---|
| scope | scope | variable | state | type |
| variable | | | | |
| field | | | | |
| state | type | | | |

(b) Allowed sub-entries

Figure 3.13: SDI Variable Entries

Figure 3.13 shows a schematic table of *variable entries* and attributes. Figure 3.13(a) depicts the required attributes for *variable* and *scope entries*. The top row shows the attributes that have a predefined semantics, from left to right: *name, offset, size, type* and *marked*. The semantics of other attributes are not defined. Other attributes can be added when needed. The left most column shows the scope and variable entry keywords, from top to bottom: *scope, variable, field* and *state*. For each entry an × in the column of an attribute denotes that the attribute is required, an *o* denotes it is optional and neither denotes it is not supposed to be defined. For example a scope must have a *size* attribute and for other variable entries it is not supposed to be defined.

Figure 3.13(b) depicts the allowed sub-entries for variable and scope entries. Scopes may contain *scope, variable, state* and *type* entries. States may only contain *type* entries which are local to the state.

### 3.4.2 Attributes

We give an explanation for each of the variable entry attributes from Figure 3.13(a). Attributes are all implicitly typed as a string attribute or an int attribute depending on their values. Numeric values imply an `int` attribute and alphabetical values imply a `String` attribute. The *name, type* and *marked* attributes are `String` value attributes and *offset* and *size* attributes are `int` value attributes.

**Name.** Each SDI entry is required to have a *name* attribute. The *name* attribute of a variable or scope is used in the name tree. The name tree is built up from the names of scopes and variables. It is the set of unique variable names that can be referred to. In programming languages variables

must have a name that is unique to the scope level. In SDI variables must have a name unique to the name space. This is a more strict requirement.

**Offset.** The *offset* attribute of an entry describes the offset in bytes within its parent entry. The parent entry of a variable entry may either be a scope or a user type. Figure 3.13(a) shows that *scope*, *variable*, *field* and *state* entries must all have an *offset* attribute.

**Size.** *Scope* entries are required to have a predefined defined size in bytes defined by the *size* attribute. *Scope* entries define the content of a specific memory range, the variables (and scopes) contained in the scope must be within scope bounds. The *size* attribute is not defined for *variable* entries. Instead, the size of a variable entry is determined by the size of its type.

**Type.** The *type attribute* of a variable is a reference to a *type entry*. The value of the type attribute may be the name attribute of any defined type entry. A variable declaration is an instance of a type and the *type entry* is a type definition. Figure 3.13(a) shows that *variable* and *field* entries must have a *type attribute*. In line with the strictly optional notion of debugging information we allow type attribute values to have the *unknown* value which does not refer to a type but to the absence of one.

**Maximum size.** The *size* attribute defines the static constant size of an entry. The size of a state is the sum of the size of its components. By definition a state may contain different components of different types. Therefore the size of the state is variable. The state size may not grow beyond the maximum size defined by the *max_size* attribute. The size and *max_size* attributes signify the initial size and the maximum size of a state which may grow and shrink dynamically.

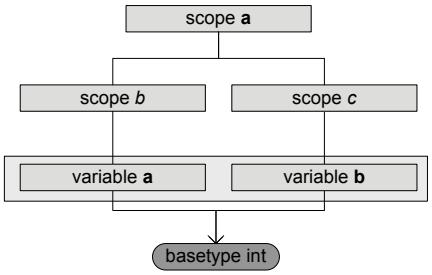
**Marked.** The *marked* attribute is an optional attribute. It describes a *marking* of an entry, annotating the entry with a label. The two predefined markings at this time are *invisible* and *descriptor* which are both used in the construction of the name space. Figure 3.13(a) shows that the *marked* attribute is optional for the *scope*, *variable* and *field* entries and is not defined for *state* entries.

In Section 3.2 the notion of the user name space and the system name space were introduced. It was also explained that both name spaces are represented by a type tree. Here we explain how the *marked* attribute can be used to annotate entries and entry sub-trees to be excluded from the user name space.

Entries may be marked *invisible* in which case their name is hidden from the user. This can be useful if the entry was not introduced in the language but a result of the compiler symbol table structure and the name is not meaningful to the user. In such a case the name of the entry can be removed from the name space. *Variable* entries which have a *marked* attribute of which the value is *invisible* correspond with the italics notation for *variable* entry shapes that was used in the modeling of SDI entries in Section 3.2.

Figure 3.14 sketches an example where the two scopes that are marked *invisible* are excluded from the name space. A compiler has a symbol table of which the name tree is depicted in Figure 3.14(a). The SDI generated for this name space is shown in Figure 3.14(b).

Because of the *invisible* markings a user may refer to variables *a.a* and *a.b* as opposed to *a.b.a* and *a.c.b* which are the names inside the compiler symbol table. The user name space consists of *a.a* and *a.b*, The middle part of the name is omitted. The symbol table name space consists of *a.b.a* and *a.c.b*. It is not always possible to use the *invisible* marking since names must be unique. If both variable entries would have a name attribute value *a* then the user name space




(a) Symbol Table Name space

```
1  (scope
2      (name a)
3      (offset 0)
4      (size 8)
5      (scope
6          (name b)
7          (marked invisible)
8          (offset 0)
9          (size 4)
10         (variable
11             (name a)
12             (offset 0)
13             (type int)
14         )
15     )
16     (scope
17         (name c)
18         (marked invisible)
19         (offset 4)
20         (size 4)
21         (variable
22             (name b)
23             (offset 0)
24             (type int)
25         )
26     )
27 )
```

(b) Invisible Marking SDI

Figure 3.14: Invisible Marking Example

would be ambiguous.

Entries may also be marked as a *descriptor* in which case their name is also hidden from the user. However, unlike sub-entries of an entry that is marked *invisible*, all sub-entries and type instances of an entry that is marked *descriptor* are also recursively marked descriptor automatically. Marking an entry a *descriptor* thus makes the whole name tree associated with the entry a descriptor. Descriptors are part of the system name space and describe system variables. These variables are in the memory, but they are not defined by the programmer as variables within in the program. Entries that have a *marked* attribute of which the value is descriptor corresponds with the dark gray colourings in the modeling of SDI entries in Section 3.2.

## 3.5 Types

This section gives a description of the type entries in the SDI language. All variables entries have a *type* attribute that refers to a type entry name. Types all contain static information about variables. There can of course be more variables that have the same *type* attribute value. Types allow the translation from a sequence of bytes to a user readable form. Whereas all variables reside at a specific location determined by their offset, types define the *size* that determines the number of bytes used to store the variable value from the variable offset. All type entries have a *name* and *size* attribute.

**Name.** The name attribute specifies the name of the type. Type names attribute values, unlike variable names are unique with respect to all types. Currently types are defined globally. This means that at this time SDI does not allow types that are valid within a limited scope.

**Size.** The size attribute specifies the type size in bytes. Types do not have an *offset* attribute because types can be instantiated in a variable and have no predefined memory location. For the same reason types may not have *marked* attributes. Types can be reused.

| | name | size | type | format | length | value | id |
|---|---|---|---|---|---|---|---|
| base type | × | × | | × | | | |
| user type | × | × | | | | | |
| array type | × | × | × | | × | | |
| enumeration type | × | × | | | | | |
| enumeration field | × | | | | | × | |
| component type | × | × | | | | | × |
| pointer type | × | × | × | | | | |

(a) Required Attributes

| | sub entries | | |
|---|---|---|---|
| base type | | | |
| user type | scope | variable | state |
| array type | | | |
| enumeration type | enumeration field | | |
| enumeration field | | | |
| component type | scope | variable | state |
| pointer type | | | |

(b) Allowed sub-entries

Figure 3.15: SDI Type Entries

Figure 3.15 shows a schematic table of type entries and attributes. Figure 3.15(a) shows the required attributes for type entries. The top row shows the attributes that have a predefined semantics, from left to right: *name, size, type, format, length, value* and *id*. The *name, type* and *format* attributes are `String` value attributes and *size, length, value* and *id* attributes are `int` value attributes. The left most column shows the type entry keywords, from top to bottom: *base type, user type, array type, enumeration type, enumeration field, component type* and *pointer type*. For each entry an × in the column of an attribute denotes that the attribute is required and an empty space denotes the attribute is not supposed to be defined. For example, a base type must have a *format* attribute but for a user type (or any other type) it is not supposed[1] to be defined. Attribute keywords that are not listed in the table do not have a predefined semantics. Type constants can be encoded using additional attributes.

Figure 3.15(b) shows the allowed sub-entries for type entries. *User type* and *component type* entries may contain *scope, variable* and *state* entries. *Enumeration type* entries may only contain *enumeration field* entries.

**Base Types.** *Base types* allow the representation of the type of a single value. Usually base types include *booleans*, *characters* and *integers*. The *format* attribute determines how this value is translated to bytes. An SDI base type used with the Furniture Factory Example is the *int* type[2] depicted in Figure 3.16.

```
1 (base_type
2     (name int)
3     (size 4)
4     (format besint)
5     (bits -31)
6 )
```

Figure 3.16: Example *int* Base Type SDI Definition

[1] It may however be defined and used for a different purpose.

[2] Note that the *bits* attribute in Figure 3.16 is not a predefined attribute, it is added to the base type for the implementation of Nips VM which uses the type bits to send base type message parts over channels.

```
1 (base_type
2     (name beuint8_t)
3     (size 1)
4     (format beuint)
5     (bits 8)
6 )
```

(a) beuint8_t

```
1 (base_type
2     (name beuint16_t)
3     (size 2)
4     (format beuint)
5     (bits 16)
6 )
```

(b) beuint16_t

```
1 (base_type
2     (name beuint32_t)
3     (size 4)
4     (format beuint)
5     (bits 32)
6 )
```

(c) beuint32_t

Figure 3.17: NIPS VM Base Type SDI Definitions

Figure 3.17 describes base type SDI definitions used to describe the descriptors of the NIPS VM. Note that these are commonly used C types. Figure 3.17(a), Figure 3.17(b) and Figure 3.17(c) describe a big endian unsigned int of one, two and four bytes respectively.

The SDI framework has a number of built-in byte formats that allow the translation from bytes to a user readable string.

| format | base type | byte order | signedness | number of bytes |
|---|---|---|---|---|
| `beuint` | int | big endian | unsigned | 1, 2 or 4 |
| `besint` | int | big endian | signed two's complement | 1, 2, 4 or 8 |
| `leuint` | int | little endian | unsigned | 1, 2 or 4 |
| `lesint` | int | little endian | signed two's complement | 1, 2, 4 or 8 |
| `char` | character | none | none (ASCII encoding) | 1 byte |
| `bool` | boolean | none | none | 1 byte |

Table 3.2: SDI Built-in Byte Formats

Table 3.2 shows the SDI built-in formats. The first column refers to the value of the *format* attribute. The second column refers to the natural name of the base type the format is used to represent. The third column refers to the byte order of the format. The forth column refers to the signedness of the format. The last column refers to the value of the *size* attribute and the constraints with respect to the format. Note that the smallest size of a format is one byte and that the *bool* format must be encoded using one byte.

There is no part in the SDI notation that allows the definition of new byte formats. SDI is designed this way because the translation of a sequence of bytes to a user readable string is dependent on the programming language used to implement it. We wish the variables to be accessible as elementary types in the implementation language as well. Note that if SDI is implemented in Java it is therefore only possible to define base types that can somehow be encoded in Java. If other formats are needed they must be added to the implementation manually. The SDI base types cover PROMELA base types and should suffice for other modeling languages as well.

**User Types.** *User type* entries allow the construction of type records. User types contain fields[3] which themselves are of a specific type. User types may contain state entries, such that part of the user type memory is reserved for components. User types may not contain any other entries. A memory location range contains one variable at most and variables are defined within type bounds. An SDI user type used with the Furniture Factory Example is the NIPS *state descriptor* user type depicted in Figure 3.18(a). It corresponds with the dark gray part of Figure 3.7. The

[3]Fields are variable entries for user type entries.

two process components in our example both contain a descriptor variable of which the type is depicted in Figure 3.18(b). The SDI description of Figure 3.18(b) corresponds with the dark gray parts of Figure 3.8 and Figure 3.9. The SDI description of Figure 3.18(c) corresponds with the dark gray parts of Figure 3.10. Although in this case the *size* attribute of the *user type entry* duplicates the size information of its child entries we choose to add it in order to allow *unknown* type attribute values and type omissions.

```
(user_type
  (name
   state_descriptor)
  (size 6)
  (variable
    (name gvar_size)
    (type beuint16_t)
    (offset 0)
  )
  (variable
    (name process_count)
    (type beuint8_t)
    (offset 2)
  )
  (variable
    (name exclusive_pid)
    (type beuint8_t)
    (offset 3)
  )
  (variable
    (name monitor_pid)
    (type beuint8_t)
    (offset 4)
  )
  (variable
    (name channel_count)
    (type beuint8_t)
    (offset 5)
  )
)
```

(a) Nips State Descriptor

```
(user_type
  (name
   process_descriptor)
  (size 8)
  (variable
    (name pid)
    (type beuint8_t)
    (offset 0)
  )
  (variable
    (name flags)
    (type beuint8_t)
    (offset 1)
  )
  (variable
    (name lvar_size)
    (type beuint8_t)
    (offset 2)
  )
  (variable
    (name ptype_id)
    (type beuint8_t)
    (offset 3)
  )
  (variable
    (name pc)
    (type beuint32_t)
    (offset 4)
  )
)
```

(b) Nips Process Descriptor

```
(user_type
  (name
   channel_descriptor)
  (size 7)
  (variable
    (name cid)
    (type beuint16_t)
    (offset 0)
  )
  (variable
    (name max_length)
    (type beuint8_t)
    (offset 2)
  )
  (variable
    (name cur_length)
    (type beuint8_t)
    (offset 3)
  )
  (variable
    (name msg_length)
    (type beuint8_t)
    (offset 4)
  )
  (variable
    (name type_length)
    (type beuint8_t)
    (offset 5)
  )
  (variable
    (name ctype_id)
    (type beuint8_t)
    (offset 6)
  )
)
```

(c) Nips Channel Descriptor

Figure 3.18: Nips VM Descriptors - User Type SDI Definitions

**Enumeration Types.** *Enumeration type* entries provide a mapping from symbolic names to byte values. Enumeration type entries must have a *format* attribute which determines the integer value of the enumeration type instance. An enumeration type contains one or more *enumeration field* entries. The enumeration type keyword is *enum_type*.

*Enumeration field* entries have a *name* and a *value* attribute and may only appear within enumeration type entries. The enumeration field's *name* and *value* attributes determine the symbolic value associated with the integer value of the enumeration type instance . Typically we may view *blue* instead of *1*, where *blue* is the symbolic value and *1* the value that is stored. The enumeration field keyword is *enum_field*.

An SDI enumeration type used with the Furniture Factory Example is the mtype depicted in Figure 3.19. The enumeration type is used to encode the Promela *mtype* feature. In the furniture factory example the value *zero* represents the symbolic value *chair* and the value *one*

```
(enum_type
    (name mtype)
    (size 1)
    (format beuint)
    (bits 8)
    (enum_field
        (name chair)
        (value 0)
    )
    (enum_field
        (name table)
        (value 1)
    )
)
```

Figure 3.19: Furniture Factory - *mtype* Enumeration Type SDI Definition

represents the symbolic value *table*.

**Array Types.** *Array Type* entries describe array type declarations. The *length* attribute, which is only used with array type entries, describes the number of array elements. The *type* attribute of the array type entry specifies the type of the elements of the array type. The value of the *size* attribute of an array type is equal to *length* times the *size* of the type of the array type entry.

The names of array elements are constructed from the variable *name* attribute referring to the array type followed by the left bracket, a number $n \in \{0...length - 1\}$ followed by the right bracket where *length* is the value of the *length* attribute of the array. e.g. *a[2]* denotes the third position the array type variable *a*.

**Component Types.** *Component types* describe the type of run-time components that together form the system state. Components are described in depth in Section 4.4. Querying a component's variable values and attribute values using the SDI framework are important API functions as will be shown in Section 4.3 and Section 4.4.

Components contain variable debugging information, information of which the value is unknown at compile time. Examples of variable debugging information are: the size of the component, the type id which determines the static type of the component and the component id which gives a unique name to the component.

Aside from the *name* and *size* attribute that are required for every type entry, component type entries must have an *id* attribute. For each component type an id unique to the component type entry must be defined. The *id* attribute is used to identify the type of a component. The type id serves as an index for a query table from which the component types can be retrieved. The *size* attribute describes the initial component size. If the component is dynamically sized it is variable debugging information for which a size variable is described. Optionally the *max_size* attribute can be used to describe the maximum size to which a component of this type may grow.

SDI contains different component types which can be used for different purposes. The *global scope type* entry is a special component type entry of which there may only exist one per memory model. The name tree of the global component type defines the global name space. The global component can be used to store a *state descriptor* and global variables. The global component entry keyword is *global_scope*. *Component types* represent the type of run-time components[4]. The

[4]We avoid the term object because SDI does not yet support the methods associated with objects.

name tree of the component type defines a name space local to the component. The component type entry keyword is *component_type*.

*Process types* are component types that represent the type of run-time processes. The process type entry keyword is *process_type*. Object types and process types are semantically the same except that they use different indexes for type id, i.e. there are object type ids and process type ids.

A special component type to describe channels is the *channel type*. Conceptually channels are a sort of array type component that contains messages. Channel *message types* are described separately from the channel type but have the same type *id* in order to relate the corresponding channel and message types to each other. Channels have a *length* attribute that describes their maximum length. To allow the access to the current channel length the channel type should contain a variable which represents a range of byte in the channel component where the length variable is stored to access this information.

```
1  (global_scope
2      (offset 0)
3      (size 16)
4      (variable
5          (name descriptor)
6          (type state_descriptor)
7          (offset 0)
8          (marked descriptor)
9      )
10     (scope
11         (name global_variables)
12         (offset 6)
13         (marked invisible)
14         (variable
15             (name sold_chairs)
16             (type int)
17             (offset 6)
18         )
19         (variable
20             (name sold_tables)
21             (type int)
22             (offset 2)
23         )
24         (variable
25             (name trucks)
26             (type cid)
27             (offset 0)
28         )
29     )
30 )
```

(a) Global Component Type

```
1  (proc_type
2      (name producer)
3      (size 12)
4      (id 1)
5      (variable
6          (name descriptor)
7          (type process_descriptor)
8          (offset 0)
9          (marked descriptor)
10     )
11     (scope
12         (name local_variables)
13         (marked invisible)
14         (offset 8)
15         (scope
16             (name s2)
17             (marked invisible)
18             (offset 0)
19             (variable
20                 (name c)
21                 (type short)
22                 (offset 0)
23             )
24             (variable
25                 (name t)
26                 (type short)
27                 (offset 2)
28             )
29         )
30     )
31 )
```

(b) *Producer* Component Type

Figure 3.20: Furniture Factory - Component Type SDI Definitions

An SDI component type used with the Furniture Factory Example is the global component type SDI definition depicted in Figure 3.20(a). On lines 4 to 9 a variable called *descriptor* is defined. Its type, specified on line 6, is *state_descriptor* of which the definition is depicted in Figure 3.18(a). On line 8 the variable is marked as a descriptor. It explains the darker gray colouring of the name tree associated with the variable in Figure 3.7. Lines 10 to 29 define the global scope for the Furniture Factory Example. The scope is has the name *global_scope* but it is marked as an invisible entry which means the name *global_scope* is removed from the name space. Figure 3.7 shows the type name tree of the global component which can be constructed

using the global component type SDI definition.

Similarly the process component type SDI definition depicted in Figure 3.20(b) can be used to construct the type name tree shown in Figure 3.8. The producer process type has an id attribute which is used to distinguish Not depicted here are the SDI definitions for the storage process type and the trucks channel type.

**Pointer Types.** Component names cannot be statically resolved in the name space built up by SDI definitions. By this we mean that at compile-time we do not know what the value of a variable holding a component identifier will be at run-time. Consequently, we do not know the name of a component in the memory even if we know its component identifier. To find a component in the memory by name, the name of the component must be linked to the component identifier in the run-time environment.

The SDI language has *pointer types* to resolve component names. In the memory model defined by SDI a pointer type variable a sequence of bytes in the memory that holds the identifier of a component. The *type* attribute denotes the type the pointer type references. We only allow the reference of component types for now because component type pointers add component names to the name space. A global pointer type variable connects the global name space to a component name space[5].

The attributes of a pointer type are those of a base type plus the *type* attribute. The *name*, *size* and *format* attribute are treated as they are in the base type. The *type* attribute is used to denote the type references. The pointer type entry keyword is *pointer_type*.

```
1 (pointer_type
2     (name producer_ptr)
3     (type producer)
4     (size 1)
5     (format beuint)
6     (bits 8)
7 )
```

(a) *producer_ptr* Pointer Type

```
1 (pointer_type
2     (name storage_ptr)
3     (type storage)
4     (size 1)
5     (format beuint)
6     (bits 8)
7 )
```

(b) *storage_ptr* Pointer Type

```
1 (pointer_type
2     (name trucks_ptr)
3     (type trucks)
4     (size 2)
5     (format beuint)
6     (bits 16)
7 )
```

(c) *trucks_ptr* Pointer Type

Figure 3.21: Furniture Factory - Pointer Type SDI Definitions

For the running example this means that the base type *pid* should be replaced by two pointer types. The first pointer type is *producer_ptr* that points to the *producer proc_type*. Its SDI definition is shown in Figure 3.21(a). The second pointer type is *storage_ptr* that points to the *storage proc_type*. Its SDI definition is shown in Figure 3.21(b). Both processes are run as an unnamed instance of the *producer proc_type* and *storage proc_type* respectively. There are no global variables of the type *producer_ptr* or *storage_ptr*. Therefore the name space of the processes is not connected to the global name space. Process variables are only accessible after a process is selected by process id. The base type *cid* should be replaced by the pointer type *trucks_ptr* that points to the *trucks chan_type*. Its SDI definition is shown in Figure 3.21(c). The global variable trucks is a *trucks_ptr* variable.

Using the SDI modeling notation we are now able to give an SDI memory model of the example state depicted in Figure 3.6. The NIPS VM example state modeled in SDI is shown in Figure 3.22. The white rectangles are not SDI entries but are run-time entities. States and components will be explained in more detail Section 4.4. In Figure 3.22 figure we abstract from variables in

[5]Provided that its value references a component and is not null.

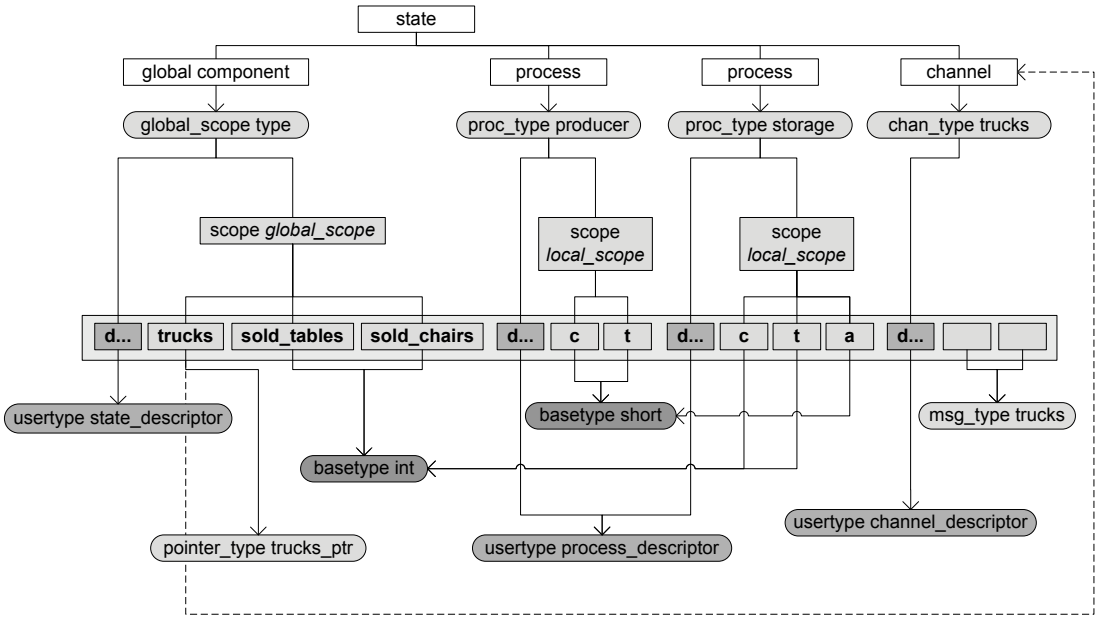



Figure 3.22: Furniture Factory - Example Nips State

state, process and channel descriptor user types and message types. It can be seen that the base type *cid* has been replaced by the pointer type *trucks_ptr*. The text *d...* in the gray rectangles is an abbreviation for *descriptor*. The dashed arrow denotes a resolved pointer. The global variable *trucks* declared on line 2 in Figure 3.5(a) is resolved to the channel component with the corresponding *id* attribute. The name space is extended with *trucks.descriptor.** where the *descriptor* variable is of the *channel_descriptor* user type and * are the variables in the *channel_descriptor* type.

## 3.6 Locations

Source level debuggers require that there exists a mapping from the source code to the target code and vice versa. Such a mapping allows a debugger to execute a target program step by step whilst keeping track of the source code.

Target code is typically described as a sequence of instructions. A target to source mapping describes for each instruction in the target code from which source line and column it is generated. Each instruction is associated with a *program counter*, a number that represents start of the instruction in the code. Points in the code, defined by a line and column number or a program counter, are referred to as *locations*. Source locations and target locations need to be related to each other by use of debugging information.

A mapping form target locations to source locations can be established in different ways. The source code can be *interleaved* with the target code such that source code references appear in between the target code instructions. This can be done by means of special instructions

or comments with a discrete debugging language or using the source code itself. SDI offers a different approach which requires no interleaving in the target code. In the SDI approach the target to source mapping is stored separately.

**Target-to-Source entries.** To create the mapping the compiler must generate target to source entries. The compiler must keep track of the instruction numbering during code generation. The target to source entry keyword is *t2s*. Target to source location entries provide a mapping between the source code and the target code which resembles a three way database tupple. The mapping is bidirectional. If the program counter is known then the source line and column can be queried. If the source line and column are known then the program counter(s) can be queried.

**Program Counter.** The *pc* attribute specifies the target line of the target-to-source entry. The target code program counter is saved with this attribute.

**Source Line.** The *line* attribute specifies the source line of the target-to-source entry. The source line refers to the line number in the compiler source.

**Source Column.** The *col* attribute specifies the source column of the target-to-source entry. The source column attribute refers to the column number within the line in the compiler source.

```
1  ...preceding t2s entries...
2    (t2s
3        (pc 143)
4        (line 10)
5        (col  3)
6    )
7    (t2s
8        (pc 146)
9        (line 10)
10       (col  3)
11   )
12   (t2s
13       (pc 149)
14       (line 10)
15       (col  3)
16   )
17 ...following t2s entries...
```

Figure 3.23: Furniture Factory - *Target-to-Source* Definition SDI Snippet

Figure 3.23 shows three SDI target-to-source entries for the Furniture Factory model which map three instructions to line 10, column 3.

The SDI language allows the user to save additional location information within the SDI entries representing the locations of variable, scope and process type declarations. It is necessary to add attributes to the entries that specify the declaration start and end points in the source code and the corresponding low and high program counters.

Table 3.3 shows the attributes that can be used to save the declaration location information. It is not necessary to add the attributes defined in the table in order to distinguish variables from each other since each variable has its own unique name and place in the name space. The memory locations of variables can be associated with source locations. The first column of Table 3.3 denotes the attribute name, the second column denotes the attribute value type, the third column gives a suggestion for when the attribute is useful and the fourth shows which location information the attribute describes.

| attribute | type | useful for entries | value denotes |
|---|---|---|---|
| `src_line_start` | *int* | all variable and type entries | declaration begin source line |
| `src_col_start` | *int* | all variable and type entries | declaration begin source column |
| `src_line_end` | *int* | all variable and type entries | declaration end source line |
| `src_col_end` | *int* | all variable and type entries | declaration end source column |
| `pc_low` | *int* | `scope` and `proc_type` | declaration begin target line |
| `pc_high` | *int* | `scope` and `proc_type` | declaration end target line |

Table 3.3: Additional Variable and Scope Attributes

## 3.7 Concluding Remarks

In this chapter we have presented the SDI language. The language has a list structure and is composed of entries and attributes for scopes, variables, types and locations. SDI is designed to describe the debugging information for modeling languages used with explicit state model checkers. SDI is used to build a memory model at run-time which relates otherwise unreadable binary state vectors back to source-level variables, names and types. We have shown that the Nips memory model can be described using SDI.

To place the SDI language in context to work related to it we give a comparison of SDI with stabs, Spin symbol table debugging information, Dwarf and the Java `class` file format which were discussed in Chapter 2.

| language or format | SDI | Dwarf | stabs | `class` | Spin |
|---|---|---|---|---|---|
| language support | modeling | programming | programming | Java | Promela |
| block structured | ✓ | ✓ | | | |
| graphical notation | ✓ | | | | |
| specifies a byte format | | ✓ | ✓ | ✓ | |
| is an object format | | | | ✓ | |

Table 3.4: Debugging Information Notations

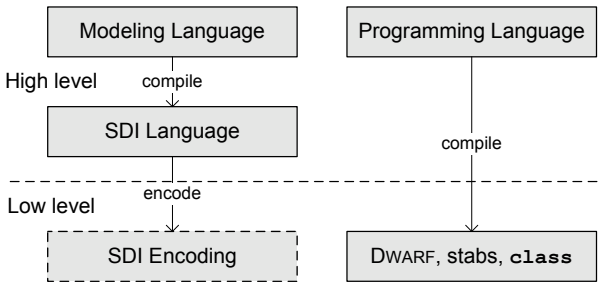



Figure 3.24: Debugging Language Levels

Table 3.4 shows the debugging information languages and the traits these languages have. It can be seen they are designed for use with different sources languages. Dwarf supports many procedural programming languages and has a rich set of entries to describe their features. Stabs

a has less rich set of predefined features. SDI is a designed to describe debugging information of modeling languages used with explicit state model checkers and as such does not support all features of modern programming languages. Notably missing features of SDI that are supported by DWARF include functions, methods and procedures and associated call frame constructs and object oriented features for access restrictions and error handling.

SDI and DWARF are both block structured as are programming languages but Figure 3.24 shows that SDI is truly a high-level modeling notation since it does not describe a low-level byte file format. From the formats Table 3.4, only the Java `class` file format contains runnable code which makes it an object file format.

The SDI language is part of the SDI Framework which uses the memory models defined by SDI to facilitate a debugging API which provides a means for source-level debugging. The SDI framework is described in Chapter 4.

# Chapter 4

# The SDI Framework

This chapter describes the Static Debugging Information (SDI) Framework. The SDI Framework is a state manipulation framework that offers support for source level debugging by constructing memory models of running programs.

The design of the SDI Framework is described in terms of its components. Section 4.1 gives a top level view of the design. Section 4.2 describes the syntactic analyser. Section 4.3 describes the symbol table. Section 4.4 describes the state API. Section 4.5 describes the transition API. Finally, Section 4.6 describes the compiler extensions and Section 4.7 describes the SDI Debugger.

## 4.1 Introduction

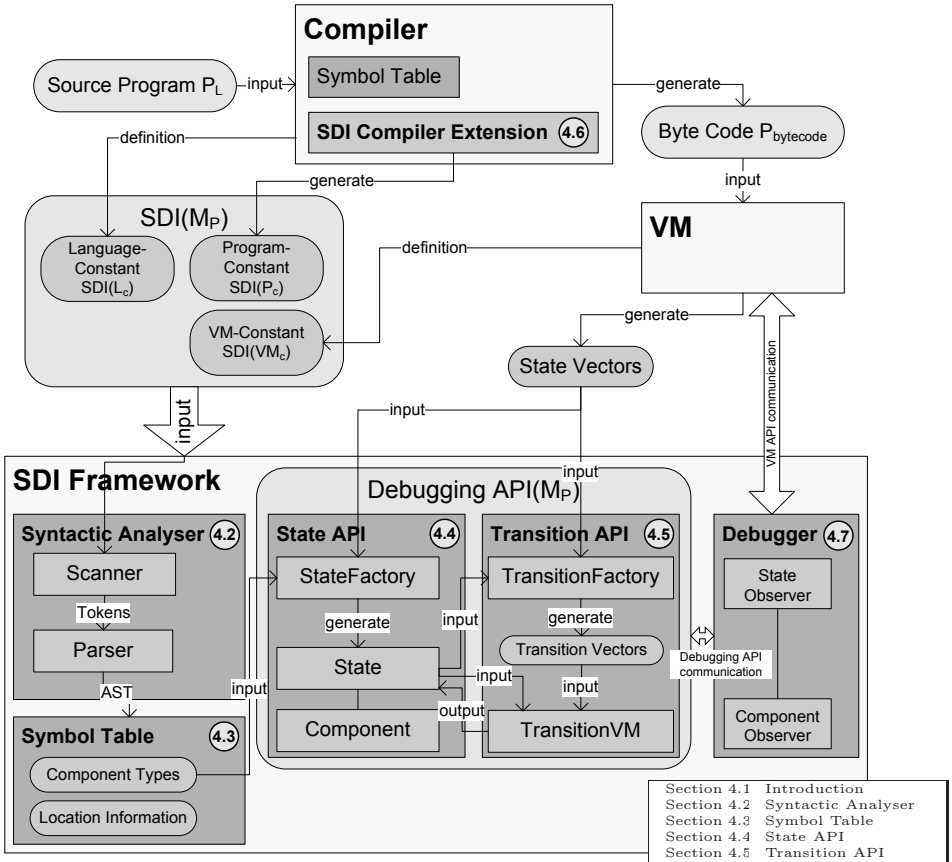
The SDI language, described in Chapter 3, drives the SDI Framework. The memory models, defined by SDI language definitions, are the basis for the debugging functionality supported by the SDI framework. Recall that Definition 1 defines a memory model of a program described by SDI as the types of components and named variables inside a program's memory at run-time. For each run-time component in the program memory, there exists an SDI *component type* that describes the memory range of the component in terms of *scopes, variables, constants* and *types*. A *name tree* is associated with each *component type* that defines the names and types of variables associated with program constants, compiler constants and VM constants that describe a run-time component. A *memory model* is the collection of all the name trees associated with run-time components in a program's memory and forms a representation of both user and system name spaces.

The SDI framework fits in a tool architecture where the tool consists of separate tool components. The memory model described by SDI definitions allows the tool components to communicate state information with each other, through a clearly defined debugging Application Programming Interface (API) called a *debugging API*.

The SDI Framework facilitates a *debugging API*, which consists of function calls that enable debuggers to access the information in state vectors associated with running programs, for which memory models have been defined using SDI. We require that this debugging API, as stated in Section 1.1, consists of function calls that can be used by a debugger when the need exists

for source-level debugging functionality, such as displaying and modifying information in the memory of running programs and logging and replaying the program's behaviour. Displaying states can be seen as a form of introspection, editing state values as a form of intercession and debugging information as a means for reification.

The debugging API should be easy to implement in existing tools that make use of state vectors to store states. In particular, we would like to use it for debugging within explicit state model checkers, specifically the NIPS VM.




(a) SDI Framework Architecture

(b) Chapter Structure

Figure 4.1: SDI Framework Design Overview

The architecture of the SDI Framework is depicted in Figure 4.1(a). The SDI state manipulation framework is composed of a *syntactic analyser*, a reconstructed *symbol table*, a generic runtime *component API*, a *component factory* that generates the component API and a *component transitions language*. The chapter structure is shown in Figure 4.1(b).

The *syntactic analyser* consists of a *scanner* and a *parser*. The *scanner* reads SDI input and generates a token stream. The *parser* reads this token stream and generates an Abstract Syntax Tree (AST). The design of the *syntactic analyser* is given in Section 4.2.

In the SDI Framework, a *symbol table* is used at run-time to relate otherwise unreadable binary target code back to high level names and types. The symbol table is used to store and retrieve debugging information defined by the SDI language. It allows the retrieval of *types*, *component types* and *source locations*. The SDI Framework symbol table is explained in Section 4.3.

The debugging API consists of two parts, the *State API* and the *Transition API*. The *State API* is designed to allow debuggers to navigate the name spaces associated with component types and the memory as a whole, in order to retrieve constants and variable values and modify variable values. The SDI Framework State API is described in Section 4.4. Central to the framework are the run-time components. For each run-time component, its SDI component type definition determines its debugging API. A *component API* is composed of functions for retrieving constants and variable values by name, as a String or an Integer. Additionally it offers functions for modifying the variable values.

To facilitate debugging of state vectors we must know what the state components are and what their type is, in order to gain access to variables and values. Furthermore, SDI components may contain a *state* variable that consists of separate components. This yields a hierarchy or graph of components. A *factory class* can be used to produce component objects from a state vector input, which is supplied by state vector generator. We call this factory class the *state factory*. The state factory produces the hierarchy of typed components. The approach for the generation of states in the SDI Framework is defined in Section 4.4.1.

The *Transition API* is designed to be used for logging, observing and replaying the behaviour of programs for which there are SDI memory models. The SDI Framework, as a state manipulation framework, uses a *transition language* that works together with components and their types. The transition language defines *instructions* for component *creation*, component *disposal*, component *resizing* and *changing* component values. Component transition instructions work on high level components, as well as the state vector. The component transition semantics is bidirectional. For every instruction there is an inverse instruction, that when executed undoes the changes made to the state. The SDI transition language is described in Section 4.5.

The *SDI compiler extensions* can assists a programmer in extending a *compiler* with SDI generation functionality. This is necessary in order to use the SDI framework and the debugging functionality it offers. The *compiler* must be extended to generate *variable*, *scope* and *type* SDI from its symbol table. *Location information* must be encoded separately to relate source and target code to each other. The *SDI compiler extensions* are described in Section 4.6.

The SDI framework is designed to be used with a graphical debugger that can display states to the user. A design of a graphical debugger that uses SDI is described in Section 4.7.

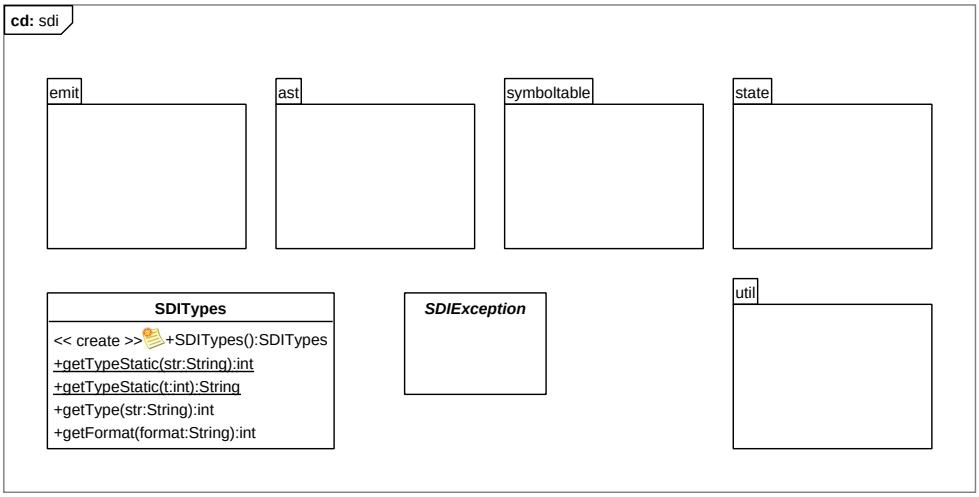



Figure 4.2: SDI Framework *packages* Class Diagram

The design of the SDI framework described here is implemented in *Java*. This is done because we would like to be able to integrate a graphical debugger into the framework that makes use of Java graphical user interface components. In particular, the Eclipse Framework [11] offers a variety of graphical component to choose from. As a result, only tools that have a Java implementation (or a Java language binding) can make use of the SDI Framework.

The Java implementation is stored in the *sdi* package, as is shown in Figure 4.2, which contains five sub packages: *emit*, *ast*, *symboltable*, *state* and *util*. The *ast* package contains an SDI *scanner* and an SDI *parser*, it serves as a syntactic analyser. The *symboltable* package contains the look-up structures for static debugging information. It includes component types, type invariants and location information. The *state* package contains the classes used to represent instances of states and components. Since the state format is an implementation choice for the explicit state model checker, and no assumptions about it can be made, it is not possible to provide a state factory class that is compatible with all explicit state model checkers. A generic state factory that can be used in some cases is provided. The *util* package contains classes that perform help functions such as the encoding of unsigned types which are not native to Java.

The next sections give a more detailed description of the modular design components and the Java implementation packages of the SDI Framework.

## 4.2 Syntactic Analyser

In order to use the SDI language, we need to read SDI specifications into the SDI Framework. The *syntactic analyser* consists of a *scanner* and a *parser*.

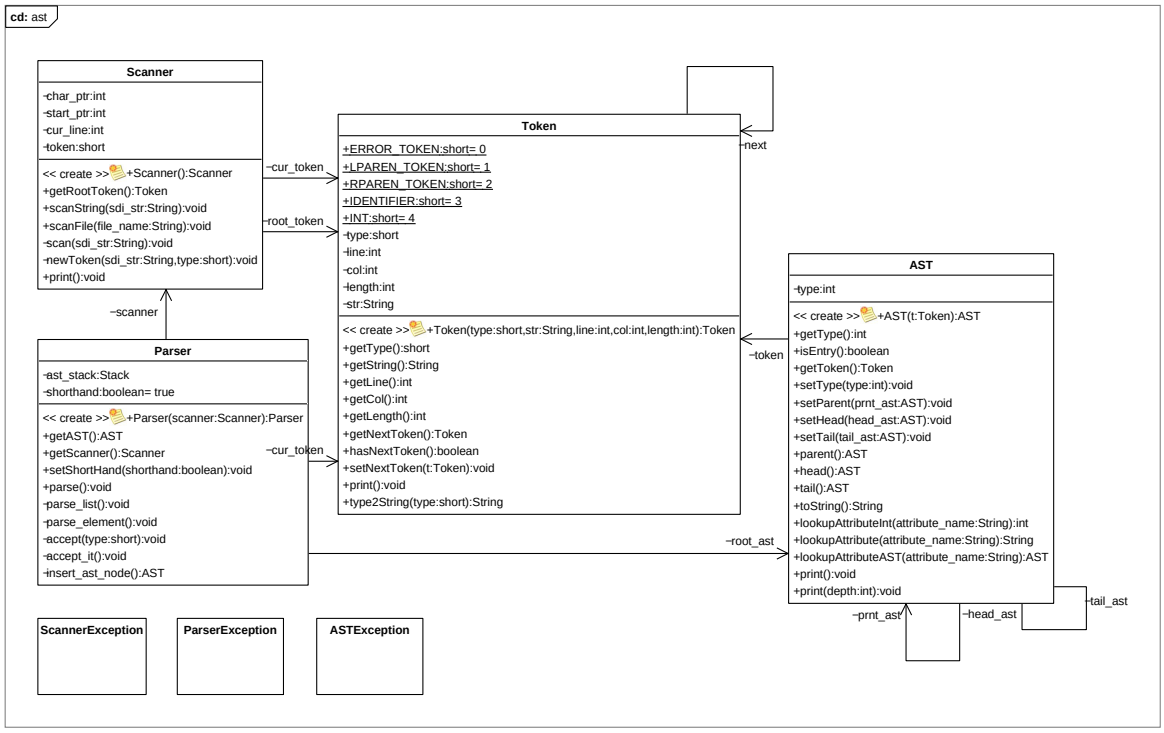



Figure 4.3: SDI Framework - *ast package* Class Diagram

Figure 4.3 shows the Class Diagram of the *ast* package. The *scanner* takes an SDI definition as input and generates a *token* stream from it. Token objects contain line and column number, the token string and a token type, which can be `LPAREN`, `RPAREN`, `IDENTIFIER` or `INT`, as is explained in Section 3.3. The *parser* takes the token stream as input and generates an Abstract Syntax Tree (AST). The AST has a list structure where every list AST node has a head and a tail. The AST contains the information defined in the SDI definition input.

## 4.3 Symbol Table

In the SDI Framework a symbol table is used at run-time, in order to relate otherwise unreadable binary target code back to high level names and types. The symbol table is used to store debugging information described in the SDI language, such as the variables, types and source locations described in Chapter 3.

The run-time symbol table is constructed in conformance with the memory model of the SDI language (see Definition 1 in Section 3.2). Recall that we defined three sources of static debugging information used to construct a memory model of a running program: the language, the program and the (virtual) machine the program runs on. The language SDI constants consist of type constants, such as base types defined with the compiler. The program SDI constants consist of user-defined variables, types and source locations that may be stored in the compiler symbol table. The machine SDI constants consist of machine types and meta structure definitions, used by the machine to manage the memory of the program. The more information is available about the source program defined in SDI, the better the source level debugging functionality will be. Incomplete information will result in an incomplete memory model.

The SDI symbol table allows the retrieval of types, state component types and location information. Types can be retrieved from the symbol table by name. Component types are retrieved from the symbol table by type identifier. Locations can be retrieved either by program counter or by source line and column number.

In this section we also explain the process of how component types are created from SDI source. The list AST created by the parser is transformed into a structure of `Entries` to facilitate a fast look-up structure for static debugging information. Each entry keyword in an SDI definition is represented by an `Entry` object at run-time, from which its attributes can be queried by name. Component type entries are transformed into `ComponentType` structures. These structures are placed in the symbol table represented by the `SymbolTable` class, from which they can be retrieved by their *type id* attribute value.

Figure 4.4 depicts the class diagram of the *symboltable* package, which contains the `SymbolTable`, `Entry`, `TypeNode`, `TypeTree`, `BaseType` and `ComponentType` classes.

Figure 4.5 depicts part the object diagram of the `SymbolTable` for the Furniture Factory example. It only shows the attribute values of the *global component* `Entry` object, neither the attributes of other `Entry` objects nor the target to source location mapping which are also stored are displayed. The values of the integer attributes *offset* (zero) and *size* (sixteen) can be retrieved from the `ComponentType` by accessing the `Entry` object it was constructed from. The *descriptor* and *global variable* entries can be accessed by *name* and *offset*. It can also be seen that the PROMELA and NIPS VM types can be retrieved from types map in the symbol table. The process and channel `ComponentTypes` named *producer*, *storage* and *trucks* can be retrieved from the symbol table by process and channel *type id* respectively.

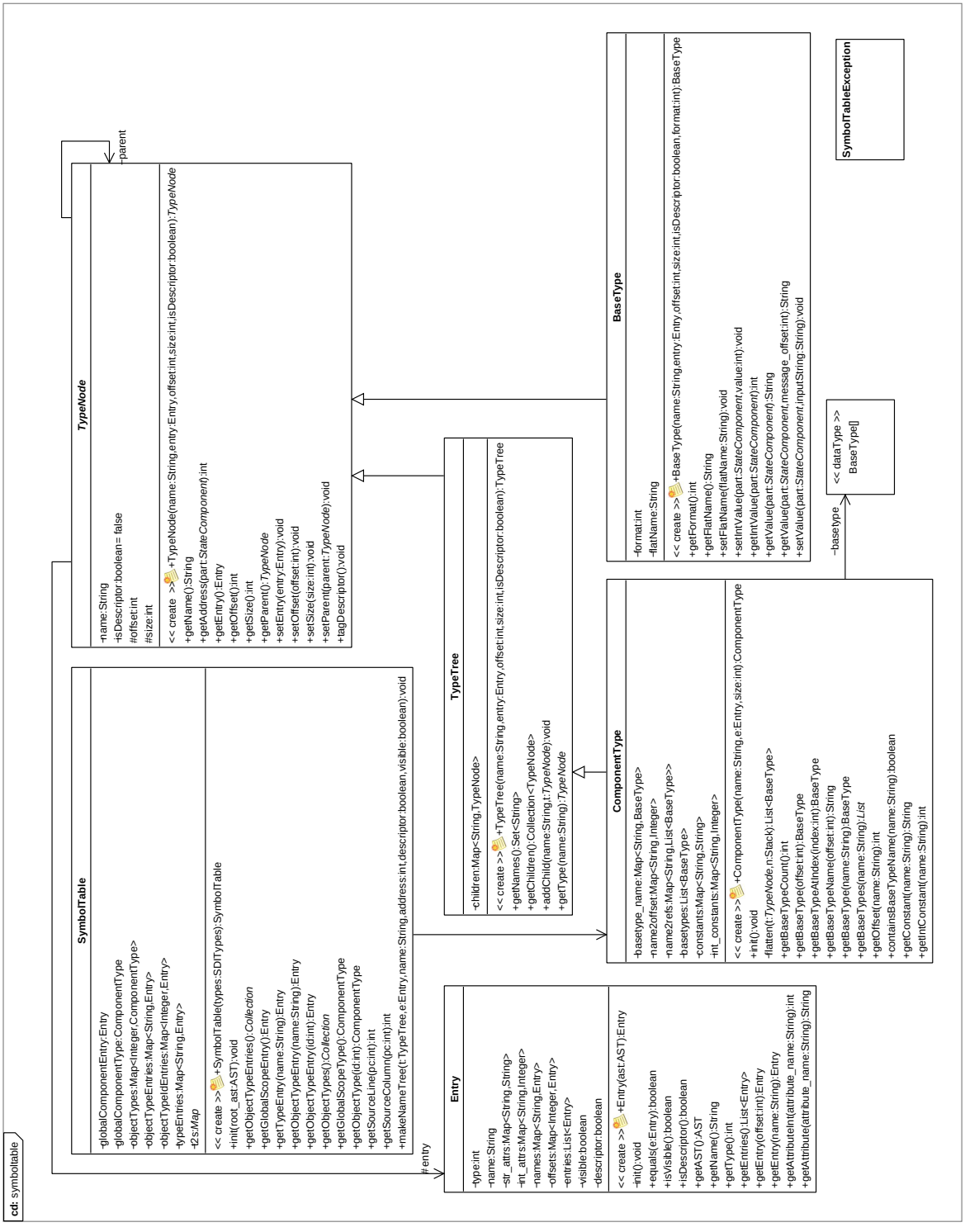



Figure 4.4: SDI Framework - *symboltable package* Class Diagram

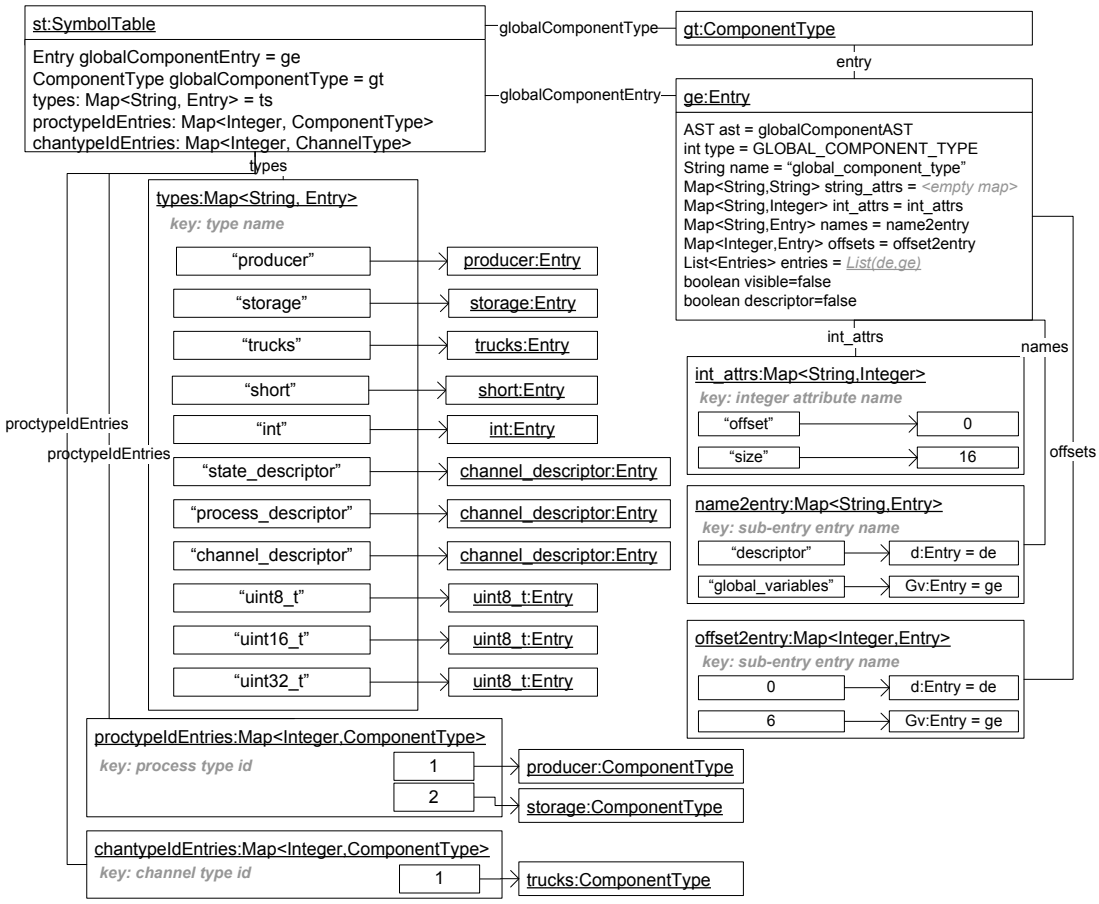



Figure 4.5: Furniture Factory - *SymbolTable* Object Diagram

### 4.3.1 Component Types

For each run-time component in the program memory, there exists an SDI *component type* that describes the memory range of the component in terms of *scopes, variables, constants* and *types*. How component types are structured and why they are defined as both trees and flat look-up structures is discussed in this sub-section.

The design decision to represent component types as tree structures is made because a *name tree* is associated with each *component type*. A modeling notation for SDI, introduced in Section 3.2, was used to depict name trees. We wish to be able to navigate such name trees in order to access uniquely named pieces of information in the program memory such that users can use a debugger to access variables by name. The design decision to represent component types as flat structures is made because the names of the variables may not be known. In that case the names must be retrievable from the type and the information must be accessible by its offset within the component, i.e. its memory location.

**Type Trees.** For each component type, one component type tree is constructed. A *component type tree* is the name tree of a component type entry, in which the types of variables are resolved to base types. Type tree leafs are base type instances that allow the translation of component values to String values or integer values.

**Navigating Type Trees.** Component type trees contain all the information there is to know about component type entries in terms of the SDI definition. They allow debuggers to to query variable values at offsets within the component type, as well as variable names from the name space. Single value queries can be seen as paths in the type tree from the root component type entry to a base type leaf. Multi-value queries can be seen as paths in the type tree from the root component type entry to a user types or a scope entry. It is even possible to query all the variable values of a component type at once. Each part of a name refines the search further. The coarsest refinement of a search is the entire state, the finest refinement is a base type instance. However, every time a query is done a search over the tree must reveal the path to the requested information. We observe that there are a limited amount of possible queries. Calling a search algorithm which performs a search on a type tree every time a query is made is not desirable. In order to prevent this situation, queries are preprocessed.

**Flat Type Trees.** The component type tree is flattened into a single component type object. When the type tree is flattened, the component name space is generated. Variable names are constructed from the variable, scope and type names, which are appended to each other and separated by dots.

Figure 4.6 depicts the object diagram of the *global component type* of the Furniture Factory example state of Figure 3.22.

A slightly more formal definition of the names of variables and constants in component types is described in Definition 3, 4, and 5. First, we give an informal definition of the type tree refinement step relation in Definition 2, which can be intuitively understood from the graphical representation of component type trees. In each of the definitions: $E$ denotes a *scope*, *variable* or *field* entry; $name(E)$ returns the name attribute string value of entry $E$; $attributes(E)$ is the set of attributes of $E$; and $visible(E)$ denotes that the entry is not marked invisible, i.e. $invisible \notin markings(E)$ and $markings(E) \subseteq \{invisible, descriptor\}$.

**Definition 2** (Type tree refinement relation $R$)**.** The type tree refinement step relation $R(E_p, E_q)$ is a binary relation defined as follows. We say that $E_q$ is a type tree refinement step of $E_p$ in a

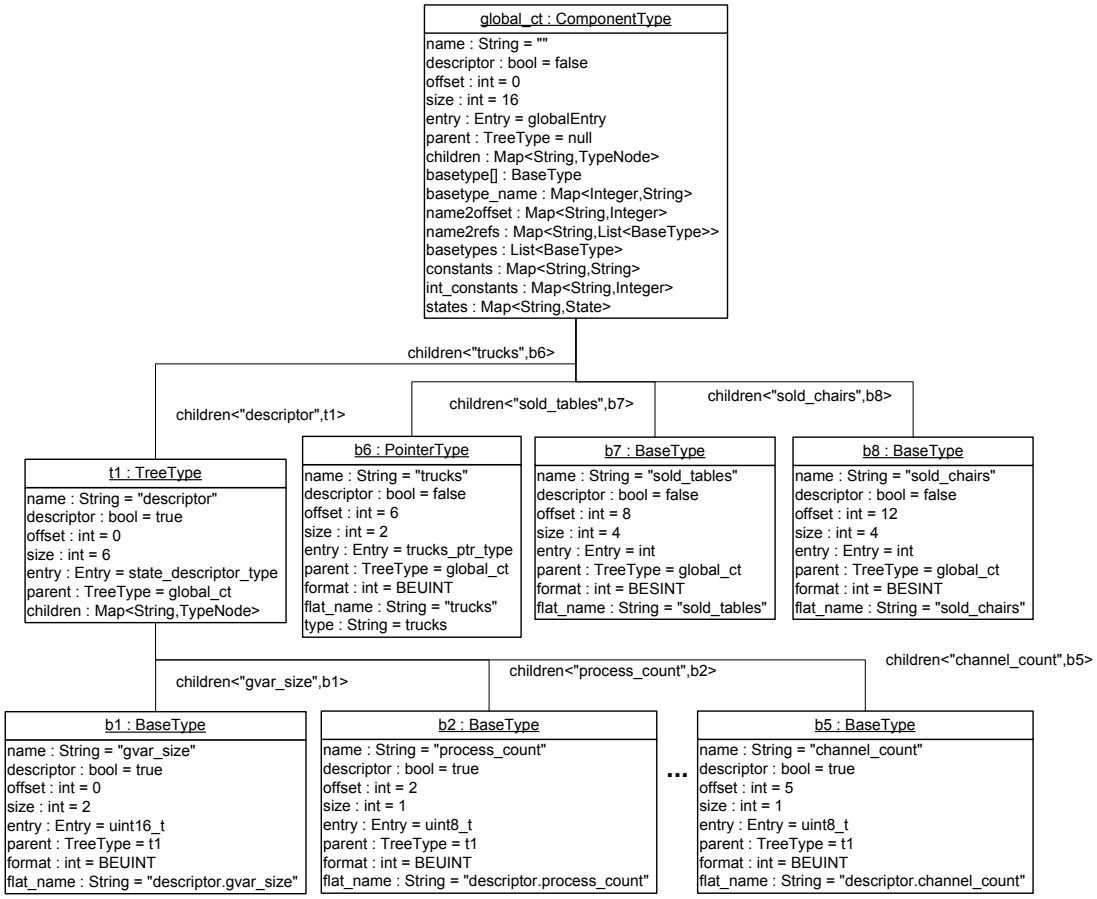



Figure 4.6: Furniture Factory - *Global Component Type* Object Diagram

component type tree $C$, when there is a path starting at the component type entry at the root of the tree $C$. $E_p$ and $E_q$ must be on this path and all variable, field and scope entries on the path between $E_p$ and $E_q$ (possibly none) are invisible if and only if $R(E_p, E_q)$.

**Definition 3** (Single value variable name)**.** If $C$ is a component type, then the name of a single value *value_name* in the memory defined by a component type $C$, is defined by

$$\textbf{String } value\_name = " name(E_1) \,.\, name(E_2) \,.\, \ldots \,.\, name(E_n)"$$

where $E_n$ is a variable that has a base type. For each two entries $E_p$ and $E_q$ whose name attribute appear in name $n$ separated by a dot, i.e. $value\_name = " \ldots \; name(E_p) \,.\, mame(E_q) \; \ldots "$, it holds that $E_q$ is a type tree refinement of $E_p$. Every base type instance has a unique name in the name space.

**Definition 4** (Multi value variable name)**.** If $C$ is a component type, then the name of a single value *multi_value_name* in the memory defined by a component type $C$, is defined by

$$\textbf{String } multi\_value\_name = " name(E_1) \,.\, name(E_2) \,.\, \ldots \,.\, name(E_n)"$$

where $E_n$ is not a base type entry, but instead is an entry tree with at least one base type leaf. For *attribute* $\in$ *attributes*$(E_n)$ and for each two entries $E_p$ and $E_q$ whose name attribute appear in name $n$ separated by a dot, i.e. *multi_value_name* = " ... *name*$(E_p)$ . *name*$(E_q)$ ...", it holds that $E_q$ is a type tree refinement of $E_p$.

The component name space also contains constants. Constant names are constructed from the variable, scope and type names just like variable names, except that a constant name ends with and attribute.

**Definition 5** (Constant name)**.** If $C$ is a component type, then the name of a constant *constant_name* in the memory defined by a component type $C$, is defined by

**String** *constant_name* = "*name*$(E_1)$ . ... . *name*$(E_n)$ . *attribute*"

where *attribute* $\in$ *attributes*$(E_n)$. For each two entries $E_p$ and $E_q$ whose name attribute appear in name $n$ separated by a dot, i.e. *constant_name* = " ... *name*$(E_p)$ . *name*$(E_q)$ ...", it holds that $E_q$ is a type tree refinement of $E_p$.

The definition purposely does not describe the names of type constants of sub types that appear in the component type tree. The reason for this is that variables and types can have the same attributes and their names would be the same. In order to prevent ambiguity and avoid a name space clash, the name of type constants can be augmented with the string *"type"*.

**Implementation.** A brief description of the Java implementation of SDI type structures is sketched. Component types are represented by the `ComponentType` class, which contains flat look-up structures that can be queried in the following way. The `ComponentType` class contains a `Map` from which `BaseType` instances can be retrieved, using the (single value) variable name of the base type instance as a key by means of the function:

`BaseType getBaseType(String name).`

The component type contains a `Map` from which sets of `BaseType` instances can be retrieved using the name of the multi-value variable name as an input parameter of the function:

`List<BaseType> getBaseTypes(String name).`

The component type contains an array of base types, which allows the retrieval of base type instances for an offset within the component type bounds using the function

`BaseType getBaseType(int offset).`

The type tree can be *navigated* using the functions `getType(String name)` to get child type trees and `getParent()` to get the parent type tree. A base type instance is a `BaseType` object, which is used to get and set values in the memory. The translation between a `String` or `int` value and a sequence of bytes is made using the *format* and *size* attributes. Sets of base type instances are computed for names that do not denote base type instances.

## 4.4 State API

This section describes the SDI Framework component API. A component is accessible through different API functions, which make use of the look-up functions defined in the component type.

*State components* (or components for short) are central to the SDI Framework. A component is a self-contained run-time entity that exists in the memory of a running program. Each component starts at a certain offset and is stored a sequence of bytes in the state vector. Components may

grow and shrink in size dynamically and may be finite entities that can be created and destroyed. Typical components within a state can be *processes*, but also *objects* or *meta structures* used in a VM. The components may be separately regarded and altered, retrieved from and stored in the state-space allowing the application of *collapse compression* [19].

Each component is associated with a component type which determines the debugging API for the component. The component API is composed of functions for retrieving constant and variable values and setting variable values. Variables can be accessed by *name*, or by *offset* within the component. The reason a component and its type are separated is that it facilitates a generic component API. In contrast, combining a component with its type yields classes of components with predefined component specific API functions, which require hard-coded function calls.

Section 4.3.1 explains how the variable and constant names are created when a component type tree is flattened. The name of a variable or constant is the argument for the API function that retrieves the constant or variable value. The name of a variable and a new value are the arguments for the API function that sets the variable value.

**Component Descriptors.** Recall that in Section 3.2 it was defined that each run-time component has a *descriptor*. Component descriptors contain variables that describe what is in the component or meta-information about the run-time state of the component. The descriptor variables we describe are its *type*, its *identifier* and its *size*. The *component type* variable relates an instance of an SDI component type to the memory. A look-up in the symbol table with its value yields the component type. The *component identifier* gives the component a unique name and place within its parent state. Components are stored in the state by increasing *component id*. If the descriptor contains a *size* variable, the size of the component is assumed to be dynamic. If the descriptor does not contain a size variable, the component size must be statically defined by use of the size attribute. If the size is dynamic, the *size* variable value can only be accessed by use of the *component type*, which defines its *memory offset*, its *number of bytes* in the memory and its the *byte format*.

**Component Hierarchy.** In the SDI Framework, a high level representation of a state consists of a hierarchy of state components, each having a component type that facilitates the component API. State components may themselves contain a state variable which consist of separate components. An important design decision is that, instead of allowing several state variables to exist inside one component, we allow only one state variable per component type. This does not restrict the hierarchy of components, since a state variable may still contain multiple components, but it ensures a well-defined name space. The name of the state variable or the name of the component it is part of uniquely denote the parent name of any possible sub-components in the name space. Furthermore, the static offset of the state variable must be the highest offset of any variable within the component. This allows the state to grow without the risk of growing out-of-bounds within its parent component, avoiding the overhead of run-time bound checks.

**Component API Implementation.** States are represented in the framework by a `State` class and components by the `StateComponent` class. Both classes are stored in the *state* package, which is depicted as a class diagram in Figure 4.7. The `StateComponent` class contains the component API functions. The function `getValue(String name)` retrieves the value of the base type instance represented by *name*. The function `getConstant(String name)` retrieves the value of the constant value of *name*. The *value* argument must also be given for setting a variable value. It is translated to bytes using the basetype format attribute. The function `setValue(String name, String value)` sets the value of the base type instance represented by *name*.

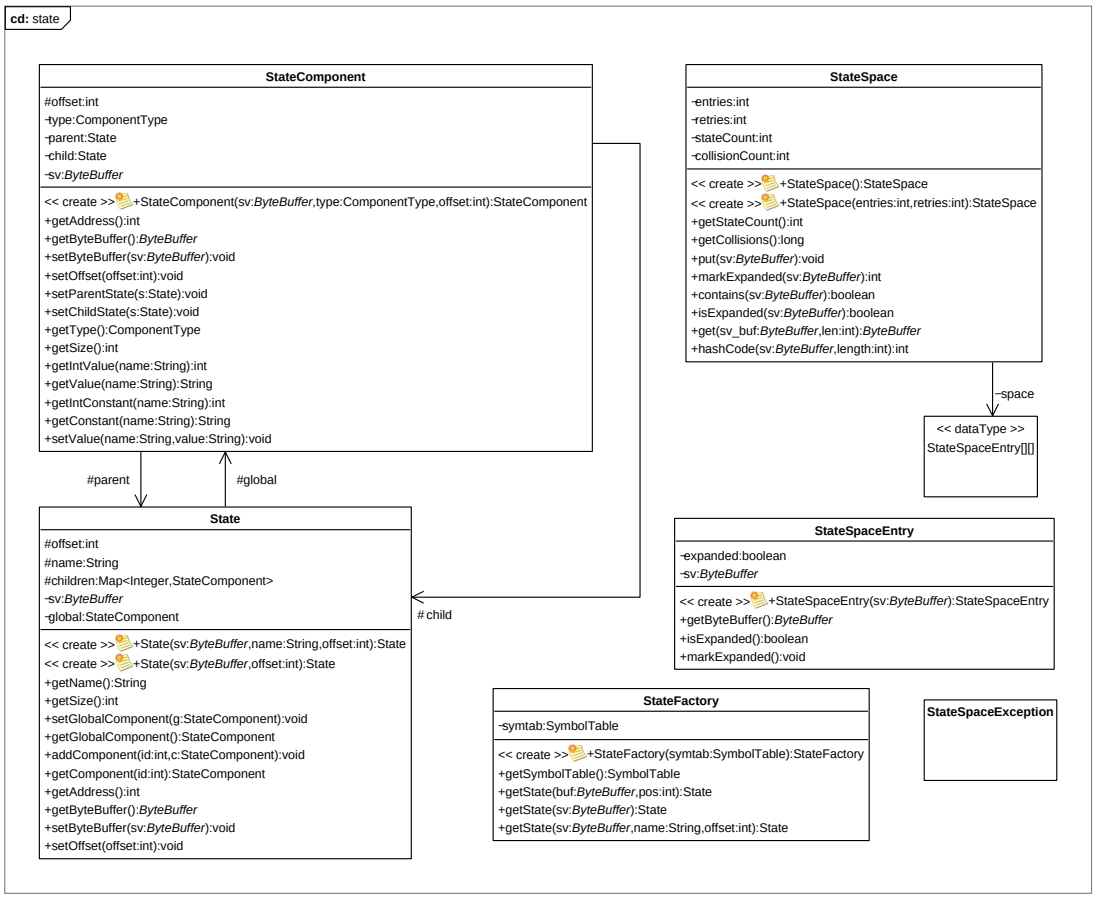



Figure 4.7: SDI Framework - *state package* Class Diagram

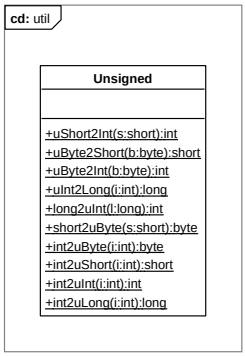



Figure 4.8: SDI Framework - *util package* Class Diagram

The *util* package contains the `Unsigned` class, which is used by the component API to retrieve and store values of variables with an unsigned type. These values must be encoded, since unsigned types are not native to Java. The conversion methods can be seen in Figure 4.8.

### 4.4.1 State Factory

The previous section explained how component types provide an API for components. This section sketches a comprehensive approach for generating state components from state vector BLOBs using SDI descriptions.

A *state factory* is a factory class which provides methods for producing *high level states* (composed of components) from a state vector. It associates the components it produces with component types and memory locations within the state. A tool that provides the state vector inputs for the state factory can be seen as its supplier. The state factory is necessary to produce the memory model of a running program as a queryable structure.

We require that the SDI Framework can be reused in different designs. It should be easy to integrate the state factory into an existing architecture. Furthermore, we would prefer not have to design a state factory for every different type of state vector and for every different tool. We would like to have a generalized state factory that can be reused to process state vectors from different kinds of tools.

SDI *state* entries are used to describe memory areas that contain state components. The components contained in the memory area must be identified and associated with a component type by the state factory. This should be done based on the format of the state vector which describes how the components are stored. The format is either described as a discrete algorithm for state generation, which is tied to a specific state format, or based on a generic algorithm. An example of a discrete algorithm for state generation is given in Algorithm 1 in Section 2.1. In this section we give a generic approach for state component generation. The generic state component production algorithm defined in Algorithm 2 is an integral part of the SDI Framework. The algorithm is used to describe the design of a generic state factory that can be reused with different state formats.

The design of the generic state factory is currently being implemented as a generalized factory class for state production. In the generic approach algorithm defined here, the type identifier is required to be in the first byte of the component descriptor. The reason for this requirement is that it would otherwise not be possible to use SDI to determine the type of the component. More drastically, without the *type identifier* variable it would not be possible to identify components in the memory at all.

Algorithm 2 describes a generic algorithm for state production. We assume there is a global component in the top level state. The components are all assumed to be dynamically sized, which means the components can grow and shrink over time. We assume that the size is stored in a variable called *descriptor.size*.

**Algorithm 2** (Algorithm for State production)**.** *generate*($sv$) is a function that generates state components and returns a *State*, which contains a graph of the generated components. Let input state vector $sv$ be an array of bytes of length $n$ be the input where $sv[i]$ denotes the $i_{th}$ byte. Let $c$ be the current component, let $a$ be the current component address, let *c.offset* be the offset of the component within $sv$ and let *c.type* be the component type of $c$. Let $st$ be a symbol table symbol table with type look-up functions. *st.getComponentType*(*int* $t$) returns the component type $ct$ with $id = t$. Finally, let *c.getValue*(*String* $n$) be the function that retrieves the value of variable $n$ from the component.

1. **Base.** Then the first step of the algorithm is to generate the global component $gc \in$ *State*. The offset *gc.offset* of the component is 0. The type *gc.type* of $gc$ is the global scope type. The size of $gc$ is in the variable named *descriptor.size*, of which the value *gc.getValue*("*descriptor.size*") represents the dynamic size of the component.
   (a) If $gc$ does not contain sub-states then the next component starts at $a = a +$ *c.getValue*("*descriptor.size*"). Goto step.
   (b) If $gc$ contains a substate $sv_s$ then apply Algorithm 2 to $sv_s$ starting at step. Goto step.
2. **Step.** The second step of the algorithm is to generate state component $c \in$ *State*, which has a component type determined by type identifier at location $sv[a]$. The component type *c.type* is *st.getComponentType*($sv[a]$).
   (a) If $sv[a] = 0$ then the type is a null-type and there is no component at this location, terminate the algorithm for this $sv$ return *State*.
   (b) If $c$ contains a sub-state $sv_s$ then apply Algorithm 2 to $sv_s$ starting at step. Goto step.
   (c) Otherwise, generate a new component with offset *c.offset* $= a$. The size of $c$ is in the variable named *descriptor.size* of which the value *c.getValue*("*descriptor.size*")) represents the dynamic size of the component. The next component starts at $a = a +$ *c.getValue*("*descriptor.size*"). If the current component address is smaller than the length of the state vector, i.e If $a < n$ goto step else return.

We observe in Algorithm 2 that sub-state vectors may be of a variable size and that, in order to compute the size of the component with a state variable, the size of the sub-state must first be computed. This must be done by the state vector generator which is the supplier of the state vectors for the algorithm. The generic state format with sub-states we create by use of Algorithm 2 requires the state vector generator to compute the component sizes. This may be done by performing a top down traversal of component structures. The size of each of the components within a sub-state is returned to the parent component.

We also observe that Algorithm 2 does not provide a canonical ordering of components within the state vector. The same component must be stored at the same location within the state vector for two states to be equal. The SDI Framework does not interfere with this internal tool design choice.

### 4.4.2 Modeling Notation

Algorithm 2 is best explained using an example. To this end, we first extend the modeling notation introduced in Section 3.2.1 with components, top level states and the null-type, in order to model states.

**States.** The abstract modeling notation for *states* is depicted in Figure 4.9. The *top level state* is a run-time entity depicted as a white rectangle. Top level states are implied and do not appear explicitly in the SDI notation. The rectangle line may be thicker than that of other shapes, to clarify the shape is a state. States consist of components but we also say states contain components. The line connector is used to denote that a state contains a component. Components are depicted below states.


Figure 4.9: States Legend

**Components.** The modeling notation for *components* is depicted in Figure 4.10. Although components have the shape of rectangles, they are not SDI entries. The white component colour signifies components are part of the SDI Framework. *Component types* have the shape of rectangles with rounded edges. Components have a component type which is denoted as usual with an arrow.

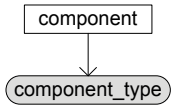



Figure 4.10: Components Legend

**Null-Type.** The null-type is depicted in Figure 4.11. It is a special pre-defined component type which signifies the absence of a component in the state. As the name implies, its *type_id* attribute value is zero. A state that does not contain components contains a *null-type*.

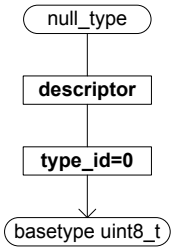



Figure 4.11: Null-Type Legend

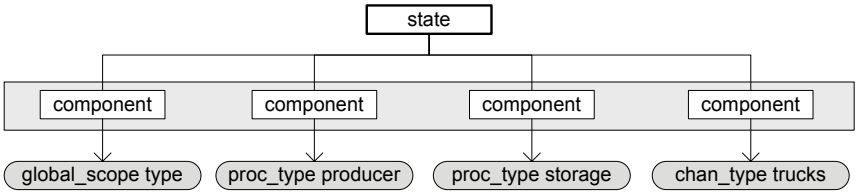



Figure 4.12: Furniture Factory - Example State

In Chapter 3, we used the running example of the Furniture Factory. The example Nips VM state of Figure 3.6 can be modeled as an SDI Framework State, as shown in Figure 4.12.

### 4.4.3 Threads Example

In this example we abstract from the modeling language and the program semantics, focusing just on the memory model. We reason that the NIPS VM uses Algorithm 2 for state production instead of Algorithm 1. The global scope contains a variable *a* which is of the *int* basetype. There are two types of processes. The first process type is called *p1*, it has a variable *b* which is of the *short* basetype. The second process type is called *p2*, it has a variable *c* which is of the *byte* basetype and a variable *s* which is a *state*. The type id of *p1* is a byte with the value 1. The type id of *p2* is a byte with the value 2.

In the Threads Example, the type trees of the component types are constructed before Algorithm 2 is applied to a sample state vector. The example is concluded by modeling the resulting state using the extended modeling notation.

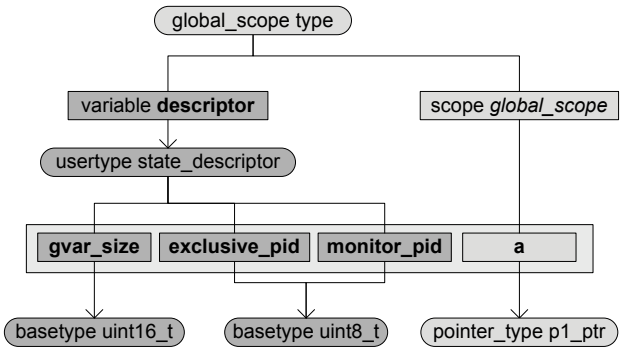



Figure 4.13: Threads - Global Component Type Name Tree

Figure 4.13 shows the global component type for the Threads Example. The state descriptor user type has been altered. The process_count and channel_count variables needed by Algorithm 1 to identify Process and Channel components are no longer needed.

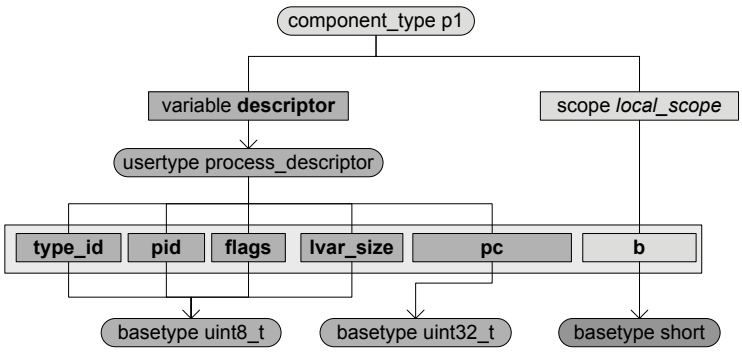



Figure 4.14: Threads - *p1* Process Type Name Tree

Figure 4.14 shows the *p1* process component type. The process descriptor user type has also been altered. The *type id* is now the first byte in the descriptor. This allows the component and the type to be identified within the memory.

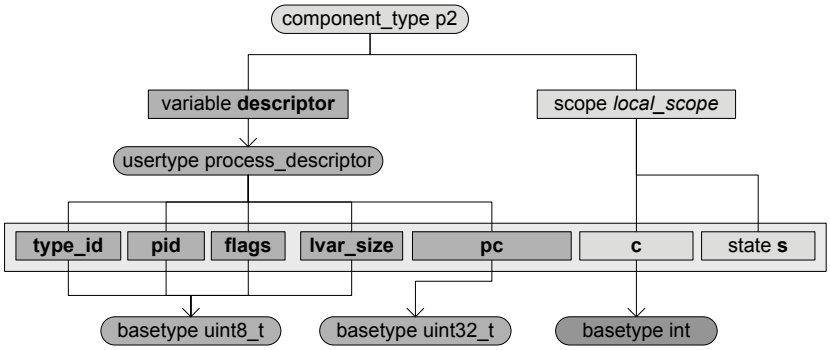
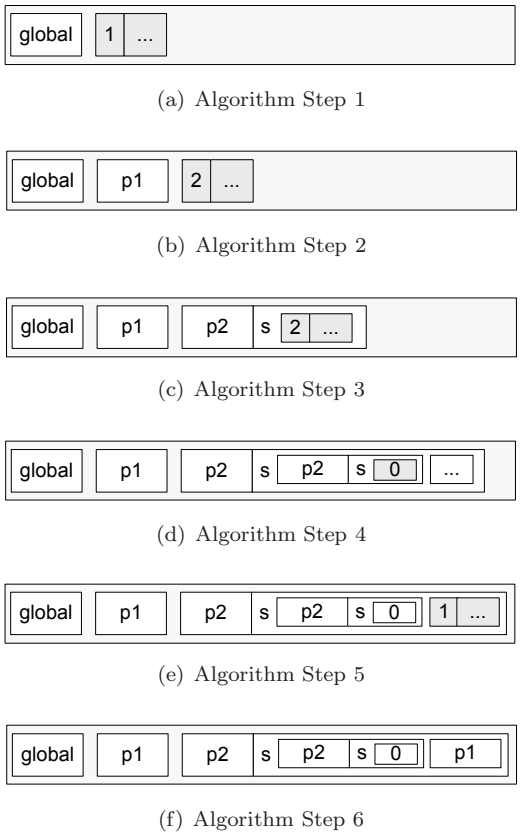



Figure 4.15: Threads - *p2* Process Type Name Tree

Figure 4.15 shows the *p2* process component type. The process descriptor is the same as in the type tree of *p1*. The process contains a state variable *s* which may contain more state components.




(a) Algorithm Step 1

(b) Algorithm Step 2

(c) Algorithm Step 3

(d) Algorithm Step 4

(e) Algorithm Step 5

(f) Algorithm Step 6

Figure 4.16: Threads - Example State Production

Figure 4.16 schematically shows how Algorithm 2 is applied to an example state vector. Each of the sub-figures represents a step in the application of Algorithm 2. The outer rectangle represents the state vector or top level state. The gray rectangles represent the next part of the state vector that should be analysed according to the algorithm. Rectangles within the top level state are level one components. Components within level one component sub-states are level two components, and so on. In the Threads Example, we assume that the altered Nips VM is able to assign the proper component identifier to each of the components and sub-components.

- **Step 1:** Figure 4.16(a) shows the first step of the state production. The global component is a level one component which always exist. The global component is produced, its size is determined and then the next component must be identified. Its *type id* is 1.
- **Step 2:** Figure 4.16(b) shows the second step of the state production. A level one process of type *p1* is produced, its size is determined and then the next component must be identified. Its *type id* is 2.
- **Step 3:** Figure 4.16(c) shows the third step of the state production. A level one component of type *p2* is produced. Processes of the type *p2* have a sub-state variable *s*. This variable must be analysed before the next level one component can be identified. The sub-state variable *s* contains a second level component of which the *type id* is 2.

- **Step 4:** Figure 4.16(d) shows the fourth step of the state production. A second level component of type *p2* is identified. The sub-state variable *s* must be analysed before the next second level component can be identified. It contains a third level component of which the *type id* is 0.

- **Step 5:** Figure 4.16(e) shows the fifth step of the state production. A null-type is identified. This means that the second level *p2* process variable state *s* does not contain any components. The next second level component must now be identified. Its *type id* is 1.

- **Step 6:** Figure 4.16(f) shows the sixth step of the state production. A second level component of type *p1* is produced. The combined size of the components is the size of the state vector. The algorithm terminates.

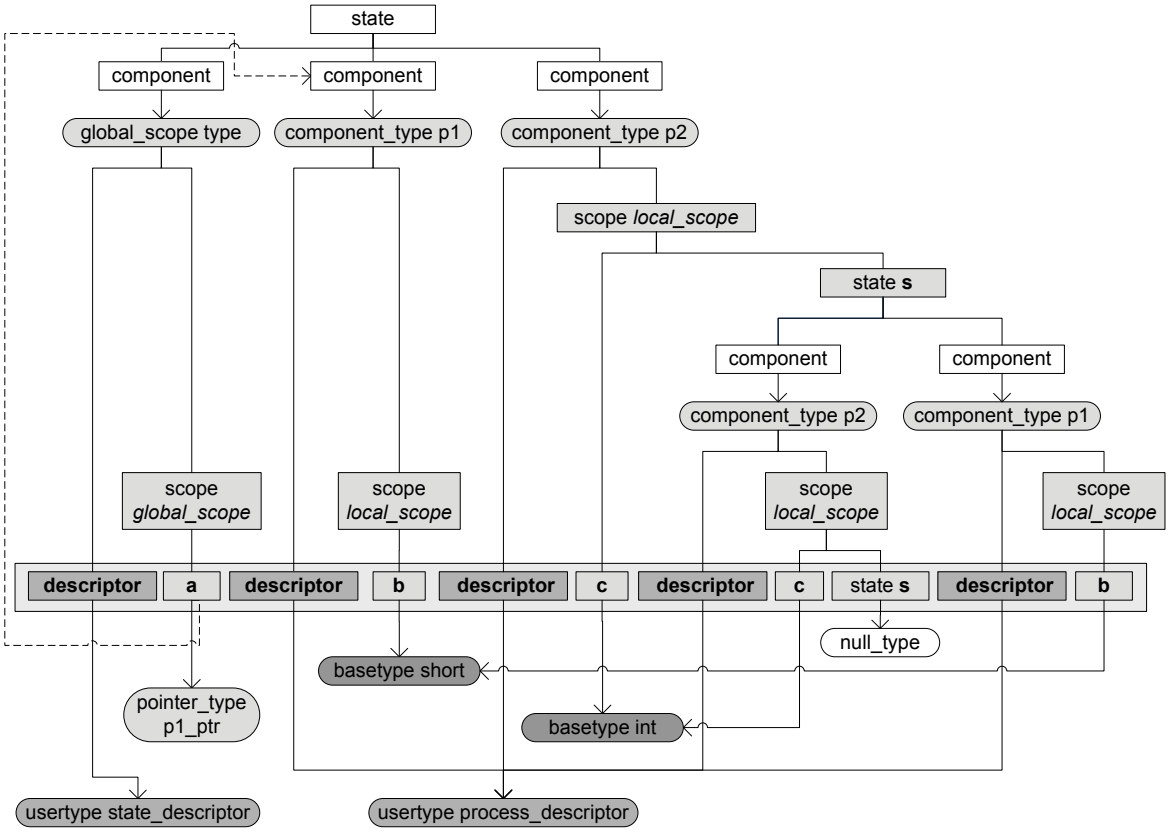



Figure 4.17: Threads - Example State

The result of the application of Algorithm 2 to the example state vector sketched in Figure 4.16 is shown in Figure 4.17.

## 4.5 Transition API

Mutations to state components can be seen as *transitions*. A semantics for the *transitions* of state components can be used to describe, log and store the behaviour of a model which can be replayed whilst the high level representation of the memory model is kept up to date. Transitions can also be used to define changes to states and apply or unapply them. We extend the SDI Framework with a *Transition API* that uses *transitions* to improve the state manipulation framework. A debugger can use SDI to display the low level changes made by transitions to the user at source level.

**Requirements.** As a primary requirement, we pose that the design facilitates a debugger with respect to logging behaviour, editing and displaying states. We require the design to provide *component transition primitives* that allow the manipulation of the high level representation of states and as a hierarchy of components as is described by the SDI Framework in Section 4.4. We require transitions to be *bi-directional*, which means that for every transition there must be a reverse transition that undoes the changes made by the transition. We require that we can *apply* transitions to states and *produce* transitions from changes that have happened to states such that transitions can be logged, analysed using a debugger and rolled back. We would like to be able to store transitions in a compact format, since transitions might be used during model checking to store on the search stack instead of state vectors as e.g. in JPF [48].

**Means.** First we define the component transition semantics in terms of *transition primitives*. The semantics should contain primitive operations for *component creation* and *component disposal* because state components may have a limited life-span. Another primitive operation is to *set the size* of a state component, since state components may grow and shrink during a transition. Because state component variable values may change, the need exists for an operation that *changes* a sequence of bytes of a component to a new value. Each of the transition primitives we identified can only be applied to make the correct changes to the state if there is a way to *select* the component the transition primitive works on. We use the *component id*, which we assume is present in the component descriptor, to select the component by an operation.

**Design.** The design of the *Transition API* is as follows. The operations for component *creation*, *disposal*, *resizing* and *changing values* are described using *transition primitives*. A *transition* is described using a sequence of instructions derived from the transition primitives. This is much like editing a stream of data such is done with SED [27].

The instructions are used to design a *transition factory* that can be used to *produce* transitions from state vector inputs and a *transition machine* that can be used to *apply* the semantics of a transition to a state. The transition factory provides a means for logging or replaying the system behaviour. The transition machine offers primitive methods that apply transition instructions as a means for state manipulation.

### 4.5.1 Component Transition Instructions

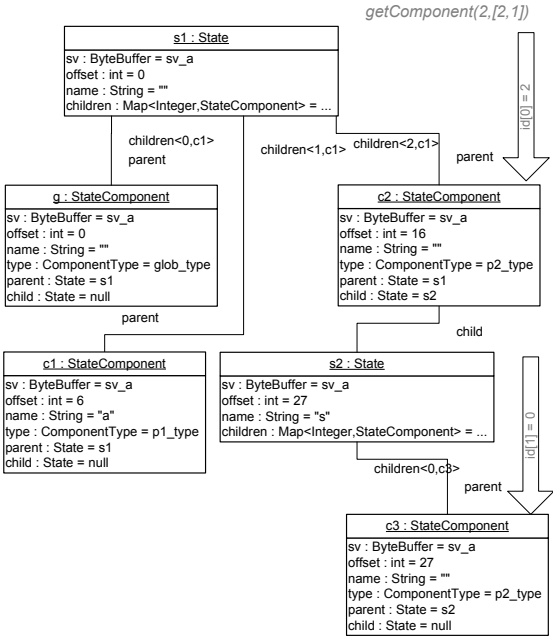
For each component we require there exists a variable *descriptor.size* that stores the component size, a variable *descriptor.id* that contains the component id and a variable *descriptor.type_id* that contains the component *type id*. This enables the transition machine that processes the transition semantics to make use of the state API. It can also notify the observers of the components and facilitate a debugger using a model view controller.

The semantics of the component transition instructions and their arguments are listed in Table 4.1.

| instruction | arguments | semantics |
|---|---|---|
| `select` | *depth id* | selects a component for an mutation |
| `new` | *depth id type size values* | creates a new component |
| `dispose` | *depth id* | destroys a component |
| `resize` | *depth id size* | resizes a component |
| `change` | *depth id offset val_size val* | changes component values |

Table 4.1: Component Transition Instructions

**Select.** Each instruction requires the selection of the component it works on. The *depth* argument in each of the instructions refers to the depth of the state component in the hierarchy of components, with zero being the top level state. A state variable in a component that is in the top level state is a second level state and so on. The *id* argument is the sequence of component ids to be traversed to reach the component selected to undergo a mutation. The argument consists of *depth* component *id*s. Each *id* denotes a component on a state level, because a component can have only one state variable. The `select` instruction is defined by "`select` *depth id*". It acts as a cursor which is placed on a component.




(a) Threads Example - *select* Example

**TransitionVM**

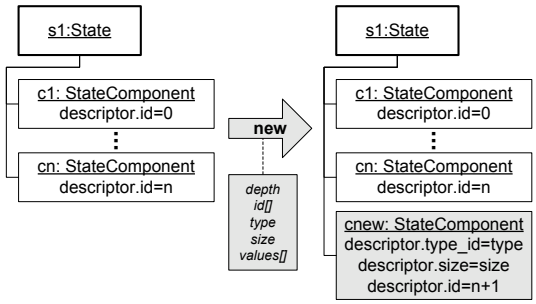
```
StateComponent getComponent(State s,
    byte depth, byte id[]){
  if(depth==0) return null;
  int level = 0;
  State cur_s = s;
  while(level<depth){
    if(level==depth){ //found component
      return cur_s.getComponent(id[level]);
    } else { //find next state
      cur_s=cur_s.getState(id);
      level++;
    }
  }
  //throw exception "component not found"
}
```

(b) `TransitionVM` *select* Java Pseudo Code

Figure 4.18: Component *selection*

Figure 4.18 gives a description of component selection, Figure 4.18(a) shows the selection of a process in the Threads example and Figure 4.18(b) shows the Java pseudo code for the `getComponent` method in the `TransitionVM` class.

**New.** A transition in which a component is created contains a `new` instruction. The creation of a component requires we know its component *type*. The link between a component and its type is preserved by storing the *type* argument value in the component descriptor in the *type_id* variable.




(a) *new* Transition Schema

**TransitionVM**

```
StateComponent newComponent(State s,
      byte depth, byte id[],
      byte type, byte size, byte values[]){
  //get the parent state and the offset
  if(depth>0){
     StateComponent prnt =
        getComponent(s, depth, id);
     State prnt_state = prnt.getState();
     int prnt_sz = //new parent component size
        prnt.getIntValue("descriptor.size")+size;
     prnt.setIntValue("descriptor.size",prnt_sz);
  } else {
     prnt_state = s;
  }
  int offset = prnt_state.getOffset(id);

  //create the new state vector
  byte sv[] = new byte[s.getSize()+size];
  //copy old sv up to offset
  sv[0..offset] = s.sv[0..offset];
  //copy (insert) new component values
  sv[offset..offset+size] = values;
  //copy old sv after offset
  sv[offset+size..sv.length] =
     s.sv[offset..s.getSize()]

  //create the new component object
  ComponentType ct =
     symboltable.getComponentType(type);
  Component c = new Component(sv, ct,
        offset);
  prnt_state.addComponent(c);
  s.setStateVector(sv);

  //creates new component observer
  prnt_state.notifyObservers();
  return c;
}
```

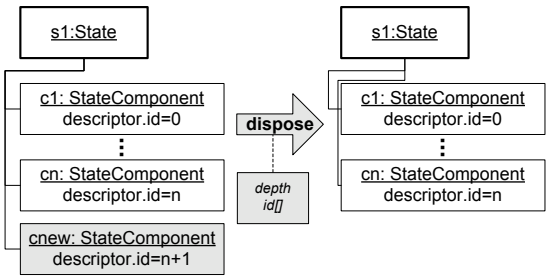
(b) `TransitionVM` *new* Java Pseudo Code

Figure 4.19: *new* Component Instruction

Figure 4.19 gives a description of the `new` instruction, Figure 4.19(a) shows the schema of the creation of a component and Figure 4.19(b) shows the Java pseudo code for the `newComponent` method in the `TransitionVM` class. The instruction for the creation of a component is defined by "`new` *depth id type size values*." The two arguments *depth* and *id* determine in which component state variable a new component is created. The *type* argument is needed to determine the component type of the newly to be created component. The argument is sized one byte and stored at offset zero within the descriptor of the component. This allows the use of Algorithm 2. The new component's id is determined by the *values* argument, this identifier must be unique to the component within the state. The components are stored by increasing *component id*. Although it seems logical that components always start their existence with the same size and

start *values*, this is not necessarily true if we consider inverse transitions. Components may be destroyed at any time and components may grow and shrink during their lifetime. Therefore, the component *size* must be added as an argument. Furthermore the *current values* of the component must also be restored. The inverse of the `new` instruction is the `dispose` instruction and vice versa.

**Dispose.** A transition in which a component is disposed contains a `dispose` instruction. The instruction for the disposal of a component is defined by "`dispose` *depth id*". The two arguments *depth* and *id* are used to determine which component is destroyed.




(a) *dispose* Transition Schema

**TransitionVM**

```
void disposeComponent(State s,
      byte depth, byte id[]){
  StateComponent c =
     getComponent(s, depth, id);
  int size = c.getSize();
  int offset = c.getOffset();
  State parent_state = c.getParent();
  parent_state.remove(c);

  //create the new state vector
  byte sv[] = new byte[s.getSize()−size];
  sv[0..offset] = s.sv[0..offset];
  sv[offset..sv.length] =
     s.sv[offset+size..sv.getSize()];
  s.setStateVector(sv);

  //destroys component observer
  parent_state.notifyObservers();
}
```

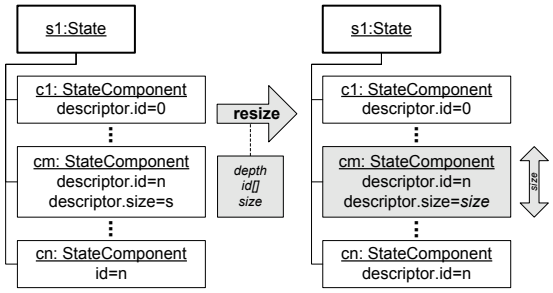
(b) `TransitionVM` *dispose* Java Pseudo Code

Figure 4.20: *dispose* Component Instruction

Figure 4.20 gives a description of the `dispose` instruction, Figure 4.20(a) shows the schema of the disposal of a component and Figure 4.20(b) shows the Java pseudo code for the `disposeComponent` method in the `TransitionVM` class.

In the design we must take into account that it may be the case that only the component with the highest *id* is allowed to be destroyed, while the other components that can be disposed of are queued for removal. This may be the case when the equals operation on state vectors naively compares the bytes within the state vectors and does not compare components. In such a case, it may be that the components must be aligned by default by means of this condition. Otherwise, state vectors are deemed unequal when they really signify the same thing and a state space explosion can be the result.

**Resize.** A transition in which a component is resized contains a `resize` instruction. The instruction for resizing components is defined by "`resize` *depth id size*". The two arguments *depth* and *id* are used to determine which component is resized. The *size* argument refers to the new component size in bytes. The size argument value is stored in the component descriptor. To undo a resize instruction the size must be reset to what it was before. The `resize` instruction is its own inverse.




(a) *resize* Transition Schema

**TransitionVM**

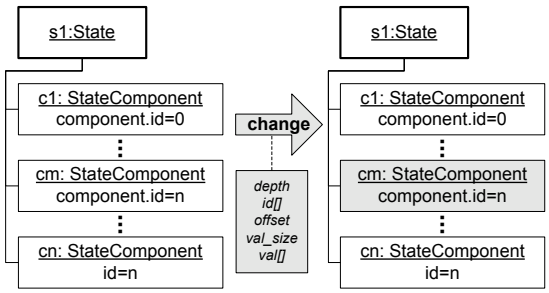
```
StateComponent resizeComponent(State s,
    byte depth, byte id[], byte size){
  StateComponent c =
    getComponent(s, depth, id);
  if(size == c.getSize()) return c;
  int offset = c.getOffset();
  c.setIntValue("descriptor.size",size);

  //create the new state vector
  int shift = size - c.getSize();
  byte sv[] = new byte[s.getSize()+shift];
  if(shift < 0){ //component shrinks
    //copy old sv including component (-
        shift)
    sv[0..offset+size] = s.sv[0..offset+size];
    //copy old sv starting from next
        component
    sv[offset+size..sv.length] =
      s.sv[offset+c.getsize()..s.getSize()];
  } else { //component grows
    //copy old sv including component
    sv[0..offset+c.getSize()] =
      s.sv[0..offset+c.getSize()];
    //pad sv with zeros
    sv[offset+c.getSize()..
      offset+getSize()+shift] = 0;
    //copy the rest of the old sv
    sv[offset+c.getSize()+diff..sv.length] =
      s.sv[offset+c.getSize()..s.getSize()]
  }
  s.setStateVector(sv);

  //updates view on state
  c.notifyObservers();
}
```

(b) `TransitionVM` *resize* Java Pseudo Code

Figure 4.21: *resize* Component Instruction

Figure 4.21 gives a description of the `resize` instruction, Figure 4.21(a) shows the schema of resizing of a component and Figure 4.21(b) shows the Java pseudo code for the `resizeComponent` method in the `TransitionVM` class.

**Change.** A transition in which component values are changed contains the `change` instruction. The instruction for changing bytes within a component is defined by "`change` *depth id offset val_size val*". The two arguments *depth* and *id* are used to determine of which component byte values are changed. The *offset*, *val_size* and *val* arguments determine at which offset within the component a number of *val_size* bytes will be changed to *val*. To undo the changes to a component, the values of the component must be changed back to what they were before the instruction was executed. The `change` instruction is its own inverse.




(a) *change* Transition Schema

**TransitionVM**

```
StateComponent changeComponent(State s,
     byte depth, byte id[], byte offset,
     byte val_size, byte val[]){
  StateComponent c =
     getComponent(s, depth, id);

  //create the new state vector
  sv = new byte[s.getSize()];
  //copy old sv completely
  sv[0..s.getsize()] = s.sv[0..s.getSize()];
  //write the new values
  sv[c.getOffset()+offset...
      c.getOffset+offset+val_size] = val;
  s.setStateVector(sv);

  //updates view on state
  c.notifyObservers();
}
```

(b) TransitionVM *change* Java Pseudo Code

Figure 4.22: *Change* Component Instruction

Figure 4.21 gives a description of the `change` instruction, Figure 4.22(a) shows the schema of changing the values of a component and Figure 4.21(b) shows the Java pseudo code for the *changeComponent* method in the *TransitionVM* class.

**Implementation.** An important implementation choice is how to represent the transition instructions. Since it can be useful to store transitions in a compact way, they are saved in transition vectors. It must be decided how many bytes are used for the instruction arguments. The number of bytes used per argument depends on the number of bytes used to store the information in the descriptor. A transition machine can be implemented in Java by the `TransitionVM` class which is discussed using pseudo-code in the examples. It offers primitive methods for state manipulation. A transition factory can be implemented in Java by the `TransitionFactory` class which creates transitions built up out of the instructions defined here as a means for logging or replaying the behaviour. The Java implementation is of the transition API is ongoing work.

## 4.6 Compiler Extensions

In this section, the design of the *SDI compiler extensions* is explained. Compilers were introduced in Section 1.1.2. The extensions can help a compiler programmer extend a compiler in order for it to generate SDI definitions, such that the debugging support offered by the SDI Framework can be used.

As a primary requirement we specify that it should be easy to implement the SDI compiler extensions into an existing compiler. Aside from the normal output, the compiler should also generate its symbol table consisting of *scopes*, *variables*, *types* and *location information* as a flattened SDI definition. We require that it should be possible to use the SDI Framework irrespective of the compiler design.

The means for the SDI compiler extensions are the SDI language description of Chapter 3, the compiler and the design of its symbol table. The SDI language is used to define the symbol table of the compiled program. The compiler determines the static memory locations and the sizes of types of variables within the memory. Source locations are implicitly introduced by generating byte code. A numbering of the instructions and the source line and column numbers is used to explicitly save this association between target and source locations. We may not assume that a fully constructed symbol table is ever available at compile time, since there exist compiler designs where the symbol table is never complete due to the design of opening and closing scopes as a lossy procedure. Even if the symbol table is not constructed as a static structure at compile time we should be able to use the SDI compiler extensions. Therefore, the best we can do is to provide functions that assist in generating SDI code.

**Design.** The design of the SDI compiler extensions is a collection of elementary code generation functions, called *emit functions*, that assist in the generation of SDI descriptions of symbol tables. Although we may not assume a scope exists as an object at compile time, we know that each scope is first opened, then defined and finally closed. Recall from Chapter 3 that all SDI entries must have names. The *open_scope(String name)* and *close_scope()* functions are defined that generate scope information for SDI. Alternatively, a scope that does not have a name meaningful name to programmers can be hidden using the function *open_scope()* instead. It generates a scope entry with a dummy name and a *marked* attribute which has the value *invisible*. We use the same approach for each of the SDI entries and provide open and close list functions for each of them. Attributes can be inserted in entries using the *insert_attribute(String name, String value)* function. Enumeration types which may be constructed from information gathered from different scopes should be treated differently. They should be stored in one place and emitted after the type is fully constructed.

**Implementation.** The Java implementation of the SDI compiler extensions is contained in the *emit* package. The class diagram of the *emit* package is depicted in Figure 4.23. The `SDIEmitter` class contains functions that are used to facilitate a structured way for generating the SDI specification in parts during the contextual analysis phase. The `EnumType` class serves as a storage class for enumeration types.

To use the SDI compiler extensions the following pattern must be followed. First an `SDIEmitter` object is created. The SDI definition list is opened using the `open()` function. Starting with an empty list, the debugging information is built up and stored in the `SDIEmitter` object using the provided functions. The entries are generated by opening them, inserting the required attributes and any other relevant attribute information using `insert_attribute(String name, String value)` and finally closing them. The SDI definition list is closed using the `close()` function.

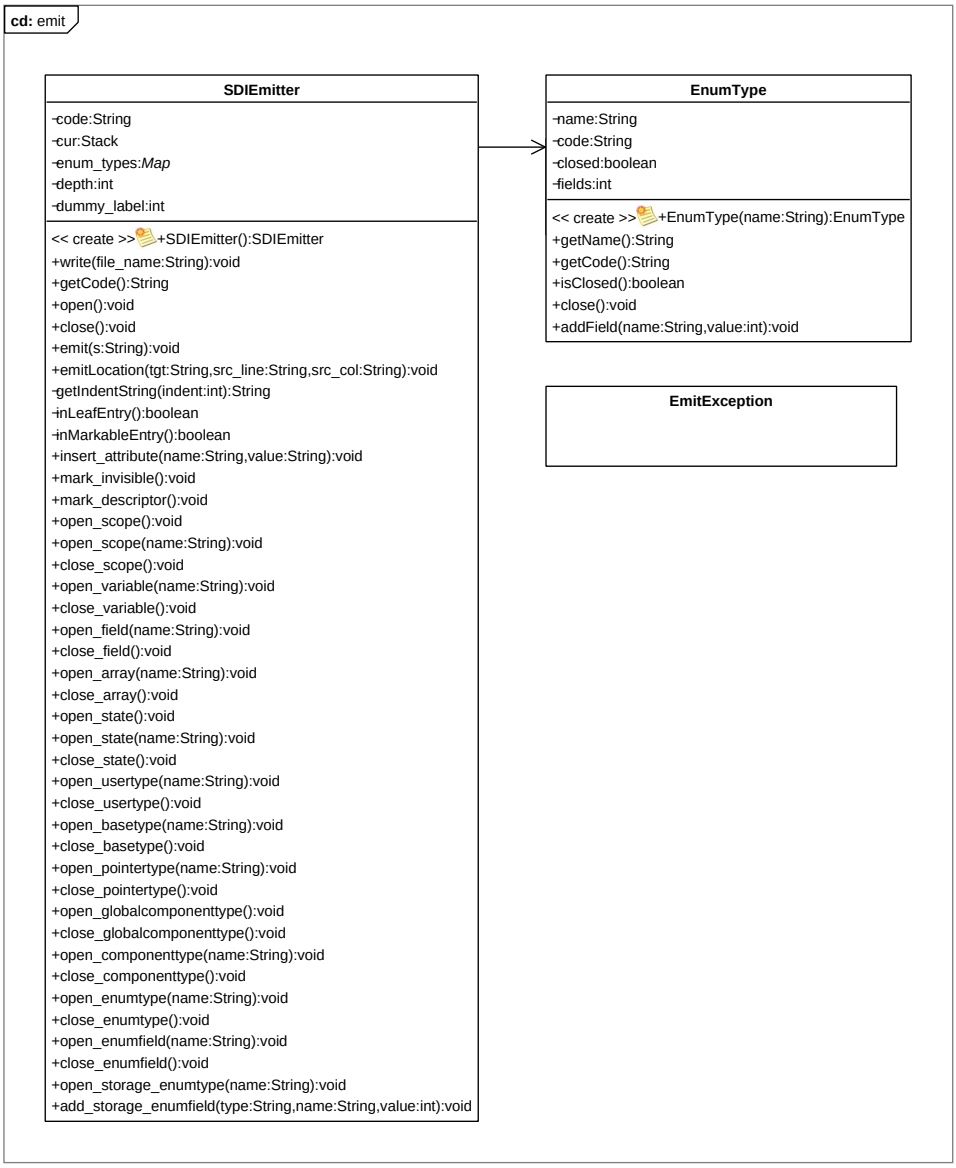



Figure 4.23: SDI Framework - *emit package* Class Diagram

The function `write(String file_name)` must then be called to write the SDI definition to a file.

**Evaluation.** The SDI Compiler Extensions can help inexperienced and experienced programmers alike to extend their compiler with SDI generation functionality. The debugging functionality of the SDI Framework thereby becomes available to a wider group of users.
The compiler extensions do not assist the compiler engineer in constructing a good compiler design. The design of the compiler is relevant to the use of the *code emitter* functions. Although

it is outside the scope of this thesis report, we mention that in case the fully constructed symbol table is intact after the contextual analysis phase, a *visitor pattern* can be used to generate the *program-constant SDI* based on the information contained within the symbol table entries. Alternatively, we can use *overloading* where each of the entry classes contains an overloaded method that is called recursively in order to acquire the debugging information. An example of this approach is the SDI extension of the Nips Promela Compiler, introduced in Section 2.1.4, which was extended using this approach. The implementation of the Nips Promela Compiler SDI extension is described in Appendix B.

## 4.7 The SDI Debugger

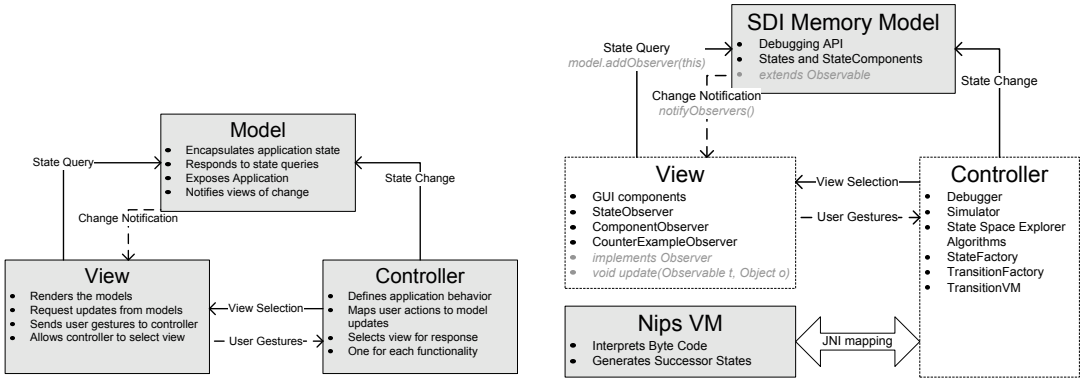
The SDI debugger is designed to be a tool that can graphically display the memory model of a running program when this model can be described using SDI. The debugger makes it easy for compiler and tool engineers to extend the feature set of their tools with a debugger, which can allows users to simulate and evaluate the behaviour of programs.

The main requirement of the SDI debugger is that it should provide users with a graphical representation of states and sequences of states, as well as with a means to choose transitions in an interactive simulation of a program. The graphical representation should be in terms of the program source. The debugger should function with all memory models that are supported by the SDI Framework. Finally, we wish to know if the Eclipse platform [11] can be used to create a Graphical user Interface (GUI) for states and components, since it offers a wealth of GUI components to choose from. The Eclipse platform has a modular design consisting of so-called plug-ins that offer interoperability for the tools using it.

**Design.** The design of the SDI debugger is incorporated in the SDI Framework. It uses the state API described in Section 4.4 for a source level view of run-time states. The Model View Controller (MVC) programming paradigm which is used to create an observer pattern for components in order to create a naturally modular design. An *observer* is an object which extends a graphical user interface component that provides a *view* on the data. The data it observes is referred to as the *model*. When the model is modified, the observer is notified of the changes and the it updates the modified fields on the screen.

The transition API introduced in Section 4.5 can be used in the observer pattern. Transitions are used to create and destroy components and their observers at the same time. The transition machine that processes the transitions is the *controller*, it changes the model (the state vector) and updates the view by notifying all concerned observers of the changes to the component memory.

Fortunately, the Graphical Editing Framework (GEF) [12] that is offered by the Eclipse platform makes use of the MVC paradigm. The design for the graphical component observers is not complete, since no design choices have been made about the exact GEF component to use for the different functionalities the debugger should have. It should provide a view on states and components. Moreover, it should display transitions and it should have a component that enables users to choose which transition to execute in an interactive simulation.
The design is complicated by the finite nature of components, because for each component an observer must be instantiated at the beginning and destroyed at the end of a component's life time. The architecture of the MVC paradigm is shown in Figure 4.24(a) and its application to the design of the SDI debugger is shown in Figure 4.24(b).




(a) General MVC Architecture (b) SDI Framework MVC Architecture

Figure 4.24: Model View Controller Architecture

**Implementation.** The implementation in Java is ongoing work. However, a schematic sketch of the implementation can already be given. We can use Java GEF components to create a GUI. GEF components, but also other GUI components such as `Frames` and `Panels` can be extend the `Observer` class. These GUI components are represented in this sketch by the `ComponentObserver` and the `StateObservers` classes. A `StateObserver` object provides a view of the current state of the memory of the running program A `ComponentObserver` object provides a view of the current status of a run-time component. `ComponentObserver` objects form a hierarchy, just like `Component` objects. Changes to both hierarchies occur simultaneously. The `Component` class implements the `Observable` interface such that `ComponentObserver` objects can register to this components (i.e. added to its observers) and *notified* of changes made to it by a *TransitionMachine* class that processes transitions.

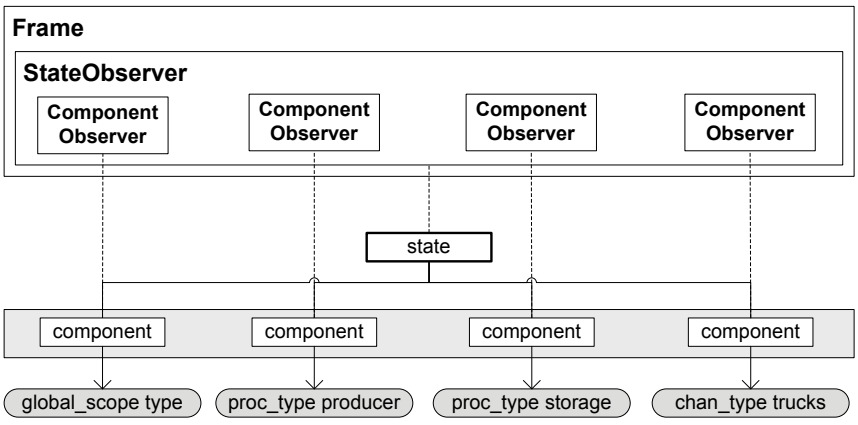



Figure 4.25: Furniture Factory - State Observer

Figure 4.25 shows a schematic view of the Furniture Factory example state. The dashed lines be-

tween the `Component` and the `ComponentObserver` objects denote that the `ComponentObserver` provides a view of the `Component`. Similarly, the dashed line between the `State` and the `StateObserver` objects denotes that `StateObserver` provides a view of the `State`.

## 4.8 Concluding Remarks

The SDI Framework is a framework for state manipulation. It provides functionality for viewing and altering values in state vectors by means of a *debugging API* that consists of a *state API* and a *transition API*. The SDI debugging API can be reused by compiler and tool engineers for different tool implementations, specifically Nips VM based tools. It easy for them to extend the feature set of their tools with a debugger which can simulate the behaviour of a program. The design of the SDI Debugger can be reused, since it works for all tools that use the SDI Framework, avoiding the need to design a new debugger for every tool. The debugging functionality can be shared between tools by using the SDI debugger and the SDI debugging API. The SDI framework can assist in the design of a modular explicit state model checking tool set, providing a means to connect and compare the tools.

It is not strictly necessary for the compiler to use byte code to represent an intermediate semantics to use the SDI framework. The SDI Framework can also be used with tools that have an internal compiler that does not produce intermediate byte code, provided that the tool can produce states as state vectors. Such tools may use internal structures such as an abstract syntax graph which is transformed to execute the program semantics. Note that without a byte code semantics it will not be possible to use the SDI target to source mapping described in Section 3.6, but source location annotations in type definitions will still be possible.

The SDI framework also has its limitations. The use of the debugger is limited to tools that are compatible with Algorithm 2. For tools that use a discrete algorithm for state production, such as Algorithm 1, parts of the SDI framework must be designed and implemented for the specific application.

| **Design Feature** | **Current Status** | **Package** | **Implementation goal** |
|---|---|---|---|
| SDI Parser | implemented | *ast* | reached |
| Symbol Table | implemented | *symboltable* | reached |
| State API | implemented | *state*, *symboltable* | reached |
| Transition API | designed only | *transition* | Transitions for state manipulation with `TransitionFactory` and `TransitionVM` classes |
| Compiler Extensions | implemented | *emit* | reached |
| Generic SDI Debugger | designed only | *sdi* | SDI MVC `Debugger` with `ComponentObserver` and `StateObserver` classes |
| SDI Language | implemented, designed pointer types | *state*, *transition*, *symboltable* | Full SDI including pointer types |

Table 4.2: SDI Framework Features Implementation Level

Furthermore, the design of the SDI Framework is not yet fully implemented. The Java implementation has a modular package structure. Table 4.2 shows for the different design features, the level of the implementation, the package that contains the implementation and the implementation goal. The difficulty in completing the implementation lies primarily in the implementation of the already designed SDI Debugger.

# Chapter 5

# Case Study

In the introduction of this thesis, the problem statement poses that the Nips VM generates states that cannot be displayed in a way the user understands. This chapter presents the Nips debugger, which provides the source-level debugging solution to this problem. This case study applies the SDI Framework to the design of extensions for Nips. It provides an opportunity for an in-depth examination of the application of the SDI Framework to a real-life example of an explicit state model checker.

Firstly, in Section 5.1 we explain the extensions to the Nips VM and the Nips Promela compiler that are necessary in order to support a debugger, as well as the design of the Nips debugger itself. We then evaluate the Nips Debugger and the design of the SDI Framework with respect to the amount of effort needed to implement it in a debugger, the amount of functionality offered and the usefulness of the results in Section 5.2. The chapter is ended with concluding remarks in Section 5.3.

## 5.1 A Debugger for Nips

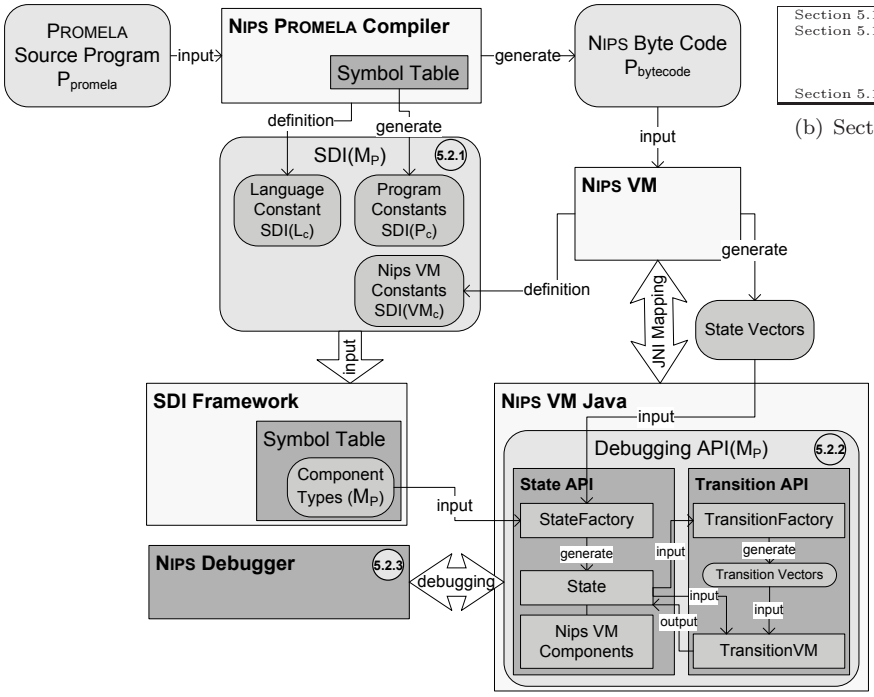
In this section the design and implementation of the Nips debugger is explained. First the requirements for the debugger are enumerated. Then the design and implementation are explained. We show that the SDI Framework fulfills the requirements for a debugger for Nips.

**Requirements.** A debugger for Nips should provide a source level representation of states that is esthetically pleasing, but it is mainly important that it provides debugging functionality. It should depict states and transitions and allow users to inspect counter-examples produced by Nips VM based tools. It should offer interactive simulation modes that allow the user to experiment with source model behaviours by choosing transitions, stepping forwards and backwards through the model as a Labelled Transition System (LTS). It should allow users to edit state variables such that users can step outside of the predefined model behaviour.

We want tool engineers to use the Nips VM as a back-end for their explicit state model checker and provide a debugger with as little extra implementation effort as possible. A debugger should be able to communicate with the Nips VM via a clearly defined debugging API. A debugging API consists of function calls that enable a debugger to display the information in Nips VM

states to the user at source-level, i.e. in terms of variable names and types. We neither wish to alter the semantics of the Nips byte code nor do we wish to alter the existing Nips VM implementation in order to fulfill the requirements.

**Design.** We use the SDI Framework, described in Chapter 4, in the description of the memory model of Nips states that facilitates a debugger for Nips by means of a debugging API composed of a state API and a transition API. The SDI Framework facilitates a debugging API that provides the function calls that allow a debugger to display and modify the information in Nips VM states. Figure 5.1(a) gives a schematic overview of the Nips debugging architecture and Figure 5.1(b) shows how the design is structured in sections.




(a) Nips Debugging Architecture

Section 5.1.1: Nips Memory Model
Section 5.1.2: Nips Debugging API
- State API
- Transition API
- Traces

Section 5.1.3: Nips Debugger

(b) Section Structure

Figure 5.1: Nips Debugging Design Overview

**Implementation.** We use the existing Java version of the Nips VM, which is a Java Native Interface (JNI) mapped version of its implementation in C. The Java implementation is extended with a debugging API based on the SDI Framework and with the Nips debugger. The changes made to the C implementation of the Nips VM are small and consist primarily of the addition of component type identifiers to the descriptors of processes and channels to allow identifying their type at run-time. The Nips byte code instructions `LRUN` and `CHNEW`, which respectively create a process and a channel respectively, receive an extra *type id* argument. Additionally, functions for querying the state-space size, the state count and the hash-table conflict count were added to `hashtab.c` in order to view state-space statistics.

Figure 5.2 is a class diagram that gives an abstract overview of the *nipsvm* Java package. The gray classes are already designed and implemented before. The white classes and packages are

added to the *nipsvm* base package. The state API is in the *nipsvm* package, the transition API is in the *transition* package, traces are supported by the *trace* package and the Nips debugger is in the *debugger* package

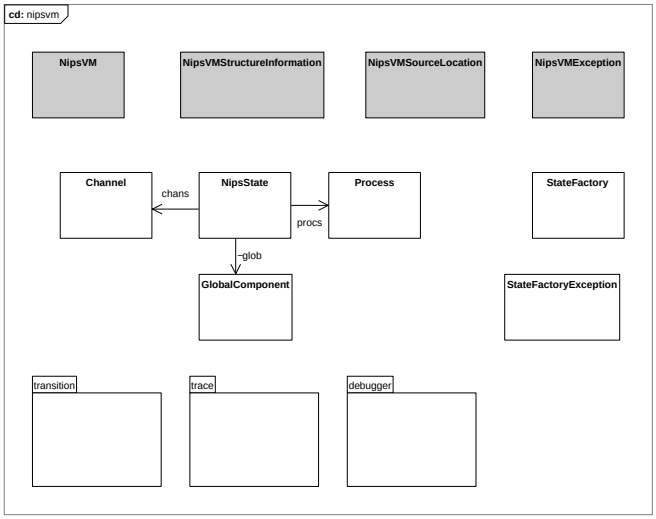



Figure 5.2: Nips VM Design Overview Class Diagram

### 5.1.1 Memory Model

In order to use the debugging API, the memory of programs running on the Nips VM must be modeled using SDI. For the Nips VM, constant SDI definitions of its run-time component, described in Section 2.1, must be modeled and maintained[1]. The SDI definitions of base types and component descriptors are shown in Figure 3.17 and Figure 3.20. The component *type id* has been added to process and channel descriptors to allow a mapping from the components to their respective component types. For the Nips Promela compiler, language constant information is modeled such as the Promela base types. Furthermore, the compiler has been extended to generate SDI definitions for each program representing the symbol table which describes the scope, variable and type definitions. Additionally, it generates location information which provides a Nips byte code to Promela source location mapping. The SDI Extensions to the Promela Compiler are described in Appendix B.

[1] Changes to the state format within the Nips VM have to be reflected within the SDI component descriptor definitions.

### 5.1.2 Debugging API

The design of the Nips VM debugging API is in line with the SDI debugging API described in Section 4.4 and Section 4.5. Therefore, only a brief description of the Nips debugging API is given. There are however some differences, since parts of the design predate the generic design of the SDI Framework and are specifically tailored towards the Nips VM. For each of the debugging API components, descriptions of its design and implementation are given. The argumentation for the design and implementation from Chapter 4 is not repeated, only the differences between the designs are explained.

#### State API

The Nips VM State API consists of a *state factory* based on Algorithm 1 instead of the generic Algorithm 2 and it produces the global, process and channel components as separate component sorts. Instead of retrieving components and returning C structs, as Algorithm 1 does, the state factory produces Nips component objects and the algorithm does not terminate until each conceptual component in the state vector is produced as an object. The Nips components, aside from inheriting generic API functions from the SDI State API, also contain specialized API functions. Furthermore, Nips VM states do not represent a hierarchy of components, but are a list of components ordered according to the state format defined in Section 2.1.

**Implementation.** The generic API functions are inherited by the `GlobalComponent`, `Process` and `Channel` classes which represent the Nips run-time components. Each of the classes contains specialized functions that complement the functionality offered by the SDI Framework. Figure 5.3 shows the class diagram of Nips VM state component classes in relation to the SDI Framework. Only the most important attributes and functions are depicted in the class diagram.

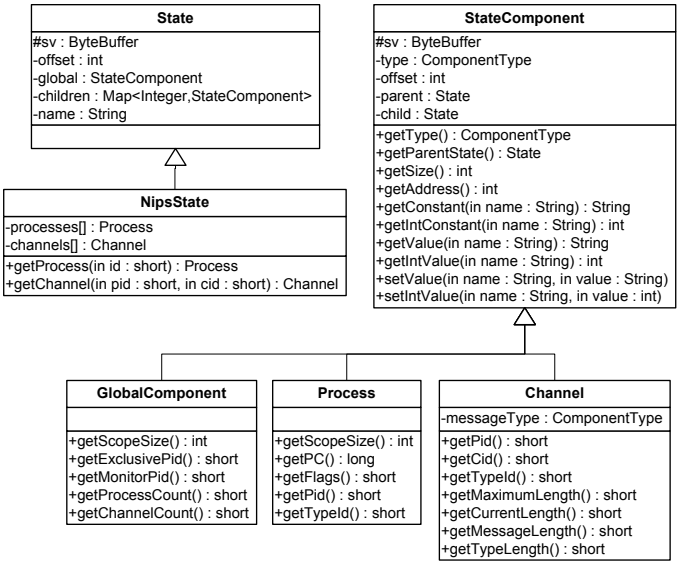



Figure 5.3: Nips VM State API Classes

The *nipsvm* package is extended with a state API. Figure 5.4 shows the class diagram of these extensions. A `StateFactory` object takes NIPS state vectors as input in the form of `ByteBuffer` objects and it returns `NipsState` objects as output. `NipsState` objects allow access to NIPS state components. `Process`, `Channel` and `GlobalComponent` each extend the `StateComponent` class from the SDI Framework. The SDI State API can be used to access variables and change variable values. Each of the components can be retrieved from a `NipsState` using an API function. The `Process`, `Channel` and `GlobalComponent` classes contain API functions to directly read descriptor values.

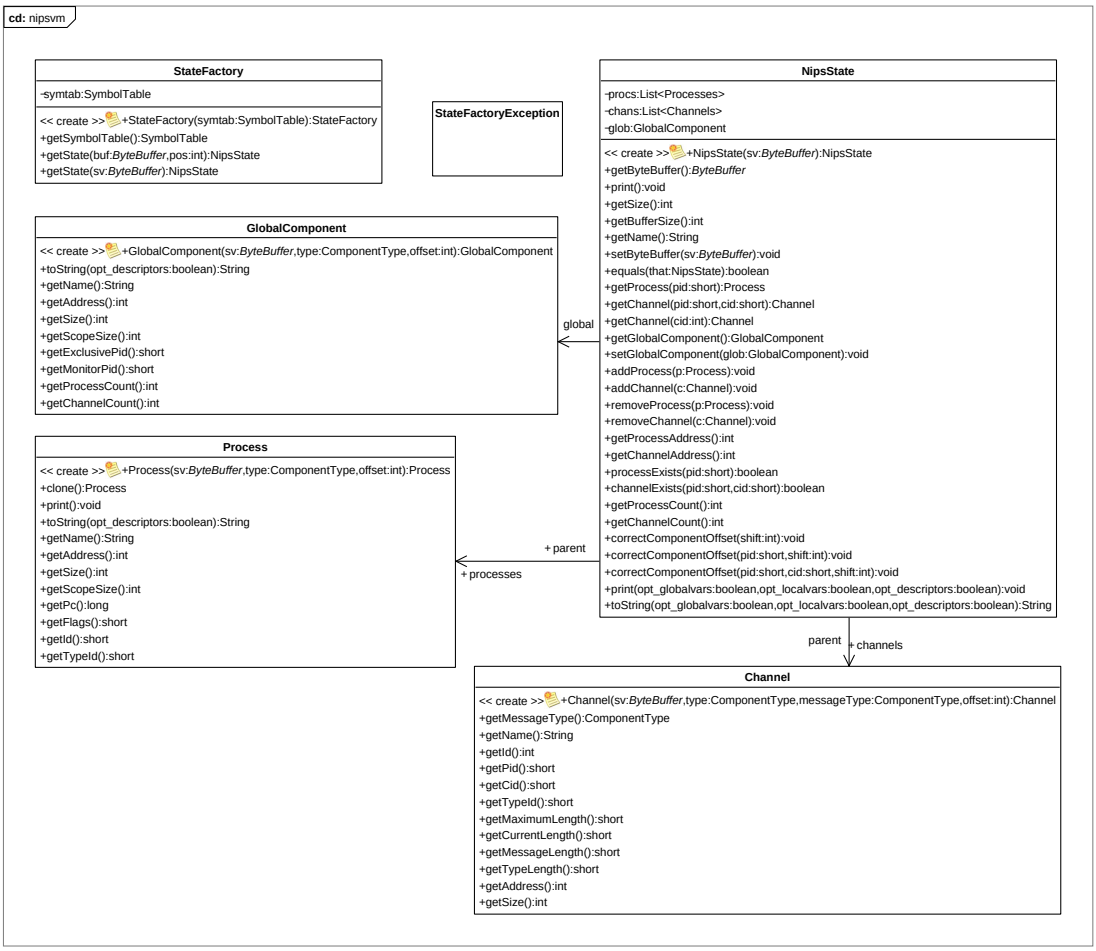



Figure 5.4: NIPS VM State API Class Diagram

## Transition API

Although the NIPS VM is used to evaluate behaviour of models, it does not offer API functions that assist in the discovery of what happened between state snapshots, it is a history-less byte code interpreter. NIPS byte code describes a unidirectional computational semantics of behaviour models. One could say that it describes the forward transitions between states. The NIPS VM does not log the control flow with the state vector successors it computes. To remedy the lack of a representation for transitions we use the SDI Transition API design, described in Section 4.5, to add support for bidirectional instructions for logging, reversing and replaying the behaviour. The approach is less precise than a full log of the control flow of each process, but does not require the design of the NIPS VM to be altered substantially.

The transition API operates using transitions described by sequences of instructions specific to NIPS VM run-time components. Transitions can help a debugger to distinguish between different types of events for NIPS components. Furthermore, transitions can be stored in a compact byte format, which may be more compact than state vectors. However, the preciseness of the description comes at the cost of loss of generality[2].

The *transition factory* generates NIPS VM component transitions for two state inputs and the *transition machine* (in this case called debugging VM) applies transitions to states. Table 5.1 shows the instructions for NIPS component transitions. Table 5.2 shows the instruction arguments and their meaning.

| Instruction | Arguments | Semantics | Inverse |
|---|---|---|---|
| `glob_new` | | Creates a new global component | `glob_kill` |
| `glob_kill` | | Disposes of the global component | `glob_new` |
| `glob_sz` | *size* | Resizes the global scope | `glob_sz` |
| `glob_diff` | *offset val_sz val* | Changes global variable values | `glob_diff` |
| `glob_excl_pid` | *pid* | Sets the exclusive executing process | `glob_excl_pid` |
| `glob_mon_pid` | *pid* | Sets the monitor process | `glob_mon_pid` |
| `proc_new` | *pid ptype_id size pc* | Creates a new process | `proc_kill` |
| `proc_kill` | *pid* | Disposes of a process | `proc_new` |
| `proc_sz` | *pid size* | Resizes a local scope | `proc_sz` |
| `proc_diff` | *pid offset val_sz val* | Changes local variable values | `proc_diff` |
| `proc_pc` | *pid pc* | Sets the program counter | `proc_pc` |
| `proc_flags` | *pid flags* | Sets the execution flags | `proc_flags` |
| `chan_new` | *cid ctype_id* | Creates a new channel | `chan_kill` |
| `chan_kill` | *cid* | Disposes of a channel | `chan_new` |
| `chan_add` | *cid pos msg* | Adds a message to a channel | `chan_remove` |
| `chan_remove` | *cid pos* | Removes a message from a channel | `chan_add` |

Table 5.1: NIPS Transition Instructions

**Constraints.** A *global component*, of which exactly one must exist at all times during the execution of a model, is implicitly declared to be of the SDI *global_component* type. The `kill_proc` instruction may not dispose of a process while processes with a higher process identifier exist, in line with the NIPS semantics. The `chan_kill` instruction cannot occur if we reason only with forward transitions because channels exist for a model checking eternity. However, since channels can be created in a forward transition by any process component at any given time, we must also consider the inverse transition where the channel is destroyed.

Channels contain *messages* that consist of a number of bytes equal to the size of the *message type*

[2]The generic SDI design was derived from the design of the NIPS component transitions.

| Argument | Denotes |
|---|---|
| *size* | The size in bytes a scope will have (excluding the preceding descriptor). |
| *offset* | The offset from the start of a scope address in bytes. |
| *val_sz* | The size in bytes of the *val* argument. |
| *val* | A sequence of bytes that will replace byte values in a scope. |
| *pid* | The process identifier of the process the instruction works on. |
| *pc* | The value the program counter of a process component will have. |
| *ptype_id* | The component type of a new process. |
| *cid* | The channel identifier of the channel the instruction works on. |
| *ctype_id* | The component type of a new channel. |
| *pos* | The position of a message in the channel where $pos \in \{0..max\_length\}$. |
| *msg* | A message that will be written into a channel. |

Table 5.2: NIPS Transition Instruction Arguments

of the channel that can be retrieved from the component type via the debugging API. Though we would expect messages to be sent from a *sender process* through a channel to a *receiver process*, it is not possible to observe who sent what message to whom by means of the information in the state vector[3]. Therefore, the component transitions only contain instructions that can *add* messages to, and *remove* messages from *positions* within the channel. It is also not always possible to determine which message was removed because the messages *shift* towards the start address of the channel when a lower positioned message is removed. It is thus indistinguishable which one of two identical adjacent messages is removed.

**Implementation.** Figure 5.5 shows the class diagram of the transition API. The transition API classes are stored in the *transition* package. The `TransitionFactory` is used to generate bidirectional transitions from two state snapshot inputs. `BiTransition` objects store two `ByteBuffer` objects which represent the forward transition and the backward transition in a compact byte code. The `DebuggingVM` executes the NIPS component transitions and applies the instructions stored in transition `ByteBuffers` objects to `NipsState` objects.

## Traces

Model checkers use search algorithms to perform state space explorations, as is described in Section 1.1. A counter-example may be returned by such an algorithm which is a *trace* or sequence of state vectors. Alternatively, traces consist of transition vectors. The NIPS VM debugging API provides support for reading and writing counter-examples to files on the hard disk and loading them to an application that can make use of them, such as a debugger.

**Implementation.** Figure 5.6 shows the class diagram of the *trace* package. The `TraceRepository` class is used to store `Trace` objects which represent NIPS VM counter-examples. `Trace` objects contain high level states represented by `NipsState` objects and transitions in the form of `BiTransition` objects. A `Trace` object can be saved to the hard-disk as a sequence of state-vectors. When read back from the hard-disk, the state vector trace is transformed into a `Trace` object again.

[3]Distinguishing sender and receiver processes would be possible with a fully logged control flow.

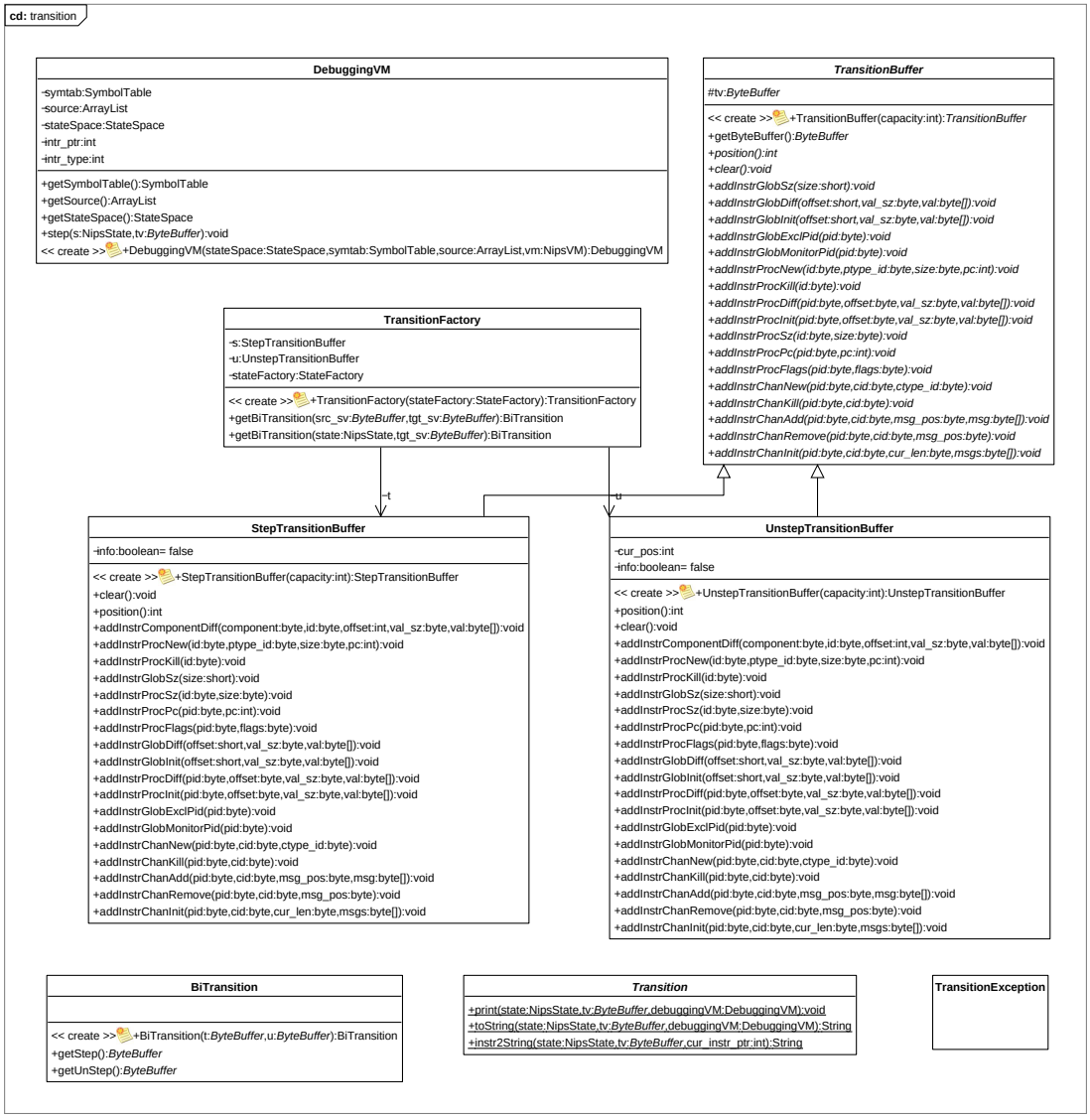



Figure 5.5: Nips VM Transition API Class Diagram


Figure 5.6: Nips VM Trace Class Diagram

### 5.1.3 The Nips Debugger

The Nips debugger is a command-line tool that makes it possible for users of tools based on the Nips VM to experiment with the behaviour of their models and evaluate the Nips VM results at source-level. A readily available debugger makes it more attractive for tool programmers to embed the VM as the back-end engine in their explicit state model checker tool.

The source-level functionality of the Nips Debugger is based on the debugging API supported by the SDI Framework. Depicting and modifying state variables is achieved by using the state API. It provides functions to view and manipulate Nips VM states in terms of named and typed global variables, processes and channels. The interactive transition choices, which offer a way to log and undo steps in the behaviour, are supported by the transition API. The component transition instructions provide a means to observe events and display them. The program counter, the `proc_pc` instruction and the target to source mapping make it possible to deduce the source location of a process at the beginning and at the end of a transition, but not the control flow in between.

States, as well as sequences of states (or counter-examples) can be passed to the debugger and it can depict them state by state, transition by transition. The debugger can perform random, interactive and guided simulations. In an interactive simulation the debugger allows the user to walk through the state space as a labelled transition system (LTS), by letting the user choose transitions. The debugger keeps a log of transitions and their inverse. Stepping back implies executing the inverse transition. In a guided simulation a trace of states is followed step by step, provided that the model can simulate the behaviour. A user may digress from the guided behaviour, switching to interactive simulation mode. A user may also switch from interactive to random mode, letting the debugger make the non-deterministic choices. Appendix D is a user manual for the Nips Debugger. It explains how to use the simulator in guided, interactive, automated and hybrid modes.

**Implementation.** The Nips Debugger is implemented in the *debugger* package depicted in Figure 5.7. The `Debugger`, which can be run from the command-line, can start `Search` threads. A class that extends `Search` is a scheduler algorithm which uses the JNI mapping of the API to schedule states. It determines when to stop a state-space exploration and when to return results. The `Simulator` class contains the simulator implementation. The `IterativeDFS`, `TransIterativeDFS`, `RecursiveDFS`, `NestedDFS` [44] and `CouvreurSearch` classes implement various experimental state-space exploration algorithms (some of which were already implemented) and algorithms to perform *sanity checks* for transitions. Sanity checks include checking whether a transition really unapplies its inverse.

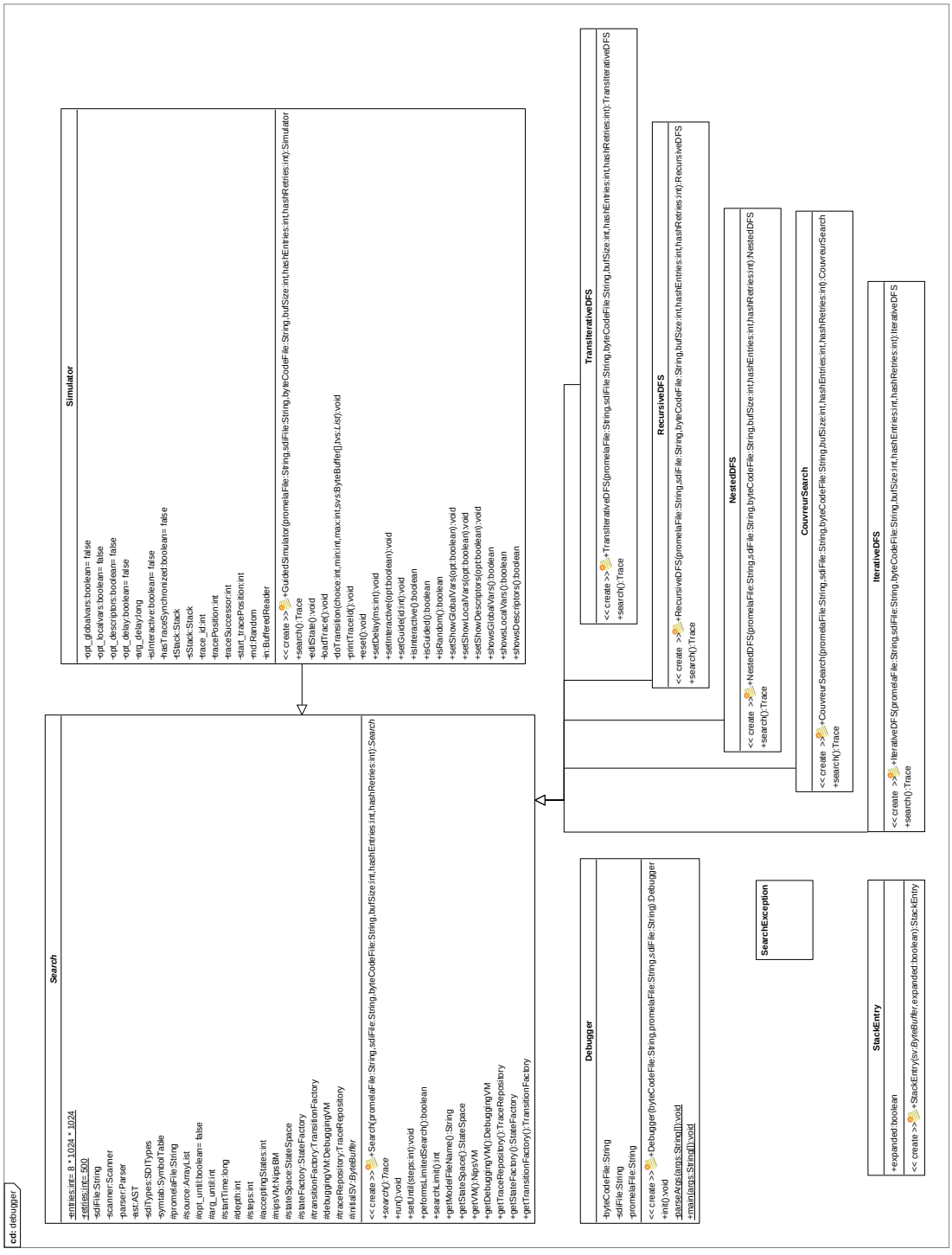



Figure 5.7: Nips VM Debugger Class Diagram

## 5.2 Evaluation

In this section an evaluation is made of the Nips Debugger and the SDI Framework it is based on. We discuss the offered functionality for debugging Promela models by means of simulation in Section 5.2.1 and the amount of effort needed to implement the Nips Debugger Section 5.2.2.

### 5.2.1 Debugging Functionality

The Nips Debugger is compared with the Spin command-line tool and the Nips VM built-in tool in order to evaluate the Nips Debugger and the SDI Framework it is based on. The built-in simulator of the Nips VM is included in the comparison to show the improvement that could be achieved with the Nips Debugger. The Spin command-line debugger is included because it is the default environment for debugging Promela models.

| | Nips Debugger | command-line Spin | Nips VM |
|---|---|---|---|
| random simulation | ✓ | ✓ | ✓ |
| display source-level states | ✓ | ✓ | |
| display descriptors | ✓ | | |
| interactive simulation | ✓ | ✓ | ✓ |
| interactive choice | transition | statement | successor state |
| undo, step back | ✓ | | |
| interactive state editing | ✓ | | |
| save current trace | ✓ | | |
| switch between modes | ✓ | | |
| guided simulation | ✓ | ✓ | |
| interactive guided simulation | ✓ | | |
| limited depth simulation | ✓ | ✓ | |
| skip first N steps | | ✓ | |
| make message sequence chart (ps) | | ✓ | |
| make state graph (graphviz) | | | ✓ |

Table 5.3: Promela Debugger Functionality

Table 5.3 shows the simulation functionality supported by the three tools. Each tool supports *random simulation* modes where the step-by-step behaviour of the model is randomly selected. An important improvement of the Nips Debugger over the Nips VM built-in tool is that it displays source-level states, in a way similar to Spin. Each of the tools support *interactive simulation* modes where the user can make choices regarding the non-deterministic control flow in the behaviour, however the interactive choices are different. A choice in the Nips Debugger is a transition, the differences between state snapshots, a choice in Spin is a source statement and a choice in the Nips VM is a binary successor state.

Added functionality of the Nips Debugger not yet supported by command-line Spin are *stepping back* in the behaviour to a previous state and *editing* state values. A step back entails performing the inverse transition of the transition that was just performed. Editing state values implies introducing a transition, which may or may not coincide with the model behaviour. A description of how to use the Nips Debugger is given in Appendix D.

We can conclude that the design of the SDI Framework provides a means to support a debugger for Nips that meets the functional requirements. The Nips debugger serves as a proof of concept for the SDI Framework and offers functionality that is comparable to that of the Spin command-line tool.

To make a fair assessment we must note that graphical shells are available for Spin such as `Xspin` that give additional functionality and provide an ease of use not commonly achieved with command-line tools. Therefore it is suggested that future work includes a graphical Nips debugger that profits from the SDI Framework. The Nips debugger represents a first step towards a reusable graphical debugger based on the SDI language.

### 5.2.2 Implementation Effort

This section gives an indication of the amount of effort it requires to use the SDI Framework in existing tool designs by enumerating the design and implementation requirements.

The goal of the research is to give the designers of explicit state model checkers a means to provide a source-level debugger for their tool with a modest effort. The idea is that the design and implementation efforts are primarily condensed in the SDI Framework. The framework provides the API functions for debugging support for modeling languages used with these tools. The effort required on the part of the tool engineer is to extend a compiler to generate SDI for its symbol table. The extensive documentation of the SDI Language and the SDI Framework in Chapter 3 and Chapter 4, the simplicity and readability of the SDI Language, and the SDI compiler extensions described in Section 4.6, can be used to facilitate that implementation.

An enumeration is given of the design and implementation efforts for the Nips Debugger.

1. The extensions to Nips VM required little implementation effort and consist of only a few lines of code. The debugging API is an addition to the VM and does not alter its design.
2. A greater amount of effort was required to extend the Nips Promela compiler to generate SDI for its symbol table, but the difficulty was primarily understanding the compiler and its symbol table design.
3. The Java implementation of the debugging API for the Nips VM predates the design of SDI Framework. Therefore it cost an additional amount of effort to implement.
4. The Nips Debugger cost a moderate amount of effort to implement, a few days of programming for one person with domain knowledge at most.

Without the use of software metrics, it cannot be stated with absolute certainty that it is easy to provide a source-level debugger for explicit state model checkers using the SDI Framework. We believe however that the amount of effort needed to gain access to its functionality is much less than to design and implement a completely new debugger. When a fully completed SDI Framework becomes available, which includes a generic SDI Debugger, these efforts can be further reduced.

## 5.3 Concluding Remarks

In this chapter we presented the Nips command-line debugger for use with tools based on the Nips VM. Although the design of the debugger itself is simple, it extends the Nips VM tool set with debugging functionality that its users require. The Nips VM, the Nips Promela Compiler and the Nips Debugger together provide a simulator comparable to that of the Spin command-line tool.

The SDI output required from the Nips Promela Compiler is supported by the SDI compiler extensions explained in Section 4.6. The simplicity of the language and its readability can help in quickly extending Nips VM based tools with debugging functionality.

Although currently the design of the Debugging API is tied to the current version of the Nips VM, the SDI Framework is flexible and can be used with future versions of the Nips VM. Because of the extensible design of the SDI Framework, it can be used in future graphical versions of the Nips Debugger.

The Nips Debugger provides a proof of concept for the Nips Debugging API and the SDI Framework it is based on. It shows that the Nips Debugging API provides the functionality needed to support a source-level debugger. More generally, the case study shows that the SDI Framework provides a means to create a source-level debugger for explicit state model checkers using state vectors.

# Chapter 6

# Conclusion

This chapter presents the general conclusions of this thesis. A summary of the main contributions is presented as well as possible research directions for future work.

## 6.1 Contributions

In this thesis we have presented an approach for extending the application field of the Nips VM by adding source level debugging functionality to it.

In order to achieve this we have developed a language with a simple multi-use readable format for Static Debugging Information (SDI) and the SDI Framework for state manipulation, which is based on this language. The framework can generally be applied to debugging designs for explicit state model checkers using state vectors. It is used in particular for a debugger for the Nips VM which is presented in this thesis. The main contributions are the following:

- **A modeling language for debugging information.** The SDI Framework makes use of the SDI Language, which is introduced in Chapter 3. SDI is a meta-language for static debugging information that describes memory models of running programs in terms of variables, types and source locations. SDI deals with the following difficulties: **i)** It can used to model the memory of modeling languages used with explicit state model checkers in order to relate otherwise unreadable binary state information back to its program source. It is particularly useful for model checkers that use a separate compiler to generate a low level intermediate representation that lacks source level information; **ii)** SDI is a high level list language that consists of entries and attributes which cover common modeling language types and structures; **iii)** It is extensible to support additional entries and attributes such that more language features can be described as meta-level objects; **iv)** Unlike most debugging information formats the SDI language is not tied to an object file format. A low level implementation is left as an implementation choice; **v)** In contrast to stabs and the Java class file format SDI is block structured like Dwarf, as is common in programming languages. However SDI supports less features than Dwarf which is meant for use with procedural programming languages; **vi)** SDI can easily be learned by means of the graphical modeling notation introduced in Section 3.2.1.

- **A framework for state manipulation.** The SDI framework is a state manipulation framework that offers support for source level debugging functionality by constructing memory models of running programs for explicit state model checkers. The SDI Framework is described in Chapter 4. The SDI framework solves the following problems regarding debugging functionality and state introspection. It provides debugging API, a means for programs e.g. debuggers, to perform meta-level operations at run-time. Such programs can: **i)** view and edit binary state vectors in terms of variables names and types via a reflective state API; **ii)** relate target programs and source programs to each other by means of a source location mapping; **iii)** Log, replay, create and undo transitions that manipulate states via a transition API. Additionally, it is made easier for compiler engineers to extend the compiler design with support for SDI code generation by means of the SDI compiler extensions.

- **A debugging API for the Nips VM** In order to support the Nips Debugger we add a debugging API to the Nips VM based on the SDI Framework in Chapter 5.

- **A debugger for the Nips VM**. We extended the Nips VM Tool Set with a debugger that allows users to simulate the behaviour of Nips byte code. The debugger graphically depict states and transitions at source-level. In an interactive simulation, which may be guided by a counter-example, users may choose transitions and edit state values. The Nips Debugger and the SDI Framework are evaluated in a case study of the debugger in comparison with Spin in Chapter 5.

In addition, this thesis provides secondary contributions.

- **Embeddable Nips VM.** The API of the Nips VM has been documented in Section A.2. It was clarified how the Nips VM, as a virtual machine for state-space generation, can be embedded as an explicit state model checker engine by means of its API. We have documented what the API functions are, when they must be called, what arguments they have and what actions they perform. This report does not answer questions regarding whether the Nips VM offers all the services needed for host-applications to embed it, and whether the API should be extended. However, some of the difficulties regarding these open issues are discussed in Appendix A, and preliminary ideas of how to enhance the Nips VM are proposed.

- **Debugging Information Education.** The design can be useful for education because of the ease of use of the SDI compiler extensions, the simplicity of the SDI notation and the possibility of sparking the imagination of students with the SDI modeling notation described in Section 3.2.1.

With the contributions of this thesis the Nips VM can more easily be used. It is now possible to embed the Nips VM and provide source level debugging support for Nips VM based tools with the Nips command-line debugger. The SDI framework is easy to reuse in existing explicit state model checking frameworks because of its simple language and graphical notation, the documentation of the SDI Framework and its debugging API and the existence of compiler extensions. We believe that the SDI Framework provides debugging support for the Nips VM that is sufficiently flexible and extensible to be reused with future implementations.

## 6.2 Future Work

The following list presents topics which are indicated for further investigation. The list is categorized to the amount of effort that we believe it costs to research.

Major future work includes:

- A Nips VM based model checker can be created that uses the Nips debugging API. The API may be made accessible to models with byte code instructions that support reflection. An interesting question is whether such a model checker can employ debugging information to facilitate techniques that reduce the state space. Particularly if it can be applied to Dynamic Partial Order Reduction and guided model checking [36, 38] since the debugging API gives access to types and values of variables by name at run-time.
- The current version of the Nips Debugger is a command-line simulator. Its user manual is described in Appendix D. It is future work to implement the design described in Section 4.7 of a generic graphical SDI debugger that is programmed with Eclipse [11, 12].

Other future work includes:

- The SDI language is currently small and describes debugging information best for modeling languages such as Promela. It could be interesting to extend SDI with functions in order to support functions and procedures of programming languages to extend its application field, e.g to support more programming languages.
- The SDI language is designed in such a way that it can naturally be combined with state collapsing, a technique for state space optimization [19]. It could be researched to what extent can the SDI Framework can be responsible for supporting and storing the state space.
- Various enhancements for the Nips VM can be conceptualized. Some of these are briefly mentioned Appendix A. Among them are language features supported by additional instructions. One such a feature could be dynamic heaps to support object oriented languages. This will require algorithms for heap canonicalization and heap compression that is similar to that in JPF [48] as is mentioned in Section A.3.4.
- The Nips byte code can be optimized using static code analysis [2]. This alters the byte code in terms of the number of lines and the memory utilization at run-time. It breaks the contract of the compiler with the SDI Framework and the debugger. It can be interesting to research debugging optimized Nips byte code as is mentioned in Section A.3.5.

# Bibliography


[1] D. Anderson. DWARF3: better than DWARF2. http://reality.sgiweb.org/davea/, December 2005.

[2] Gustavo Quiros Araya. Static Byte-Code Analysis for State Space Reduction. Master's thesis, Faculty of Mathematics, Computer Science and Natural Sciences of RWTH University, Aachen, Germany, March 2006.

[3] J. Barnat, L. Brim, I. Cerna, and P. Simecek. DiVinE the distributed verification environment. In M. Leucker and J. van de Pol, editors, *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC05)*, July 2005.

[4] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in context: the shape of the design space. *Object-oriented programming: the CLOS perspective*, pages 29–61, 1993.

[5] Gilad Bracha and David Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. *SIGPLAN Not.*, 39(10):331–344, 2004.

[6] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusets, 2000.

[7] Edmund M. Clarke and Jeannette M. Wing. Formal methods: state of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[8] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building Your Own Software Model Checker Using The Bogor Extensible Model Checking Framework. Technical Report SAnToS-TR2005-1, Department of Computing and Information Science, Kansas State University, January 2005.

[9] Matthew B. Dwyer, John Hatcliff Robby, and Matthew Hoosier. Supporting Model Checking Education using BOGOR/Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 88–92, New York, NY, USA, 2004. ACM Press. http://model-checking.courses.projects.cis.ksu.edu/.

[10] Michael J. Eager. Introduction to the DWARF Debugging Format. http://dwarfstd.org/, February 2007.

[11] Eclipse.org. Eclipse Platform. Available at http://www.eclipse.org/.

[12] Eclipse.org. Graphical Editing Framework. Available at http://www.eclipse.org/gef/.

[13] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed Explicit-State Model Checking in the Validation of Communication Protocols. *Int. J. Softw. Tools Technol. Transf.*, 5(2):247–267, 2004.

[14] Éric Bruneton and Romain Lenglet and Thierry Coupaye. ASM: a Code Manipulation Tool to Implement Adaptable Systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, November 2002.

[15] FSG. *DWARF Debugging Information Format, Version 3*. Free Standards Group, December 2005.

[16] Etienne M. Gagnon and Laurie J. Hendren. SableCC, an Object-Oriented Compiler Framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140, Washington, DC, USA, 1998. IEEE Computer Society. http://sablecc.org/.

[17] Moritz Hammer and Michael Weber. "To Store Or Not To Store Reloaded": Reclaiming Memory On Demand. In Lubos Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Application and Technology (FMICS'2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer-Verlag, 2006.

[18] John Hatcliff, Matthew B. Dwyer, and Robby. Bogor: An Extensible Framework for Domain-Specific Model Checking. Technical Report SAnToS-TR2004-9, Department of Computing and Information Science, Kansas State University, December 2004. To appear in the Newsletter of European Association of Software Science and Technology (EASST).

[19] Gerard J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proc. of the 3th International SPIN Workshop*, 1997.

[20] Gerard J. Holzmann. *The SPIN Model Checker Primer and Reference Manual*. Addison-Wesley, 2003.

[21] Gerard J. Holzmann and Anuj Puri. A Minimized Automaton Representation of Reachable States. *Draft version of paper that appeared in STTT*, Vol. 2, No. 3:270–278, 1999.

[22] Steven C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[23] Joost-Pieter Katoen. *Principles of Model Checking (Draft)*. unpublished, 2004.

[24] M. E. Lesk and E. Schmidt. Lex A Lexical Analyzer Generator. pages 375–387, 1990.

[25] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification, 2nd edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[26] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[27] Lee E. McMahon. *Sed A Non-Interactive Text Editor*, pages 389–397. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[28] Julia Menapace, Jim Kingdon, and David MacKenzie. The stabs debug format. Technical report, Free Software Foundation, Inc., 2001.

[29] Sun Microsystems. *Stabs Interface Sun Studio 9*. Sun Microsystems, Sun Microsystems, Inc. 4150 Network Circle Santa Clara, CA 95054 U.S.A., 2004.

[30] Terence Parr. ANother Tool for Language Recognition. http://www.antlr.org/.

[31] Dale Parson, David J. Murray, and Yu Chen. Object-Oriented Design Patterns for Debugging Heterogeneous Languages and Virtual Machines. *Softw. Pract. Exper.*, 35(3):255–279, 2005.

[32] PLSIG. *DWARF Debugging Information Format*. Programming Languages Special Interest Group, Unix International, Parsippany NJ., 1.1.0 edition, October 1992.

[33] PLSIG. *DWARF Debugging Information Format, Version 2.0*. Programming Languages Special Interest Group, Unix International, July 1993.

[34] Roger S. Pressman. *Software Engineering: A Practitioner's Approach (European Adaptation)*. McGraw-Hill Higher Education, 2000.

[35] Norman Ramsey and David R. Hanson. A Retargetable Debugger. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27 7, pages 22–31, New York, NY, 1992. Princeton University, ACM Press.

[36] Venkatesh Prasad Ranganath, John Hatcliff, and Robby. Enabling Efficient Partial Order Reductions for Model Checking Object-Oriented Programs Using Static Calculation of Program Dependences. Technical Report SAnToS-TR2007-2, Department of Computing and Information Science, Kansas State University, March 2007.

[37] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A Flexible Framework for Creating Software Model Checkers. In *In the Proceedings of Testing: Academic & Industrial Conference - Practice And Research Techniques. Technical Report, SAnToS-TR2006-2.*, 2006.

[38] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-Reduction Strategies for Model Checking Dynamic Systems. In *In the Proceedings of 2003 Workshop on Software Model Checking (SoftMC 2003). Technical Report, SAnToS-TR2003-5*, May 2003.

[39] Sukyoung Ryu and Norman Ramsey. Source-level Debugging for Multiple Languages with Modest Programming Effort. In *14th International Conference on Compiler Construction*, pages 10–26. Division of Engineering and Applied Sciences Harvard Univsersity, April 2005.

[40] Bastian Schlich, Michael Rohrbach, Michael Weber, and Stefan Kowalewski. Model Checking Software for Microcontrollers. Technical Report AIB-2006-11, University of Aachen, 2006.

[41] Stefan Schürmans. Promela - Operationelle Semantik. http://homepages.cwi.nl/∼weber/nips/PromelaOpSem.pdf, August 2005.

[42] Stefan Schürmans. Promela - Statische Semantik. http://homepages.cwi.nl/∼weber/ nips/PromelaStatSem.pdf, June 2005.

[43] Stefan Schürmans. Promela - Virtuelle Maschine. http://homepages.cwi.nl/∼weber/ nips/PromelaVM.pdf, November 2005.

[44] Stefan Schwoon and Javier Esparza. A Note on On-The-Fly Verification Algorithms. Technical Report 2004/06, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, November 2004.

[45] Andrew P. Tolmach and Andrew W. Appel. Debugging Standard ML Without Reverse Engineering. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 1–12, New York, NY, USA, 1990. ACM Press.

[46] Ronald Veldema, Michael Klemm, and Michael Philippsen. Personal communication on the tapir programming language. http://www2.informatik.uni-erlangen.de/Forschung/Projekte/Tapir/?language=en, 2006.

[47] TOC View. IEEE Standard for Microprocessor Universal Format for Objectmodules. *IEEE Std 695-1990*, 1990.

[48] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[49] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.

[50] Michael Weber. An Embeddable Virtual Machine for State Space Generation. In D. Bosnacki and S. Edelkamp, editors, *Proceedings of the 14th International Workshop on Model Checking Software SPIN'07*, volume LNCS4595, pages 168–185. Springer, July 2007.

[51] Michael Weber and Stefan Schürmans. A Virtual Machine for State Space Generation. *SENVA 2005 Workshop*, 2005.

[52] Phil Winterbottom. Acid: A Debugger Built From a Language. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, 2000.

# Appendix A

# Enhancing the Nips VM

The Nips VM was introduced in Section 2.1. This appendix serves to clarify how the Nips VM can be embedded in host-tools as an explicit state model checker engine and how the tool set can be enhanced with new components. The most important Nips VM API functions are briefly explained in Section A.2. Design suggestions which stem from experience with the Nips VM gained during the thesis are discussed in Section A.3.

## A.1 Introduction

A Nips VM based tool, depicted in Figure A.1, consists of a *tool compiler* that encodes the language semantics as Nips byte code, the Nips VM interpreter and state vector generator, and a *scheduler algorithm* that manages the *state space* and the search.

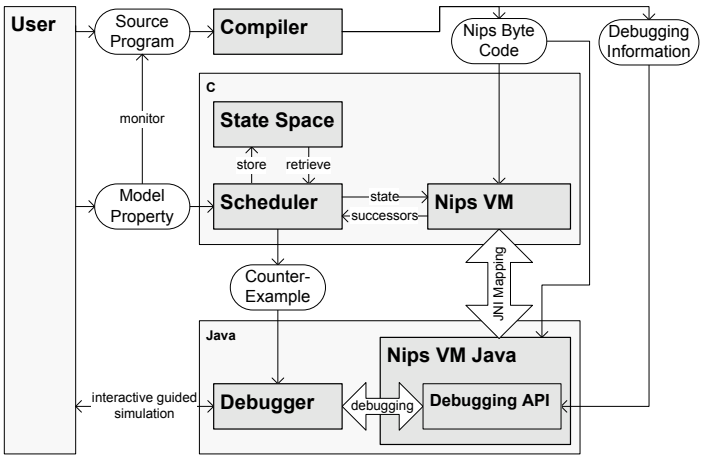



Figure A.1: Nips VM Tool Schema

In order to use the Nips VM with a new tool, some tool components may have to be built while others are readily available such as the Nips Promela Compiler, DFS and BFS scheduler algorithms in *search.c* and a Hash-Table to store states in *hashtab.c*.

Developers that use the Nips VM as the explicit state model checker engine in their tool may wish to design a new modeling language specific to a particular domain. For a new modeling language a new compiler must be defined. The semantics of the language must be defined using Nips byte code in order to be able to use the Nips VM. Byte code instructions encode the features of the language, which can be statements and control flow constructs such as `if` or `while` statements, *variables* and *types*. The run-time components must be represented by a *global component* plus sequence of *process* and *channel* components. The tool compiler can extended to provide an *optional SDI specification* of the run-time memory model that provides a means for debugging as explained in Section 4.6.

## A.2 The Nips VM API

The Nips VM is a byte code interpreter that generates state vectors given a state vector input, a schema of its API is depicted in Figure A.2. The Nips VM does not log instructions while it executed. The control flow of what has happened is intentionally hidden.

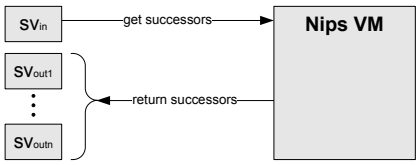



Figure A.2: Nips VM API

The Nips VM is a modular design which is consists of the Nips VM byte code *interpreter back-end* and *scheduler* algorithms which control the state space exploration.

### A.2.1 Initialization

To initialize the Nips VM the following steps have to be performed.

1. Initialize the Nips VM module.
   ```
   nipsvm_module_init();
   ```
2. Load the Nips byte code from a file.
   ```
   st_bytecode *bytecode;
   char *bytecode_file = ...
   char *bytecode_module = ...
   bytecode = bytecode_load_from_file( bytecode_file, bytecode_module );
   ```
3. Initialize the run-time environment.
   ```
   nipsvm_t nipsvm;
   ```

```
nipsvm_scheduler_callback_t search_succ_cb = ...
nipsvm_error_callback_t search_err_cb = ...
nipsvm_init (&nipsvm, bytecode, &search_succ_cb, &search_err_cb); //return 0 = ok
```

The first three steps are straightforward. In step four the *nipsvm_init* function is called to set the run-time environment context of the VM. The first argument is the address of NIPS VM structure. The second argument is the loaded byte code. The third argument is the *scheduler callback function* which returns the successors and thread of control to the scheduler after the successors are computed. The fourth argument serves to replace the default error handling which translates the error code and prints the error message on the screen, the error context consists of process run-time information (process id and program counter).

### A.2.2 Scheduler

Tools that embed the NIPS VM as a tool back-end must use a *scheduler* algorithm that iteratively determines which state is expanded in the search and is responsible for the state space. The NIPS VM source contains a built-in scheduler algorithm in *search.c*. The results of the NIPS VM are stored in a hash-table. Schedulers must provide their own search context such as the *maximum search depth*, the *hash-table*, the *state buffer* in which the results are returned, but it must also keep track of run-time status information. A typical scheduler is set up by performing the following steps.

1. Allocate a buffer for states.
   ```
   unsigned long buffer_len = ...
   p_buffer = (char *)malloc( buffer_len );
   ```
2. Allocate a Hash-Table.
   ```
   unsigned long hash_entries = ...
   unsigned long hash_retries = ...
   t_hashtab hashtab = hashtab_new( hash_entries, hash_retries ); //built-in hash-table
   ```
3. Get the initial state.
   ```
   nipsvm_state_t *state;
   state = nipsvm_initial_state();
   ```
4. Schedule states: iteratively choose states for which to get successors.
   ```
   ctx = ... //private search context
   nipsvm_scheduler_iter (nipsvm, state, ctx);
   ```
5. Scheduler termination condition.
   ```
   nipsvm_state_monitor_accepting (state) //1 = true, 0 = false
   nipsvm_state_monitor_terminated (state) //1 = true, 0 = false
   nipsvm_state_monitor_acc_or_term (state) //1 = true, 0 = false
   ```

Schedulers use the *monitor process* to check the system state which can be *excepting* or *terminated* and determine if a counter-example has been found Counter-examples must be represented by the scheduler as a sequence of states which can be analysed using the NIPS Debugger which is described in Section 5.1.3.

For the deinitialisation following steps must be performed.

1. Free the State Buffer and any other memory allocated for the scheduler.
2. Unload the byte code.
   `bytecode_unload( bytecode );`
3. Finalize the Nips VM.
   `nipsvm_finalize (&nips_vm);`

## A.3 Design Suggestions

Unfortunately the scarcely documented software design of the Nips VM makes it difficult to add API functionality without a redesign of the VM. The research in this thesis has been hampered by this. The goal is to use the experience to formulate constructive suggestions for the next design iteration of the Nips VM.

### A.3.1 Depth First Search

The Nips VM API returns the state vector successors given an input state vector. For Breadth First Search (BFS) this is desirable because each of the successors will be used in the next evaluation step. Figure A.3 shows a BFS search schema where the black dots are states and arrows are transitions.

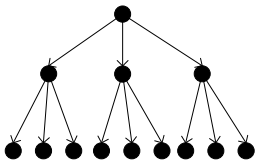


Figure A.3: Breadth First Search

For Depth First Search (DFS) state space explorations this poses a difficulty. Only one successor is needed at a time but all successors are constructed by the VM and returned at the same time. To not waste the effort the unevaluated state vectors are cached in the state space and stored on the DFS stack once they are expanded in the search they are marked with a byte value to denote this. Figure A.4 shows the DFS schema for the Nips VM.

It is a waste of time and space for the computation of these potentially unnecessary states (depicted gray). We can conclude that the Nips VM is biased towards performing a BFS. In contrast, Spin is biased towards DFS and it provides the BFS only as a user friendly option which is only effective for safety properties [20].

A solution to the Nips BFS-bias is not easily found. Theo Ruys has coined the idea of a DFS state iterator which is an extension to the Nips API that only calculates and returns one successor at a time.

The difficulty in adding such an iterator to the Nips VM is that the state of the environment of the VM is scattered throughout the functions that compute the successors. The successors

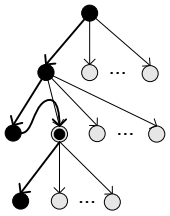


Figure A.4: NIPS VM Depth First Search

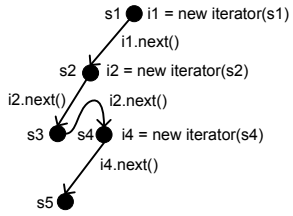



Figure A.5: NIPS VM Depth First Search Iterator

evaluation-state, the variables and values of functions that compute the successors, must be stored within the state iterator which itself is placed on the DFS stack. It requires a redesign of the NIPS VM in order to make it possible to store the successor evaluation-state in state iterators. Even if this design is implemented it may be that iterators are too large to store on the DFS stack to efficiently perform a DFS.

### A.3.2 State Space Organisation

Using the JNI mapping of the NIPS VM API to calculate successors has the side-effect of storing states in the state space. Tool engineers that want to create their own state space can only do so at the cost of a duplicate state space. The NIPS API JNI mapping needs to be adapted in order to give easy support for alternative state space organisations and to turn off the built-in state cashing routine.

**State Compression.** Only one hash-table is provided with the NIPS VM which is not optimized in any way. In [19] a static compression technique called *byte masking* is suggested, where constant fields in the state-vector are marked by a byte mask. Before the state vector is stored only the unmarked bytes are copies, the others are ignored. An example of constant information in NIPS VM channels are the type constants in channel descriptors the *maximum length*, *message length*, the *type length* and and the *type bits*, these can be marked and they can be stored in and retrieved from a type table built up from SDI. This means that invariant debugging information is no longer saved in the state space. The author of [19] also suggests to byte mask any fields that are not used, rendez-vous channel content, compiler-added byte alignment padding, the number of active processes and channels and process identifiers. The SDI language can be extended with

an extra *marked* attribute (discussed in Section 3.4) value, e.g. *masked* that denotes the byte mask of a sequence of bytes to convey this information.

**Collapse Compression.** The NIPS VM debugging API Chapter 5 regards the representation of a state as a hierarchy of components. This hierarchy can be compressed using collapse compression discussed in [19] where each local state-vector component, encoding the local state of a single process or channel, is stored separately in the hash-table, and assigns it a unique index-number. The index-numbers for all components together form the compressed global state-vector. The SDI states that represent a sub-state within a state component must also be indexed. This approach constitutes an iterative collapse compression such that a complete graphs of components can be stored in the state space as separate components and the compressed state is saved as a global state-vector. The global state-vector can be decompressed by looking up the component index and the debugging API allows reading from typed named variables stored in the component vectors. This simple form of collapse compression can be optimized using *recursive indexing* where the number of components and the number of bytes used per index in the global state vector as well [19].

### A.3.3 Error Handling

Error handling can pose a difficulty for both embedding the NIPS VM in a C tool as model checking engine and for API Error handling for Java programs using the API JNI mapping. The NIPS VM API allows tools to replace the default error handling callback function and the global *errno* variable can be queried for an error code. The JNI mapping does not return error values to the Java thread of control and therefore errors cannot be thrown or caught and proper Java error handling cannot be done. In future versions care must be taken to ensure that error values are retrieved with the return value of the called API function.

### A.3.4 Language Support

The claim is that NIPS byte code can be more widely applied than just to Promela semantics, i.e. cover more languages. However implementing a compiler with *functions* and *procedures* is complex because the NIPS byte code support is not offered with instructions for *activation frames* and only `call` and `return` instructions are offered. Be that as it may, a domain specific programming language for systems programming called TAPIR has been created that uses NIPS byte code for its semantics [46].

Adding more instructions for dynamic language features to the NIPS byte code to support imperative languages can extend its application possibilities but will also make the VM considerably more complex as the need arises for model checking techniques to reduce the state space. Possible additions include *objects* and *heaps* for dynamic object creation. To curb the state space explosion resulting from different heap orderings, *heap symmetry reductions* ([48, 38]) will then need to be implemented.

As the NIPS VM is a virtual machine for state space generation and it alone decides the visible, relevant states, a compiler-specific implementation for heap symmetry reductions seems to contradict the VM's design. Furthermore, the compiler-specific design may be difficult to reuse in other tools and violates the modularity concern. There is a balance between the generality of the byte code instructions and the difficulty to use these instructions to build complex reusable language features and explicit state model checker technologies.

An implementation of a modeling language feature that is supported by Nips byte code, makes it easier to provide flexible, reusable debugging support via the Nips VM debugging API. For features natively supported by the Nips VM, reflection can be offered via meta structures called mirrors [5]. Mirrors can be used for reflective purposes such as debugging and meta-programming applications such as model checking with state introspection.

### A.3.5 Code Optimization

In the Chapter 1 it was explained that static analyses can be used for state space reduction. The static code optimization used with Nips byte code neither preserves the mapping between target and source code nor does it update the memory model in case variables are removed. In future versions the byte code control flow graph transformation described in [2] should also transform the debugging information to provide a means to debug optimized code.

### A.3.6 Transitions

Part of the research of this thesis is to find out what transitions are within the Nips VM and how they can be represented. Transition information is especially useful to view events and these events can be used to study the model behaviour by means of a debugger. The design of the VM is a state centric history-less black-box interpreter. Like Spin the focus is on states rather than transitions or events, and this also extends to the way in which correctness properties are verified [20]. Transition choices are made by the VM, it does not allow being steered by the scheduler which cannot select statements and determine the control flow. Rudimentary transition information is returned with the state successors.

The *component transitions* presented in Section 4.5 are constructed from the source and the target state after the transition has been performed by the Nips VM. They express changes to components in between state snapshots without the knowledge of the control flow of the process program counters and allows users of Nips VM based tools to observe the system behaviour with the Nips debugger. As opposed to Nips byte code, Component transitions are bi-directional meaning what is done by a Nips byte code *step* can be undone by the inverse component transition.

The Nips API is state-centric in that the focus is on states and transitions are implicitly defined by the execution of the byte code. A design suggestion is to optionally *log the control flow* of the executed byte-code with the computed successor states for debugging purposes as is depicted in Figure A.6. It returns *transition vectors*, sequences of control flow program counters choice points associated with `ndet` instructions, with each state vector it returns. A transition vector is associated with each source statement.

**Partial Order Reduction.** If the Nips VM is to support Partial Order Reduction it must make an educated choice to execute a statement. Using static code analysis the control flow of the program an be constructed as a graph and an analysis can be made to determine *independence relation* between statements [6] which can be used to identify *partial orderings* on-the-fly that in turn can be used to prevent states from ever being stored in the state space while preserving coverage of the state space.

The Nips VM must be allowed to choose statements by itself, the API schema depicted in Figure A.7 is not feasible because they force a design with a tremendous overhead and optimizations

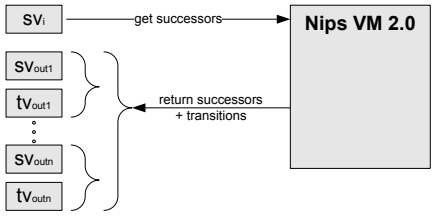



Figure A.6: Nips VM API + transition log

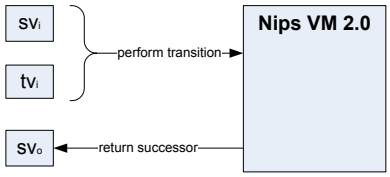



Figure A.7: infeasible Nips VM API concept: control flow transition

such as reusing partial results cannot be used.

Instead of implementing the POR by letting the scheduler choose statements it should be placed in the Nips VM itself. For a DFS scheduler POR can be combined with the notion of a state iterator introduced in Section A.3.1, to an ample-set state iterator.

**Reflective Layer.** The research into a debugging API for the Nips VM has been primarily focused on debugging with the Java JNI mapping of the Nips VM. However it has also been considered that debugging information can be used for model checking purposes using the debugging API as a reflective layer.

The Nips VM can make decisions at run-time about which statement to execute next based on the values of variables relevant to the verification property. The debugging API provides a means for directed model checking.

## A.4 Concluding Remarks

Some design suggestions can be implemented with the current version of the Nips VM such as alternative schedulers and state space organisations, state collapsing and channel space optimizations, and JNI error handling, but others require a more rigorous redesign of the Nips VM such as the state iterator, transition logging and activation frames to support additional imperative programming languages. Hopefully the experiences regarding the Nips VM can be put to good use.

# Appendix B

# Nips Compiler Extensions

The Nips Promela Compiler must be extended to generate SDI with the Nips byte code in order to use the SDI Framework to support source-level debugging. First SDI definitions which are constant to the Promela language are defined in Section B.1. Then an approach is given in Section B.2 used to describe the symbol table in a flattened SDI definition.

## B.1 Predefined SDI

The `Promela` modeling language has a number built-in types which include: `int`, `short`, `byte` and `bit`. The Nips VM uses big endian byte ordering and a two's complement memory notation for signed integers. Using this information we can now create the predefined *basetypes* which are used with every program generated by the Nips Promela Compiler.

```
1 (base_type
2     (name bool)
3     (size 1)
4     (format bool)
5     (bits 1)
6 )
```

(a) `bool`

```
1 (base_type
2     (name bit)
3     (size 1)
4     (format beuint)
5     (bits 1)
6 )
```

(b) `bit`

```
1 (base_type
2     (name byte)
3     (size 1)
4     (format besint)
5     (bits 8)
6 )
```

(c) `byte`

```
1 (base_type
2     (name short)
3     (size 2)
4     (format besint)
5     (bits -15)
6 )
```

(d) `short`

```
1 (base_type
2     (name int)
3     (size 4)
4     (format besint)
5     (bits -31)
6 )
```

(e) `int`

Figure B.1: Promela base type SDI definitions

Figure B.1 shows the *base type* SDI definitions for Promela on the Nips VM. The five basetypes are defined in terms of *name*, *size* and *format* as is required by the definition of SDI *base_type* entries requires in Section 3.5. The *bits* attribute is added to the basetypes which is used by

channels within the NIPS VM. Note that the `bool` type could also have been defined using an SDI *enumeration type* entry with two *enum_field* sub-entries describing the symbolic values for zero and one which are `false` and `true` respectively.

## B.2 Program defined SDI

Program defined SDI cannot be predefined since it differs per program. For different PROMELA statements different NIPS byte code is generated and because of this different source locations are computed. Programs may contain different variable, scope and type declarations.

To generate program defined SDI adjustments to the symbol table of the compiler have to be made. Each of the classes in the NIPS Compiler symbol table is extended with the `generateSDI` function that both returns a `String` and fills up a debugging information structure. The SDI generated by the `Nips` Compiler consists of, program scopes, type declarations, and variables and program source and target locations.

### B.2.1 Variables

PROMELA *variables* and *scope levels* are encoded into debugging information using the respective SDI entries. For each of the symbol table entries code templates show how debugging information is generated. For variable declarations stored in `SymTabVarBase` or `SymTabVarUType` class objects

*evaluate*(*SymTabVarBase e*) =
(variable
(name *e.name*)
(type *e.type*)
(offset *e.offset*)
)

Figure B.2: Variable SDI code template

variable entries are generated. Figure B.2 shows the variable SDI code template. The *name*, *offset* and *type* attributes values are determined by information contained in the symbol table entry. The type of the variable is also defined in the symbol table. It is safe to assume this because the contextual analysis phase should stop the compilation when type errors are encountered. For array declarations *array type* entries are generated.

Array variables are stored in `SymTabArrayBase` class objects. The *name*, *offset*, *length* and *type* attributes of the *array type* entry are determined by the information in the `SymTabArrayBase` entry. Figure B.3 shows the array SDI code template.

Scope levels are the *global scope* and processes *local scopes*. In the NIPS Compiler, processes also use *argument scopes*. For each scope level entry in the symbol table an SDI scope entry is generated. Figure B.4 shows how SDI code is generated for the symbol table `Scope` class.

There is only one `Scope` class in the `SymbolTable` which represents scope information for the global component as well as local and argument scopes. The code generation template shows the two cases. If the *id* of the scope is zero then it is the global scope and the function returns

```
evaluate(SymTabArrayBase e) =
        (variable
                (name e.name)
                (type e.type+label)
                (offset e.offset)
        )
        (array_type
                (name e.name)
                (type e.type)
                (length e.length)
                (offset e.offset)
        )
```

Figure B.3: Array SDI code template

```
evaluate(Scope e) =
    if(e.id = 0) return
        (
            (global_scope
                (offset 0)
                (size size(state_descriptor) + size(e.symbol_table))
                (variable
                    (name descriptor)
                    (type state_descriptor)
                    (offset 0)
                    (marked descriptor)
                )
                (scope
                    (name global_variables)
                    (offset 6)
                    (marked invisible)
                    evaluate(e.symbol_table)
                )
            )
            and predefined_promela_types
            and sdi.types
            and sdi.mtype
            and sdi.locations
        )
    else return
        (scope
            (name concat("s", id))
            (marked invisible)
            (offset offset)
            evaluate(e.symbol_table)
        )
```

Figure B.4: Global Component SDI code template

a list of all the Promela SDI. The global component SDI is generated and its sub-symboltable is evaluated. If the *id* of the scope is not zero then the scope is an ordinary scope and its sub-symboltable is evaluated.

During the evaluation a structure called *sdi* is used to store information, an instance of the `DebuggingInfo` class The types are saved in the *sdi.types* variable and the `mtype` are saved in the a *sdi.mtype* variable. The location information is stored in the *sdi.locations* variable and is not deduced from the symbol table.

### B.2.2 Types

The information regarding the type definitions is stored in the symbol table. Promela allows type definitions by use of the `typedef` keyword. For each `typedef` a *user type* entry is generated. Depending on the type definition sub-entries specify the fields of the user type.

The `mtype` Promela feature allows the user to define one enumeration type which allows the user to view symbolic values instead of bytes. The `mtype` is encoded in debugging information using the *enum_type* entry with the type name `mtype` which is a reserved type name in PROMELA. Since *mtype* declarations may be done anywhere in the model the enumeration type entry can only be generated when all declarations have been inspected. For each of the symbolic `mtype` values the appropriate *enum_field* entry is generated as a sub-entry of the enum_type entry.

Array type declarations are stored in `SymTabArrayUType` class objects. An *array_type* entry is generated for these type declarations. The *name*, *size*, *length* and *type* attributes of the array are determined by information contained within the `SymTabArrayUType` class entry.

For each *process type* or *channel type* an *id* attribute value is generated. A global label variable is incremented every time a component type is added to the symbol table. There is one counter variable for process types and one for channel types. NIPS VM process and channel components each have one byte available to store the component *type id* thus there can be 256 component types of each. Figure B.5 shows the process type code template.

For each process type declaration a *process_type* entry is generated. The process type starts with a variable named *descriptor* which contains the process descriptor. The process descriptor type is defined by the NIPS VM. The process local scope follows the descriptor. Since the NIPS Compiler does not save offsets with respect to the component but only with respect to the local variables the offset of the local scope within the component must be displaced using an invisible scope level that offsets the local scope to the correct offset.

For each channel declaration a *channel_type* entry is generated. The channel type starts with a variable named *descriptor* which contains the channel descriptor. Figure B.6 shows the channel code template.

### B.2.3 Locations

Location information is buried between the translation from source code to target code. NIPS Compiler AST's neither allow easy access to this information nor do the functions for code generation. The approach sketched in Section 4.6 does therefore not work. The designers of the compiler choose a different approach for generating debugging information. Information strings were output as special information instructions which are filtered out by the assembler. The

```
evaluate(SymTabProc e) = sdi.addType(
        (proc_type
                (name e.name)
                (size size(process_descriptor) + size(e.argument_scope))
                (id e.proctype_id)
                (pc_low e.pc_low)
                (pc_high e.pc_high)
                (variable
                        (name descriptor)
                        (type process_descriptor)
                        (offset 0)
                        (marked descriptor)
                )
                (scope
                        (name local_variables)
                        (marked invisible)
                        (offset size(process_descriptor))
                        evaluate(e.argument_scope)
                        evaluate(e.body_scope)
                )
        )
)
```

Figure B.5: Process SDI code template

location information contains both names and locations with *start* and *end* tags. The output stream of instructions is scanned for these instructions to filter out a target to source mapping which is saved in *t2s* SDI entries. The location information in the byte code is left intact, it can be accessed via the NIPS VM API.

```
evaluate(SymTabVarChan e) = sdi.addType(
        (message_type
                (name concat("msg_type", ctype_id))
                (id ctype_id)
                (size sum(msg_parts))
                for(int i = 0; i¡msgs.length; i++){
                        (variable
                                (name concat("msg_part", i))
                                (type msg[i].type)
                                (offset msg_offset)
                                (marked invisible)
                        )
                }
        )
)
and sdi.addtype(
        (chan_type
                (name e.name)
                (size size(channel_descriptor) + size(e.type_preamble))
                (id ctype_id)
                (max_len maxLength)
                (variable
                        (name descriptor)
                        (marked descriptor)
                        (type channel_descriptor)
                        (offset 0)
                )
                for(int i = 0; i¡msg.length; i++){
                        (variable
                                (name concat("type_bits_msg_part", i))
                                (type byte)
                                (offset size(channel_descriptor) + i)
                                (marked descriptor)
                        )
                        msg_addr + +;
                }
        )
)
and return
        (variable
                (name e.name)
                (size e.size)
                (type cid)
                (offset e.offset)
        )
```

Figure B.6: Channel SDI code template

# Appendix C

# Furniture Factory Example

In the Furniture Factory Example introduced in Section 3.2 we use the Nips Promela Compiler to translate the Promela model of Figure 3.5(a) to Nips Byte Code. Because of the amount of Byte Code and SDI generated only part of it was shown in Figure 3.5(b). The full Byte Code can be found in a separate Appendix not included with this thesis report, the full SDI is given in Appendix C.1.

## C.1 SDI

```
(
    (proc_type
        (name init)
        (size 12)
        (id 0)
        (pc_low 0)
        (variable
            (name descriptor)
            (type process_descriptor)
            (offset 0)
            (marked descriptor)
        )
        (variable
            (name magic_variable)
            (type int)
            (offset 8)
            (marked descriptor)
        )
    )
    (base_type
        (name int)
        (size 4)
        (format besint)
        (bits -31)
    )
    (base_type
        (name short)
        (size 2)
        (format besint)
        (bits -15)
    )
    (base_type
        (name byte)
        (size 1)
        (format besint)
        (bits 8)
    )
    (base_type
        (name bit)
        (size 1)
        (format beuint)
        (bits 1)
    )
    (base_type
        (name bool)
        (size 1)
        (format bool)
        (bits 1)
    )
    (base_type
        (name pid)
        (size 1)
        (format beuint)
        (bits 8)
    )
    (base_type
        (name cid)
        (size 2)
        (format beuint)
        (bits 16)
    )
    (base_type
        (name beuint8_t)
        (size 1)
        (format beuint)
        (bits 8)
```

```
    )
    (base_type
        (name beuint16_t)
        (size 2)
        (format beuint)
        (bits 16)
    )
    (base_type
        (name beuint32_t)
        (size 4)
        (format beuint)
        (bits 32)
    )
    (user_type
        (name channel_descriptor)
        (size 7)
        (variable
            (name cid)
            (type beuint16_t)
            (offset 0)
        )
        (variable
            (name max_length)
            (type beuint8_t)
            (offset 2)
        )
        (variable
            (name cur_length)
            (type beuint8_t)
            (offset 3)
        )
        (variable
            (name msg_length)
            (type beuint8_t)
            (offset 4)
        )
        (variable
            (name type_length)
            (type beuint8_t)
            (offset 5)
        )
        (variable
            (name ctype_id)
            (type beuint8_t)
            (offset 6)
        )
    )
    (user_type
        (name process_descriptor)
        (size 8)
        (variable
            (name pid)
            (type beuint8_t)
            (offset 0)
        )
        (variable
            (name flags)
            (type beuint8_t)
            (offset 1)
        )
        (variable
            (name lvar_size)
            (type beuint8_t)
            (offset 2)
        )
        (variable
            (name ptype_id)
            (type beuint8_t)
            (offset 3)
        )
        (variable
            (name pc)
            (type beuint32_t)
            (offset 4)
        )
    )
    (user_type
        (name state_descriptor)
        (size 6)
        (variable
            (name gvar_size)
            (type beuint16_t)
            (offset 0)
        )
        (variable
            (name process_count)
            (type beuint8_t)
            (offset 2)
        )
        (variable
            (name exclusive_pid)
            (type beuint8_t)
            (offset 3)
        )
        (variable
            (name monitor_pid)
            (type beuint8_t)
            (offset 4)
        )
        (variable
            (name channel_count)
            (type beuint8_t)
            (offset 5)
        )
    )
    (global_component_type
        (offset 0)
        (size 16)
        (variable
            (name descriptor)
            (type state_descriptor)
            (offset 0)
            (marked descriptor)
        )
        (scope
             (name global_variables)
             (offset 6)
             (marked invisible)
             (variable
                 (name sold_chairs)
                 (type int)
                 (offset 6)
             )
             (variable
                 (name sold_tables)
                 (type int)
                 (offset 2)
             )
             (variable
                 (name trucks)
                 (type cid)
                 (offset 0)
             )
        )
    )
    (proc_type
        (name producer)
        (size 12)
        (id 1)
        (pc_low 138)
        (pc_high 525)
        (variable
            (name descriptor)
            (type process_descriptor)
            (offset 0)
            (marked descriptor)
```

```
        )
        (scope
            (name local_variables)
            (marked invisible)
            (offset 8)
        (scope
            (name s2)
            (marked invisible)
            (offset 0)
            (variable
                (name chairs)
                (type short)
                (offset 0)
            )
            (variable
                (name tables)
                (type short)
                (offset 2)
            )
        )
        )
    )
    (proc_type
        (name storage)
        (size 18)
        (id 2)
        (pc_low 526)
        (pc_high 743)
        (variable
            (name descriptor)
            (type process_descriptor)
            (offset 0)
            (marked descriptor)
        )
        (scope
            (name local_variables)
            (marked invisible)
            (offset 8)
            (scope
                (name s18)
                (marked invisible)
                (offset 0)
                (variable
                    (name amount)
                    (type short)
                    (offset 8)
                )
                (variable
                    (name chairs)
                    (type int)
                    (offset 0)
                )
                (variable
                    (name tables)
                    (type int)
                    (offset 4)
                )
            )
        )
        )
    )
    (message_type
      (name msg_type0)
      (id 0)
      (variable
          (name msg_part0)
          (type mtype)
          (offset 0)
      )
      (variable
          (name msg_part1)
          (type short)
          (offset 1)
      )
      (size 3)
    )
    (chan_type
      (name trucks)
      (size 9)
      (id 0)
      (max_len 2)
      (variable
          (name descriptor)
          (marked descriptor)
          (type channel_descriptor)
          (offset 0)
      )
      (variable
          (name type_bits_msg_part0)
          (type byte)
          (offset 7)
          (marked descriptor)
      )
      (variable
          (name type_bits_msg_part1)
          (type byte)
          (offset 8)
          (marked descriptor)
      )
    )
    (enum_type
        (name mtype)
        (size 1)
        (format enum)
        (bits 8)
        (field
            (name chair)
            (value 0)
        )
        (field
            (name table)
            (value 1)
        )
    )
    ... t2s entries ...
)
```

# Appendix D

# Nips Debugger User Manual

The simulator is used to visualize states and transitions. It allows stepwise walking through the model. And we may choose to view global variables, local variables and variables related to the state format.

| **Visualization** | **Argument** | **Default** |
|---|---|---|
| Global variables | -g | on |
| Local variables | -l | on |
| State variables | -v | off |

Table D.1: Visualization Modes

Where for exhaustive model checkers for any given state each outgoing transition is evaluated the simulator evaluates only one outgoing transition. The simulator visualizes the current state and the outgoing transitions. The visualization modes are shown in Table D.1.

It either performs a step forward choosing a transition to a successor of the current state or it performs a step back to the predecessor of the current state. Note that forward steps can only be performed if a successor state exists and that a step back can only be performed is a predecessor exists. e.g. the initial state has no predecessor and deadlock states have no successors. Successor states are numbered from one to n with one the first successor and n the last, the predecessor state is always numbered zero.

The simulator may be run in one of four main modes. The simulator runs either interactively or automatically and it runs either guided or unguided. Each of the modes behaves differently. The simulation modes are shown in Table D.2.

| **Mode** | **Automatic** | **Interactive** |
|---|---|---|
| **Unguided** | Random (default) | Interactive |
| **Guided** | Automatic Guided | Interactive Guided |

Table D.2: Simulation Modes

In automatic mode the simulator can be influenced to run a finite number of steps. By default the simulator runs indefinitely in random mode unless a finite number of steps is given in which case it terminates after this number of steps. Furthermore the simulator may be delayed between

transitions such that it runs at a user readable speed. The automatic simulation options are shown in Table D.3 and the interactive simulation options are shown in Table D.4.

| Mode | Argument | Default | Description |
|---|---|---|---|
| **Finite** | -uN | off | Terminate after N steps. |
| **Delayed** | -dN | off | Delay transitions by N milliseconds. |

Table D.3: Automatic Simulation Options

| Mode | Argument | Description |
|---|---|---|
| **Interactive** | -i | Run the simulator interactively. |
| **Guided** | -tN | Load trace number N related to the model and run the simulator guided by a trace. |

Table D.4: Interactive Simulation Options

## Random Simulation

In default mode the simulator runs randomly starting at the initial state. It chooses one of the numbered transitions. This may either be a step forward or a step back. Note that there is no guarantee for progress as it can repeatedly step forward and back between the same two states.

## Automatic Guided Simulation

In automatic guided simulation the simulator repeatedly chooses the transition within the trace until the trace has been completed or until it is no longer possible. If the transition form the trace cannot be simulated the user is prompted to ether go to random simulation or quit. The trace transition is always numbered one.

## Interactive Simulation

In interactive mode the simulator allows the user to choose a transition. The user may step back by selecting transition zero or let the simulator perform a step forward by selecting a transition number equal or or higher than one. The stack trace can be saved to a file at any moment. The simulator may be set to random mode also in which case user input is no longer accepted. The interactive commands are shown in Table D.5.

Figure D.1 gives a view of the command-line debugger. In this situation a state is shown from the Furniture Factory example, introduced in Section 3.2.2, where only two transitions are possible. A message is either removed from channel *trucks* or the previous transition is undone.

| Command | Description |
|---|---|
| **0** | Do a step back. Pops the top from the stack trace. |
| **1..n** | Do a step forward. Pushes the state and transition. |
| **r** | Go to random mode. |
| **lN** | Load trace number N and start a guided simulation. |
| **s** | Save the current stack trace. |
| **e** | Edit the current state. |
| **exit** | Exit. |

Table D.5: Interactive Commands

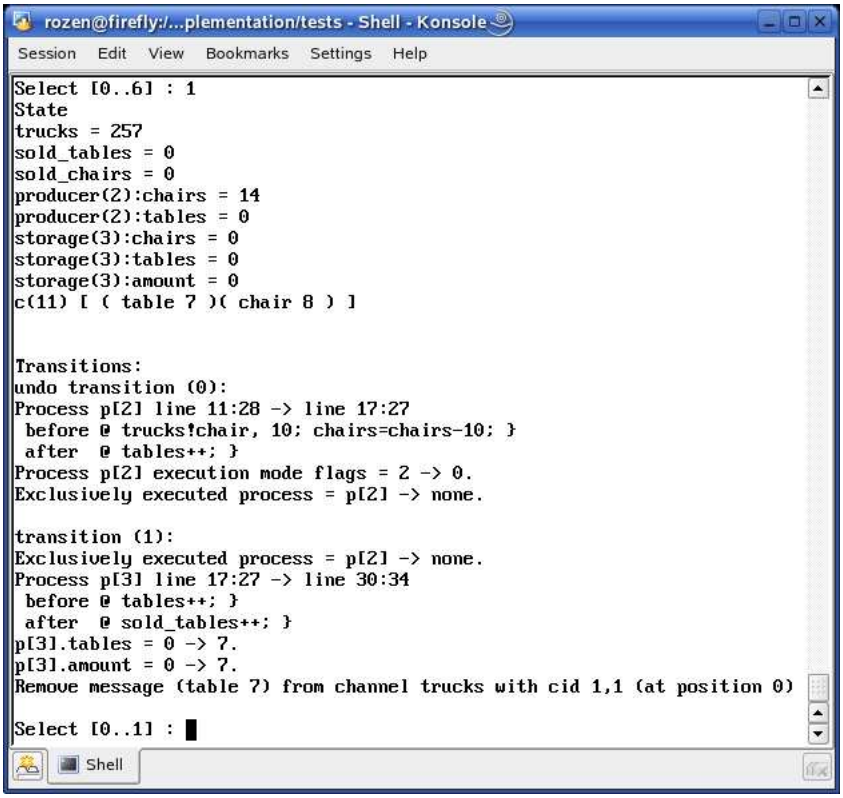



Figure D.1: NIPS Debugger Screen-shot

## Interactive Guided Simulation

In interactive guided simulation the user may select either a transition from the trace if this is possible or may choose to enter interactive unguided simulation. When the transition from the trace cannot be simulated the user is prompted to either go to unguided interactive simulation or quit.

## Interactive State Editing

In interactive state editing mode the user may change variable values. First the user inputs the variable name. Variable names are freely accessible from the state printouts which occur after every transition. Next, the user is asked to input a new value which is set in a buffer state vector. New values may be entered until the user commits the changes and returns to the simulation. Committing to the newly created state is shown as a transition to the new state. The changes can also be cancelled in which case the current state is kept. The transition can be undone by selecting the inverse transition at the next choice. This feature offers the freedom to explore unreachable system states. It is left up to the user to decide which interactive simulations are deemed useful.