Slicing Techniques Applied to Concurrent Languages

Salvador Tamarit Muñoz

Departamento de Sistemas Informáticos y Computación

Universidad Politécnica de Valencia

Memoria presentada para optar al título de:

Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Director:

Germán Fco. Vidal Oriola

Valencia, Diciembre 2008

Prologue

Abstract

Program slicing is a well-known technique in imperative programming for extracting those statements of a program that may affect a given program point. This thesis introduces new slicing techniques for concurrent languages. These languages are CSP and Petri Nets.
The Communicating Sequential Processes (CSP) language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis, reliability analysis, refinement checking, etc.) Here, a static slicing based technique to slice CSP specifications is described. Given a particular event in a CSP specification, our technique allows us to know what parts of the specification must necessarily be executed before this event, and what parts of the specification could be executed before it in some execution. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the CSP specification. Our technique is based on a new data structure which extends the synchronized CFG. We show that this new data structure improves the synchronized CFG by taking into account the context in which processes are called and, thus, makes the slicing process more precise. Furthermore, we

give a detailed explanation of the implementation of the technique. This implementation helps us to show that the theoretical work is giving successful results.
The other language, Petri nets, provides a means for modelling and verifying the behavior of concurrent systems. In this context, computing a net slice can be seen as a graph reachability problem. In this thesis, we propose two slicing techniques for Petri nets that can be useful to reduce the size of the considered net, thereby simplifying subsequent analysis and debugging tasks by standard Petri net techniques.

Acknowledgment

This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008.

Contents

Chapter 1

Introduction

In this thesis are presented some slicing techniques for two concurrent languages: CSP and Petri Nets. This introductory chapter explains the motivation for this work and gives a preliminary idea of what will be seen in the main chapters of the thesis. First, we will see what exactly means the concept of program slicing and after, in two separate sections, its application to CSP and Petri Nets, respectively.

Program slicing is a well-known technique to extract the part of a program which is related to some point of interest known as slicing criterion. It was first proposed as a debugging technique [Wei79] to allow a better understanding of the portion of code which revealed an error. In particular, Weiser's proposal was aimed at using program slicing for isolating the program staments that may contain a bug, so that finding this bug becomes simpler for the programmer. In general, slicing extracts the statements that may affect some point of interest, referred to as *slicing criterion*. Nowadays, it has been successfully applied to a wide variety of software engineering tasks, such as program understanding, debugging, testing, specialization, etc.

Program slicing is also a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of

```
(1)  read(n) ;
(2)  i := 1 ;
(3)  sum := 0 ;
(4)  product := 1 ;
(5)  while i <= n do
        begin
(6)        sum := sum + i ;
(7)        product := product * i ;
(8)        i := i + 1 ;
        end ;
(9)  write (sum) ;
(10) write (product) ;
```

(a) Example program.

```
read(n) ;
i := 1 ;

product := 1 ;
while i <= n do
   begin

      product := product * i ;
      i := i + 1 ;
   end ;

write (product) ;
```

(b) Program slice w.r.t. (`10`,`product`).

Figure 1.1: Sub-figures 1.1(a) and 1.1(b) show an example of program slicing.

those parts of a program which are (potentially) related with the values computed at some slicing criterion. Program slices are usually computed from a *Program Dependence Graph* (PDG) [FOW87] that makes explicit both the data and control dependencies for each operation in a program. Program dependencies can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*. A slice is said to be *static* if the input of the program is unknown (this is the case of Weiser's approach). On the other hand, it is said to be *dynamic* if a particular input for the program is provided, i.e., a particular computation is considered. A survey on slicing can be found, e.g., in [Tip95, BG96].

Let us illustrate this technique with an example taken from [Tip95]. Figure 1.1(a) shows a simple program which requests a positive integer number n and computes the sum and the product of the first n positive integer numbers. Figure 1.1(b) shows a slice of this program w.r.t. the slicing criterion (`10`,`product`), i.e., variable `product` in line `10`. As can be seen in the figure, all the computations that do not contribute to the final value of the variable `product` have been removed from the slice.

Here, we apply program slicing to two well-known languages that provide a means for modelling and verifying the behavior of concurrent systems. The first one is the *Communicating Sequential Processes* (CSP) [Hoa83] language, a formal language member of the family of mathematical theories of concurrency known as process algebras. The second one used is Petri Nets, one of several mathematical modeling languages for the description of discrete distributed systems. The rest of the chapter gives a brief introduction to the slicing techniques designed for both of this languages.

1.1 Slicing CSP

The *Communicating Sequential Processes* (CSP) [Hoa83] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [LS95], reliability analysis [KSS95], refinement checking [RGGHJS95], etc.). Other aspects of CSP also lead to undecidability: which face undecidable problems due to the nondeterministic execution order of parallel and interleaved processes.

In Chapter 3 we introduce a static analysis technique for CSP which is based on program slicing. Our technique allows us to extract the part of a CSP specification which is related to a given event (referred to as the slicing criterion) in the specification. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the CSP specification.

In particular, given an event in a specification, our technique allows us to extract those parts of the specification which must be executed before the specified event (thus they are an implicit precondition); and those parts of the specification which could, and could not, be executed before it.

Example 1.1.1 *Consider the following specification*[1] *with three processes (*`STUDENT`*,* `PARENT` *and* `COLLEGE`*) executed in parallel and synchronized on common events. Process* `STUDENT` *represents the three-year academic courses of a student; process* `PARENT` *represents the parent of the student who gives her a present when she passes a course; and process* `COLLEGE` *represents the college who gives a prize to those students which finish without any fail.*

$$\mathtt{MAIN = (STUDENT \mathbin{||} PARENT) \mathbin{||} COLLEGE}$$

$$\mathtt{STUDENT = year1} \rightarrow \underline{\mathtt{(}}\mathtt{pass} \rightarrow \mathtt{YEAR2} \;\square\; \underline{\mathtt{fail} \rightarrow \mathtt{STUDENT)}}$$

$$\mathtt{YEAR2 = year2} \rightarrow \underline{\mathtt{(}}\mathtt{pass} \rightarrow \mathtt{YEAR3} \;\square\; \boxed{\mathtt{fail}} \; \underline{\rightarrow \mathtt{YEAR2)}}$$

$$\mathtt{YEAR3 = year3} \rightarrow \mathtt{(pass} \rightarrow \mathtt{graduate} \rightarrow \mathtt{STOP} \;\square\; \mathtt{fail} \rightarrow \mathtt{YEAR3)}$$

$$\mathtt{PARENT = pass} \; \underline{\rightarrow \mathtt{present} \rightarrow} \; \mathtt{pass} \rightarrow \mathtt{present} \rightarrow \mathtt{pass} \rightarrow$$

$$\mathtt{present} \rightarrow \mathtt{STOP}$$

$$\mathtt{COLLEGE = fail} \rightarrow \mathtt{STOP} \;\square\; \mathtt{pass} \rightarrow \mathtt{C1}$$

$$\mathtt{C1 = fail} \rightarrow \mathtt{STOP} \;\square\; \mathtt{pass} \rightarrow \mathtt{C2}$$

$$\mathtt{C2 = fail} \rightarrow \mathtt{STOP} \;\square\; \mathtt{pass} \rightarrow \mathtt{prize} \rightarrow \mathtt{STOP}$$

In this specification, we are interested in determining what parts of the specification must be executed before the student fails in the second year, hence, we mark event `fail` *of process* `YEAR2` *(thus the slicing criterion is (*`YEAR2`*,* `fail`*)). Our slicing technique automatically extracts the slice composed by the black parts. We can additionally be interested in knowing which*

[1] We refer those readers non familiarized with CSP syntax to Section 2.1 where we provide a brief introduction to CSP.

parts could be executed before the same event. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event. Note that, in both cases, the slice produced could be made executable by replacing the removed parts by "`STOP`" or by "$\longrightarrow$ `STOP`" if the removed expression has a prefix.

It should be clear that computing the minimum slice of an arbitrary CSP specification is a non-decidable problem. Consider for instance the following CSP specification:

```
MAIN = P ⊓ Q

P = X ; Q

Q = a → STOP

X = Infinite Process
```

together with the slicing criterion (`Q`, `a`). Determining whether X does not belong to the slice implies determining that X is an infinite process which is known to be an undecidable problem [Wei84].

We explain our static slicing technique for CSP in Chapter 3. Here, we give a previous overview of this chapter through the following points:

- We define two new static analyses for CSP and propose algorithms for their implementation. Despite their clear usefulness we have not found these static analyses in the literature.
- We define the context-sensitive synchronized control flow graph and show its advantages over its predecessors. This is a new data structure able to represent all computations taking into account the context of

process calls; and it is particularly interesting for slicing languages with explicit synchronization.

- We have implemented our technique in a prototype integrated in ProB [LB08, BL05, LF08]. Preliminary results are very encouraging.

1.2 Slicing Petri Nets

A Petri net [Mur89, Pet81] is a graphic, mathematical tool used to model and verify the behavior of systems that are concurrent, asynchronous, distributed, parallel, non-deterministic and/or stochastic. As a graphic tool, they provide a visual understanding of the system and the mathematical tool facilitates its formal analysis. State space methods are the most popular approach to automatic verification of concurrent systems. In their basic form, these methods explore the transition system associated with the concurrent system. The transition system is a graph, known as the *reachability graph*, that represents the system's reachable states as nodes: there is an arc from one state s to another s', whenever the system can evolve from s to s'. In the worst case, state space methods have to explore all the nodes and transitions in the transition system. This makes the method useless in practice, even though it is simple in concept, due to the state-explosion problem that occurs when a Petri net is applied to nontrivial real problems. The technique is costly even in bounded nets with a finite number of states since, in the worst case, the reachable states are multiplied beyond any primitive recursive function. For this reason, various approaches have been proposed to minimize the number of system states to be studied in a reachability graph [Rau90].

Program slicing has a great potential here since it allows us to syntactically reduce a model in such a way that the reduced model is composed only of those parts that may influence the slicing criterion. Since it was originally defined by Weiser, program slicing has been applied to different formalisms which are not strictly programming languages, like attribute grammars [SH96], hierarchical state machines [HW97], Z and CSP-OZ specifications [CR94, Brü04, BW05], etc. Unfortunately, very little work has been carried out on slicing for Petri nets (some notable exceptions are [CW86, LCKK00, Rak07, Rak08]). For instance, Chang and Wang [CW86] present a static slicing algorithm for Petri nets that slices out all sets of paths, known as concurrence sets, so that all paths within the same set should be executed concurrently. In [LCKK00], a static slicing technique for Petri nets is proposed in order to divide enormous P/T nets into manageable modules so that the divided model can be analyzed by a compositional reachability analysis technique. A Petri net model is partitioned into concurrent units (Petri net slices) using minimal invariants. In order to preserve all the information in the original model, uncovered places should be added into minimally-connectable concurrent units since minimal invariants may not cover all the places. Finally, in [Rak07, Rak08], Rakow presents another static slicing technique to reduce the Petri net size and, thus, lessen the problem of state explosion that occurs in the *model checking* [CGP00] of Petri nets [BH05]. From the best of our knowledge, there is no previous proposal for *dynamic* slicing of Petri nets. This is surprising because considering an initial marking and/or a particular sequence of transition firings would allow us to further reduce the size of the slices and focus on a particular use of the considered Petri net.

In Chapter 4, we explore two different alternatives for dynamic slicing of Petri nets. Firstly, we present a slicing technique that extends the slicing criterion in [Rak07, Rak08] in order to also consider an initial marking. We show that this information can be very useful when analyzing Petri nets and, moreover, it allows us to significantly reduce the size of the computed slice. Furthermore, we show that our algorithm is, in the worst case, as precise as Rakow's algorithm. This can still be seen as a lightweight approach to slicing since its cost is bounded by the number of transitions in the Petri net. Then, we present a second approach that further reduces the size of the computed slice by only considering a particular execution—here, a sequence of transition firings. Clearly, in this case the computed slice is only useful to analyze the considered firing sequence. We illustrate both techniques with examples.

Chapter 2

Preliminaries

In order to keep the thesis self-contained, in this chapter we introduce separably some basic notions of CSP and Petri Nets. Here we introduce some notations and concepts and provide a precise terminology that will be used in the rest of the thesis.

2.1 Communicating Sequential Processes

Figure 2.1 summarizes the syntax constructions used in our CSP specifications. A specification is a finite collection of definitions. The left-hand side of each definition is the name of a different process, which is defined in the right-hand side by means of an expression that can be a call to another process or a combination of the following operators:

Prefixing. It specifies that event a must happen before expression π.

Internal choice. The system chooses (e.g., nondeterministically) to execute one of the two expressions.

External choice. It is identic to internal choice but the choice comes from outside the system (e.g., the user).

$$
\begin{array}{lclll}
S & ::= & D_1 \ldots D_m & \text{(entire specification)} & \textit{Domains} \\
D & ::= & P = \pi & \text{(definition of a process)} & P, Q, R \ldots \ \textsf{(processes)} \\
\pi & ::= & Q & \text{(process call)} & a, b, c \ldots \ \textsf{(events)} \\
 & | & a \rightarrow \pi & \text{(prefixing)} & \\
 & | & \pi_1 \ \sqcap \ \pi_2 & \text{(internal choice)} & \\
 & | & \pi_1 \ \Box \ \pi_2 & \text{(external choice)} & \\
 & | & \pi_1 \ ||| \ \pi_2 & \text{(interleaving)} & \\
 & | & \pi_1 \ ||_{\{\overline{a_n}\}} \ \pi_2 & \text{(synchronized parallelism)} & \text{where } \overline{a_n} = a_1, \ldots, a_n \\
 & | & \pi_1 \ ; \ \pi_2 & \text{(sequential composition)} & \\
 & | & SKIP & \text{(skip)} & \\
 & | & STOP & \text{(stop)} &
\end{array}
$$

Figure 2.1: Syntax of CSP specifications

Interleaving. Both expressions are executed in parallel and independently.

Synchronized parallelism. Both expressions are executed in parallel with a set of synchronized events. In absence of synchronizations both expressions can execute in any order. Whenever a synchronized event $a_i, 1 \leq i \leq n$, happens in one of the expressions it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that expressions are synchronized in all common events.

Sequential composition. It specifies a sequence of two processes. When the first finishes, the second starts.

Skip. It finishes the current process. It allows us to continue the next sequential process.

Stop. It finishes the current process; but it does not allow the next sequential process to continue.

Note, to simplify the presentation we do not yet treat process parameters, nor input and output of data values on channels.

We define the following notation for a given CSP specification $\mathcal{S}$: $\mathcal{P}roc(\mathcal{S})$ and $\mathcal{E}vent(\mathcal{S})$ are, respectively, the sets of all (possible repeated) process calls and events appearing in $\mathcal{S}$. Similarly, $\mathcal{P}roc(P)$ and $\mathcal{E}vent(P)$ with $P \in \mathcal{S}$, are, respectively, the sets of all (possible repeated) processes and events in P. In addition, we define $choices(A)(parallel(A))$, where A is a set of processes, events and operators; as the subset of operators that are either an internal choice or an external choice (an interleaving or a synchronized parallelism).

We use $a_1 \rightarrow^* a_n$ to denote a feasible (sub)execution which leads from a_1 to a_n; and we say that $b \in (a_1 \rightarrow^* a_n)$ iff $b = a_i, 1 \leq i \leq n$. In the following, unless we state the contrary, we will assume that programs start the computation from a distinguished process `MAIN`.

We need to define the notion of *specification position* which, roughly speaking, is a label that identifies a part of the specification. Formally,

Definition 2.1.1 *(Position and Specification Position)*
Positions are represented by a sequence of natural numbers, where Λ denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:

$$\begin{array}{ll} \pi|_\Lambda = \pi & \textit{for all process } \pi \\ (\pi_1 \; op \; \pi_2)|_{1.w} = \pi_1|_w & \textit{for all operator } op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\} \\ (\pi_1 \; op \; \pi_2)|_{2.w} = \pi_2|_w & \textit{for all operator } op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\} \end{array}$$

Given a specification $\mathcal{S}$, we let $\mathcal{P}os(\mathcal{S})$ denote the set of all specification positions in $\mathcal{S}$. A specification position is a pair $(P, w) \in \mathcal{P}os(\mathcal{S})$ that addresses the subexpression $\pi|_w$ in the right-hand side of the definition, $P = \pi$, of process P in $\mathcal{S}$.

Example 2.1.2 *In the following specification each term has been labelled (in grey color) with its associated specification position so that all labels are unique.*

$\texttt{MAIN =}$

$(P_{(Main,1.1)}||_{\{b\}(Main,1)}Q_{(Main,1.2)});_{(Main,\Lambda)}(P_{(Main,2.1)}||_{\{a\}(Main,2)}R_{(Main,2.2)})$

$P = a_{(P,1)}\rightarrow_{(P,\Lambda)}b_{(P,2.1)}\rightarrow_{(P,2)}SKIP_{(P,2.2)}$

$Q = b_{(Q,1)}\rightarrow_{(Q,\Lambda)}c_{(Q,2.1)}\rightarrow_{(Q,2)}SKIP_{(Q,2.2)}$

$R = d_{(R,1)}\rightarrow_{(R,\Lambda)}a_{(R,2.1)}\rightarrow_{(R,2)}SKIP_{(R,2.2)}$

We often use indistinguishably an expression and its specification position (e.g., (Q,c) *and* $(Q,2.1)$*) when it is clear from the context.*

2.2 Petri Nets

A Petri net [Mur89, Pet81] is a directed bipartite graph, whose two essential elements are called *places* (represented by circles) and *transitions* (represented by bars or rectangles). The edges of the graph form the *arcs*, which are labelled with a positive integer known as *weight*. Arcs run from places to transitions and vice versa. The *state* of the system modeled by the net is represented by assigning non-negative integers to places. This is known as a *marking*, and is shown graphically by adding small black circles to the places, known as *tokens*. The *dynamic behavior* of the system is simulated by changes in the markings of a Petri net, a process which is carried out by the firing of the transitions. The basic concepts of Petri nets are summarized as follows:

Definition 2.2.1 *(Petri Net) A* Petri net *[Mur89, Pet81] is a tuple* $\mathcal{N} = (P, T, F)$*, where:*

- *P is a set of* places.
- *T is a set of* transitions*, such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.*
- *F is the* flow relation *that assigns weights to arcs: $F : P \times T \cup T \times P \to \mathbb{N}$.*

The marking *M of a Petri net is defined over the set of places P. For each place $p \in P$ we let $M(p)$ denote the number of tokens contained in p.*

A marked Petri net *Σ is a pair $(\mathcal{N}, M)$ where $\mathcal{N}$ is a Petri net and M is a marking. We denote by M_0 the* initial marking *of the net.*

In the following, given a marking M and a set of places P, we denote by $M|_P$ the restriction of M over P, i.e., $M|_P(p) = M(p)$ for all $p \in P$ and $M|_P$ is undefined otherwise.

Definition 2.2.2 *(Covering [Pet81]) Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a marking M'* covers *a marking M if $M' \geq M$, i.e., $M'(p) \geq M(p)$ for each $p \in P$.*

Given a Petri net $\mathcal{N} = (P, T, F)$, we say that a place $p \in P$ is an *input (resp. output) place* of a transition $t \in T$ iff there is an *input (resp. output) arc* from p to t (resp. from t to p). Given a transition $t \in T$, we denote by ${}^\bullet t$ and $t^\bullet$ the set of all input and output places of t, respectively. Analogously, given a place $p \in P$, we denote ${}^\bullet p$ and $p^\bullet$ the set of all input and output transitions of p, respectively.

Definition 2.2.3 *(Enabled Transitions) Let $\Sigma = (\mathcal{N}, M)$ be a marked Petri net, with $\mathcal{N} = (P, T, F)$. We say that a transition $t \in T$ is* enabled *in M, in symbols $M \xrightarrow{t}$, iff for each input place $p \in P$ of t, we have $M(p) \geq F(p, t)$. A transition may only be fired if it is enabled.*

The firing *of an enabled transition t in a marking M eliminates $F(p,t)$ tokens from each input place $p \in {}^\bullet t$ and adds $F(t,p')$ tokens to each output place $p' \in t^\bullet$, producing a new marking M', in symbols $M \xrightarrow{t} M'$.*

We say that a marking M_n is *reachable* from an initial marking M_0 if there exists a *firing sequence* $\sigma = t_1 t_2 \ldots t_n$ such that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$. In this case, we say that M_n is reachable from M_0 through σ, in symbols $M_0 \xrightarrow{\sigma} M_n$. This notion includes the empty sequence ϵ; we have $M \xrightarrow{\epsilon} M$ for any marking M. We say that a firing sequence is *initial* if it starts from an initial marking.

The set of all possible markings which are reachable from an initial marking M_0 in a marked Petri net $\Sigma = (\mathcal{N}, M_0)$ is denoted by $R(\mathcal{N}, M_0)$ (or simply by $R(M_0)$ when $\mathcal{N}$ is clear from the context).

The following notion of *subnet* will be particularly relevant in the context of slicing (roughly speaking, we will identify a slice with a subnet). Let $P' \times T' \cup T' \times P' \subseteq P \times T \cup T \times P$, we say that a flow relation $F' : P' \times T' \cup T' \times P' \to \mathbb{N}$ is a restriction of another flow relation $F : P \times T \, \cup \, T \times P \to \mathbb{N}$ over P' and T', in symbols $F|_{(P',T')}$, if F' is defined as follows: $F'(x,y) = F(x,y)$ if $(x,y) \in P' \times T' \;\cup\; T' \times P'$ and F' is not defined otherwise.

Definition 2.2.4 *(Subnet [DE95]) A* subnet *$\mathcal{N}' = (P', T', F')$ of a Petri net $\mathcal{N} = (P, T, F)$ is a Petri net such that $P' \subseteq P$, $T' \subseteq T$ and F' is a restriction of F over P' and T', i.e., $F' = F|_{(P',T')}$.*

Chapter 3

Slicing CSP

This chapter presents a static analysis technique based on program slicing for CSP specifications. Given a particular event in a CSP specification, our technique allows us to know what parts of the specification must necessarily be executed before this event, and what parts of the specification could be executed before it in some execution. Our technique is based on a new data structure which extends the *Synchronized Control Flow Graph* (SCFG). We show that this new data structure improves the SCFG by taking into account the context in which processes are called and, thus, makes the slicing process more precise. We called this new data structure *Context-Sensitive Synchronized Control Flow Graph* (CSCFG). This chapter also describes *SOC* (Slicer for CSP Specifications), the tool implemented to perform this analysis. Given a CSP specification, *SOC* generates its associated CSCFG and produces from it two different kinds of slices; which correspond to two different static analyses. We present the tool's architecture, its main applications and the results obtained from experiments conducted in order to measure the performance of the tool.

Part of the material of this chapter has been presented in the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08), held in Valencia (Spain) in 2008 [LLOST08, LLOST08b] and some parts will be present in ACM SIGPLAN 2009 Workshop on Partial Evaluation and Program Manipulation (PEPM'09), that will be held celebrated in Savannah (Georgia,USA) in 2009 [LLOST09].

3.1 Context-Sensitive Synchronized Control Flow Graphs

As it is usual in static analysis, we need a data structure able to finitely represent the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [Tip95], neither the *Interprocedural Control Flow Graph* (ICFG) [HRS98] because they cannot represent multiple threads and, thus, they can only be used with sequential programs. In fact, for CSP specifications, being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [Kri98, Kri03] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronizations between threads. Callahan and Sublok introduced a data structure [CS88], the *Synchronized Control Flow Graph* (SCFG), which explicitly represents synchronizations between threads with a special edge for synchronization flows.

For convenience, the following definition adapts the original definition of SCFG for CSP; and, at the same time, it slightly extends it with the "*start thread*" and "*end thread*" notation from tCFGs.

Definition 3.1.1 *(Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, its* Synchronized Control Flow Graph *is a directed graph, $SCFG = (N, E_c, E_s)$ where:*

- *nodes $N = \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$;*
- *and $\mathcal{S}tart(\mathcal{S}) = \{$"start P", "end P" $\mid P \in \mathcal{P}roc(\mathcal{S})\}$.*

Edges are divided into two groups:

- control-flow edges *(E_c)*
- *and* synchronization edges *(E_s).*

E_s is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two (event) nodes. E_c is a set of one-way edges (denoted with $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff one of the following is true:

- $n = P \wedge n' =$ "*start P*" *with* $P \in \mathcal{P}roc(\mathcal{S})$
- $n =$ "*start P*" $\wedge\ n' = first((P, \Lambda))$ *with* $P \in \mathcal{P}roc(\mathcal{S})$
- $n \in \{\sqcap, \Box, |||, ||\} \wedge n' \in \{first(n.1), first(n.2)\}$
- $n \in \{\rightarrow, ;\} \wedge n' = first(n.2)$
- $n = n'.1 \wedge n' = \rightarrow$
- $n \in last(n'.1) \wedge n' =\ ;$
- $n \in last((P, \Lambda)) \wedge n' =$ "*end P*" *with* $P \in \mathcal{P}roc(\mathcal{S})$

where

$$first(n) = \begin{cases} n.1 & if\ n = \rightarrow \\ first(n.1) & if\ n =\ ; \\ n & otherwise \end{cases}$$

$$last(n) = \begin{cases} \{n\} & if\ n = SKIP \\ \emptyset & if\ n = STOP \vee \\ & (n \in \{|||, ||\} \wedge \\ & (last(n.1) = \emptyset \vee last(n.2) = \emptyset)) \\ last(n.2) & if\ n \in \{\rightarrow, ;\} \\ last(n.1) \cup last(n.2) & if\ n \in \{\sqcap, \Box\} \vee \\ & (n \in \{|||, ||\} \wedge \\ & last(n.1) \neq \emptyset \wedge last(n.2) \neq \emptyset) \\ \{\text{“}end\ P\text{”}\} & if\ n = P \end{cases}$$

The SCFG can be used for slicing CSP specifications as it is described in the following example.

Example 3.1.2 *Consider the specification of Example 2.1.2 and its associated SCFG shown in Fig. 3.1 (left); for the sake of clarity we show the term represented by each specification position. If we select the node labelled* `c` *and traverse the SCFG backwards in order to identify the nodes on which* `c` *depends, we get the whole graph except nodes* `end MAIN`*,* `end R` *and* `SKIP` *at process* `R`*.*

This example is twofold: on the one hand, it shows that the SCFG could be used for static slicing of CSP specifications. On the other hand, it shows that it is still too imprecise as to be used in practice. The cause of this imprecision is that the SCFG is context-insensitive because it connects all the calls to the same process with a unique set of nodes. This causes the SCFG to mix different executions of a process with possible different synchronizations, and, thus, slicing lacks precision. For instance, in Example 2.1.2 process `P` is called twice in different contexts. It is first executed in parallel with `Q` producing the synchronization of their `b` events. Then, it is executed in parallel with

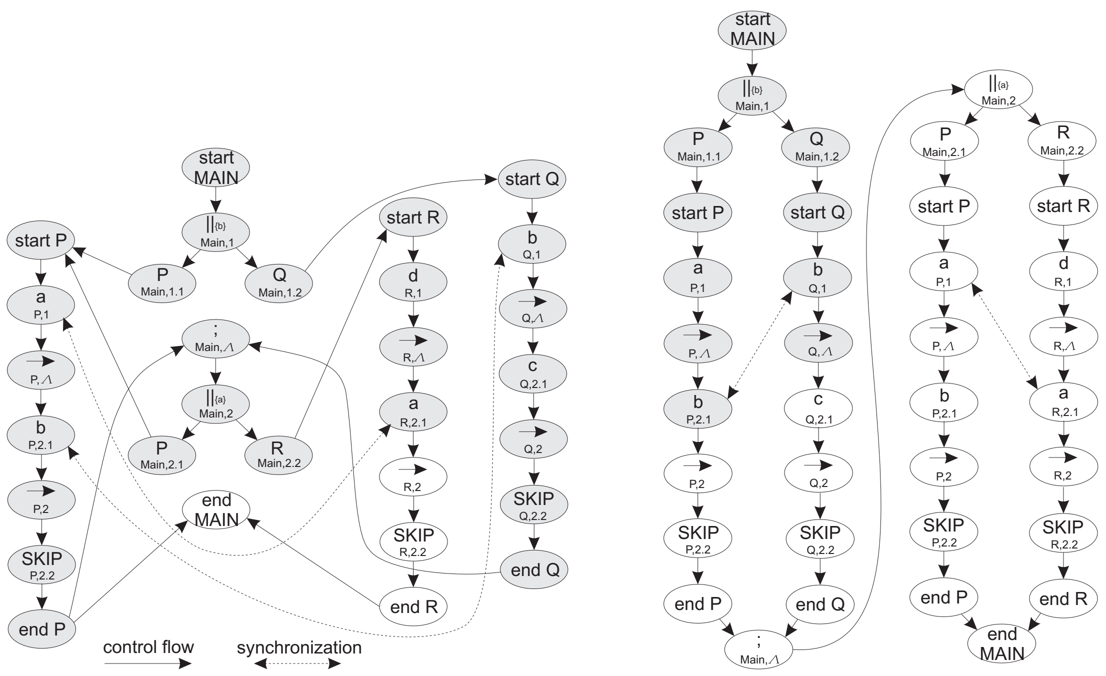


Figure 3.1: SCFG (left) and CSCFG (right) of the program in Example 2.1.2

R producing the synchronization of their a events. This makes the complete process R be part of the slice, which is suboptimal because process R is always executed after Q.

To the best of our knowledge, there do not exist other graph representations which face this problem. In the rest of this section, we propose a new version of the SCFG, the context-sensitive synchronized control flow graph (CSCFG) which is context-sensitive because it takes into account the different contexts on which a process can be executed.

Contrarily to the SCFG, inside a CSCFG the same specification position can appear multiple times. Hence, we now use labelled graphs, with nodes labelled by specification positions. Therefore, we use $l(n)$ to refer to the label of node n. We also need to define the context of a node in the graph.

Definition 3.1.3 *(Context) A* path *between two nodes* n_1 *and* m *is a sequence* $l(n_1), \ldots, l(n_k)$ *such that* $n_k \mapsto m$ *and for all* $0 < i < k$ *we have*

$n_i \mapsto n_{i+1}$. The path is loop-free *if for all $i \neq j$ we have $n_i \neq n_j$.*
Given a labelled graph $G = (N, E_c)$ and a node $n \in N$, the context of n, $\mathcal{C}on(n) = \{m \mid l(m) =$ "start P", $P \in \mathcal{P}roc(\mathcal{S})$ and exists a loop-free path $\pi = m \mapsto^* n$ with "end P" $\notin \pi\}$.

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. If we focus on a node $n \in \mathcal{P}roc(\mathcal{S})$ we can use the context to identify loops.

Definition 3.1.4 *(Context-Sensitive Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, a* Context-Sensitive Synchronized Control Flow Graph *$CSCFG = (N, E_c, E_l, E_s)$ is a SCFG graph, except in two aspects:*

1. *There is a special set of* loop edges *(E_l) denoted with $\rightsquigarrow$. $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) = P \in \mathcal{P}roc(\mathcal{S})$, $l(n_2) =$ "start P" and $n_2 \in \mathcal{C}on(n_1)$, and*

2. *Two nodes can have the same label. Every node in $\mathcal{S}tart(\mathcal{S})$ has one and only one incoming edge in E_c. Every process call node has one and only one outgoing edge which belongs to either E_c or E_l.*

For slicing purposes, the CSCFG is interesting because we can use the edges to determine if a node must be executed or not before another node, thanks to the following properties:

- if $n \mapsto n' \in E_c$ then n must be executed before n' in all executions.
- if $n \rightsquigarrow n' \in E_l$ then n' must be executed before n in all executions.
- if $n \leftrightarrow n' \in E_s$ then, in all executions, if n is executed there must be some n'' which is executed at the same time than n with $n \leftrightarrow n'' \in E_s$.

The key difference between the SCFG and the CSCFG is that the latter unfolds every process call node except those that belong to a loop. This is very convenient for slicing because every process call which is executed in a different context is unfolded, thus, slicing does not mix computations. Moreover, it allows to deal with recursion and, at the same time, it prevents from infinite unfolding of process calls; because loop edges prevent from infinite unfolding. One important characteristic of the CSCFG is that loops are unfolded once, and thus all the specification positions inside the loops are in the graph and can be collected by slicing algorithms. For slicing purposes, this representation also ensures that every possibly executed part of the specification belongs to the CSCFG because only loops (i.e., repeated nodes) are missing.

The CSCFG provides a different representation for each context in which a procedure call is made. This can be seen in Fig. 3.1 (right) where process `P` appears twice to account for the two contexts in which it is called. In particular, in the CSCFG we have a fresh node to represent each different process call, and two nodes point to the same process if and only if they are the same call (they are labelled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

Lemma 3.1.5 *(Finiteness) Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

Proof 3.1.6 *We show first that there do not exist infinite unfolding in a CSCFG. Firstly, the same start process node only appears twice in the same control loop-free path if it belongs to a process which is called from different process calls (i.e., with different specification positions) as it is ensured by the first condition of Definition 3.1.4. Therefore, the number of repeated nodes in the same control loop-free path is limited by the number of different process calls appearing in the program. However, the number of terms in the specification is finite and thus there is a finite number of different process calls. Moreover, every process call has only one outgoing edge as it is ensured*

```
MAIN = P ; P

P = Q

Q = P
```

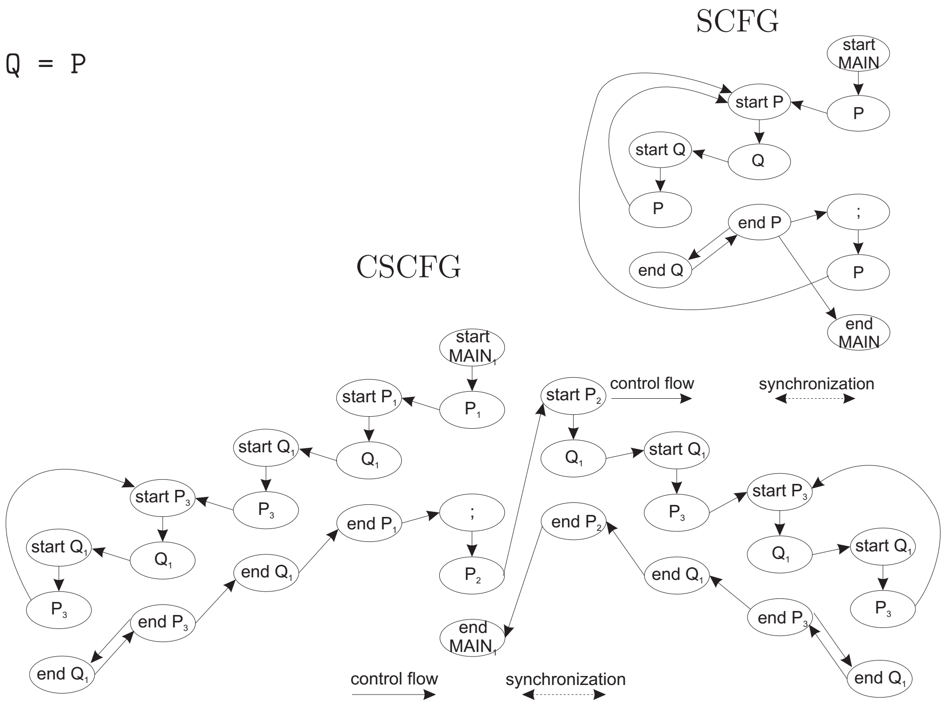


Figure 3.2: SCFG and CSCFG representing an infinite computation

by the second condition of Definition 3.1.4. Therefore, the number of paths is finite and the size of every path of the CSCFG is limited.

Example 3.1.7 *This example makes clear the difference between the SCFG and the CSCFG, consider the specification in Fig. 3.2. While the SCFG only uses one representation for the process* `P` *(there is only one* **`start P`***), the CSCFG uses four different representations because* `P` *could be executed in four different contexts. Note that due to the infinite loops, some parts of the graph are not reachable from* **`start MAIN`***; i.e., there is not possible control flow to* **`end MAIN`***. However, it does not hold in the SCFG.*

3.2 Static Slicing of CSP Specifications

We want to perform two kinds of analyses. Given an event or a process in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it; and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it.

We can now formally define our notion of slicing criterion.

Definition 3.2.1 *(Slicing Criterion) Given a specification $\mathcal{S}$, a slicing criterion is a specification position $(P, w) \in \mathcal{P}roc(\mathcal{S}) \cup \mathcal{E}vent(\mathcal{S})$.*

Clearly, the slicing criterion points to a set of nodes in the CSCFG, because the same event or process can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. As an example, consider the slicing criterion (P, a) for the specification in Example 2.1.2, and observe in its CSCFG in Fig. 3.1 (right) that two different nodes are pointed out by the slicing criterion.

This means that a slicing criterion $\mathcal{C} = (P, w)$ is used to produce a slice with respect to *all* possible executions of w. We use function $nodes(\mathcal{C})$ to refer to all the nodes in the CSCFG pointed out by the slicing criterion $\mathcal{C}$.

Given a slicing criterion (P, w), we use the CSCFG to calculate MEB. In principle, one could think that a simple backwards traversal of the graph from $nodes(\mathcal{C})$ would produce a correct slice. Nevertheless, this would produce a rather imprecise slice because this would include pieces of code which cannot be executed but they refer to the slicing criterion (e.g., dead code). The union of paths from `MAIN` to $nodes(\mathcal{C})$ is neither a solution because it would be too imprecise by including in the slice parts of code which are executed before the slicing criterion only in some executions. For instance, in the process `(b`$\rightarrow$`a`$\rightarrow$`STOP)`$\Box$`(c`$\rightarrow$`a`$\rightarrow$`STOP)`, `c` belongs to one of the paths to `a`, but it must be executed before `a` or not depending on the choice. The intersection of paths is neither a solution as it can be seen in the process `a→((b→SKIP)||(c→SKIP));d` where `b` must be executed before `d`, but it does not belong to all the paths from `MAIN` to `d`.

Before we introduce an algorithm to compute MEB we need to formally define the notion of MEB slice.

Definition 3.2.2 *(MEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; a MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $MEB(\mathcal{S},\mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N$ and $\forall\ X = (MAIN \rightarrow^* m)$, $m \in nodes(\mathcal{C})\ .\ n \in X\}$, $E'_c = \{(n, m) | n \mapsto m \in E_c$ and $n, m \in N'\}$, $E'_l = \{(n, m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$ and $E'_s = \{(n, m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}$.*

Algorithm 3.2.3 can be used to compute the MEB analysis. It basically computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification which must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set is computed with an iterative process that takes a node and (i) it follows backwards all the control-flow edges. (ii) Those nodes that could not be executed before it are added to a black list (i.e., they are discarded because they belong to a non-executed choice). And (iii) synchronizations are followed in order to reach new nodes that must be executed before it.

The algorithm always terminates. We can ensure this due to the invariant $pending \cap Meb = \varnothing$ which is always true at the end of the loop (8). Then, because Meb increases in every iteration (5) and the size of N is finite, $pending$ will eventually become empty and the loop will terminate.

Algorithm 3.2.3

Input:

- *A CSCFG* (N, E_c, E_l, E_s) *of a specification* $\mathcal{S}$
- *and a slicing criterion* $\mathcal{C}$

Output:

- *A CSCFG's subgraph*

```
Function buildMeb(n) :=
(1) $pending := \{n' \mid (n' \mapsto o) \in E_c\}$ where $o \in \{n\} \cup \{o' \mid o' \leftrightarrow n\}$
(2) $Meb := pending \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m,\ m \in pending\}$
(3) $blacklist := \{p \mid p \in N \backslash Meb$ and $o \mapsto^* p$, with $o \in choices(Meb)\}$
(4) $pending := \{q \mid q \in N \backslash (blacklist \cup Meb)$
                                    and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(5) while $\exists\ m \in pending$ do
(6)     $Meb := Meb \cup \{m\} \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m\}$
(7)     $sync := \{q \mid q \in N \backslash (blacklist \cup Meb)$
                                    and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(8)     $pending := (pending \backslash Meb) \cup sync$
(9) return $Meb$
```

Return: $MEB(\mathcal{S},\mathcal{C}) = (N' = \bigcap_{n \in nodes(\mathcal{C})} buildMeb(n),$

$$E'_c = \{(n,m) | n \mapsto m \in E_c \text{ and } n,m \in N'\},$$
$$E'_l = \{(n,m) | n \rightsquigarrow m \in E_l \text{ and } n,m \in N'\},$$
$$E'_s = \{(n,m) | n \leftrightarrow m \in E_s \text{ and } n,m \in N'\}).$$

The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node n. This means that all those nodes that must be executed before n are included, but also those nodes that are executed before n in some executions, and they are not in other executions (e.g., due to non synchronized parallelism). Formally,

Definition 3.2.4 *(CEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; a CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $CEB(\mathcal{S},\mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N \textit{ and } \exists \texttt{ MAIN} \rightarrow^* n \rightarrow^* m \textit{ with } m \in nodes(\mathcal{C})\}$, $E'_c = \{(n,m) | n \mapsto m \in E_c \textit{ and } n, m \in N'\}$, $E'_l = \{(n,m) | n \rightsquigarrow m \in E_l \textit{ and } n, m \in N'\}$, $E'_s = \{(n,m) | n \leftrightarrow m \in E_s \textit{ and } n, m \in N'\}$.*

Therefore, $MEB(\mathcal{S},\mathcal{C}) \subseteq CEB(\mathcal{S},\mathcal{C})$. The graph $CEB(\mathcal{S},\mathcal{C})$ can be computed with Algorithm 3.2.5 which, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in $MEB(\mathcal{S},\mathcal{C})$.

Algorithm 3.2.5

Input:

- *A CSCFG (N, E_c, E_l, E_s) of a specification $\mathcal{S}$*
- *and a slicing criterion $\mathcal{C}$*

Output:

- *A CSCFG's subgraph*

Initialization:

```
        $Ceb := \{m \mid m \in N_1 \textit{ and } MEB(\mathcal{S},\mathcal{C}) = (N_1, E_{c1}, E_{l1}, E_{s1})\}$
        $loopnodes := \{n \mid n_1 \mapsto^+ n \mapsto^* n_2 \rightsquigarrow n_3 \textit{ and } (n \leftrightarrow n') \notin E_s$
                $\textit{with } n' \notin Ceb,\ n_1 \in choices(Ceb) \textit{ and } n_3 \in Ceb\}$
        $Ceb := Ceb \cup loopnodes$
        $pending := \{m \mid m \notin (Ceb \cup nodes(\mathcal{C})) \textit{ and } (m' \mapsto m) \in E_c,$
```

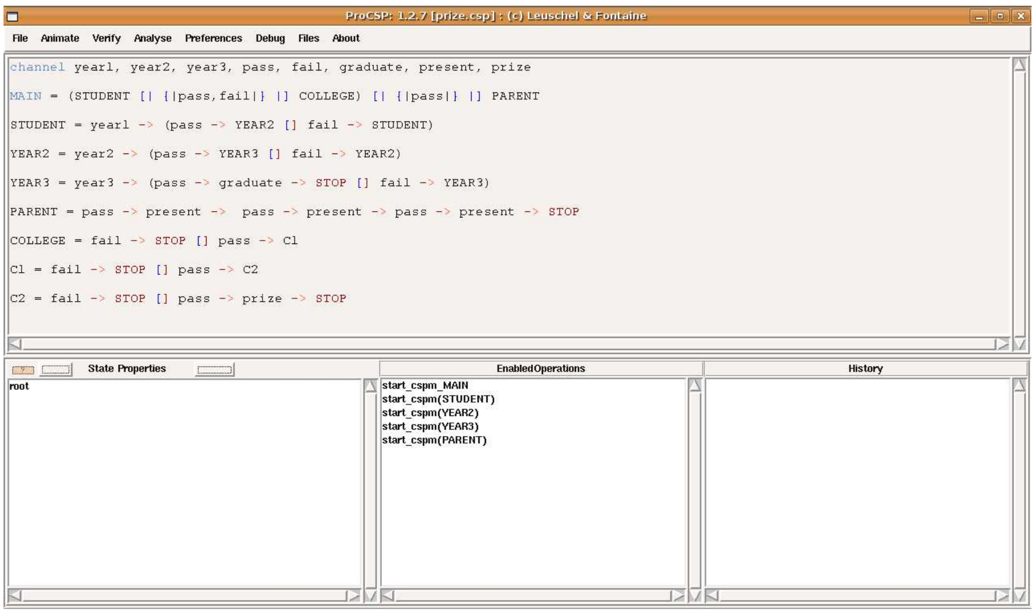
```
                                        $with\ m' \in Ceb\backslash choices(Ceb)\}$
                      $\cup\ \{p_1 \mid p \mapsto p_1 \in E_c\ and\ p \mapsto p_2 \in E_c\ and\ p_1 \neq p_2,$
                        $with\ p \in parallel(loopnodes)\ and\ p_2 \in loopnodes\}$

  Repeat
  (1) $if\ \exists\ m \in pending \mid (m \leftrightarrow m') \notin E_s\ or$
                        $((m \leftrightarrow m') \in E_s\ and\ m' \in Ceb)$
  (2) $then\ pending := pending\backslash\{m\}$
  (3)       $Ceb := Ceb \cup \{m\}$
  (4)       $pending := pending \cup \{m'' \mid (m \mapsto m'') \in E_c\ and\ m'' \notin Ceb\}$
  (5) $else\ \ if\ \exists\ m \in pending\ and\ \forall\ (m \leftrightarrow m') \in E_s\ .\ m' \in pending$
  (6)       $then\ candidate := \{m' \mid (m \leftrightarrow m') \in E_s\}$
  (7)            $Ceb := Ceb \cup \{m\} \cup candidate$
  (8)            $pending := (pending\backslash Ceb) \cup \{n \mid n \notin Ceb\ and\ m \mapsto n\}$
                      $\cup\ \{o \mid o \notin Ceb\ and\ m' \mapsto o,\ with\ m' \in candidate\}$
  Until a fix point is reached
```

Return: $CEB(\mathcal{S},\mathcal{C}) = (N' = Ceb,$

$$E'_c = \{(n,m) | n \mapsto m \in E_c\ and\ n,m \in N'\},$$
$$E'_l = \{(n,m) | n \rightsquigarrow m \in E_l\ and\ n,m \in N'\},$$
$$E'_s = \{(n,m) | n \leftrightarrow m \in E_s\ and\ n,m \in N'\}).$$

The algorithms presented can extract a slice from any specification formed with the syntax of Fig 2.1. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelisms (because they introduce synchronizations). Other operators such as prefixing, interleaving or sequential composition can be treated similarly.



```
channel year1, year2, year3, pass, fail, graduate, present, prize

MAIN = (STUDENT [| {|pass,fail|} |] COLLEGE) [| {|pass|} |] PARENT

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present ->  pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 3.3: Graphical user interface of ProB

3.3 Implementation

We have implemented the MEB and CEB analyses and the algorithm to build the CSCFG for ProB. ProB [LB08] is an animator for the B-Method which also supports other languages such as CSP [BL05, LF08]. ProB has been implemented SICStus Prolog [13] and it is publicly available at: `http://www.stups.uni-duesseldorf.de/ProB`.

It uses Tcl/Tk for the Graphical User Interface (a Java version is also available) and dot/dotty from the Graphviz package. In Figure 3.3 we show the graphical user interface of ProB. The menu bar contains the various commands to access the features of ProB. It includes the `File` menu, with a submenu `Recent Files` to quickly access the files previously opened in ProB. Notice the two couples of commands `Open/Save` and `Reopen/Save and Re-open`, the latter reopening the currently opened file and reinitialising completely the state of the animation and the model checking processes. The `About` menu provides help on the tool, including a command to check if an update is available on the ProB website. By default, ProB starts with a limited set of commands in the `Beginner` mode. The `Normal` mode gives access to more features and can be set in the menu `Preferences` → `User Mode`. Under the menu bar, the main window contains four panes:

- In the top pane, the specification of the machine is displayed with syntax highlight, and can also be edited by typing directly in this pane;
- At the bottom, the animation window is composed of three panes which display, at the current point during the animation:
  - The current state of the machine (`State Properties`), listing the current values of the machine variables;
  - The enabled operations (`Enabled Operations`), listing the operations whose preconditions and guards are true in this state;
  - The history of operations that leaded to this state (`History`).

The animation facilities of ProB allow users to gain confidence in their specifications. These features are user-friendly as the user does not have to guess the right values for the operation arguments or choice variables, and she uses the mouse to operate the animation. At each point during the animation process, several useful commands displaying various information on the machine are available in the `Analyse` menu.

The implementation of our slicer has three main modules. The first module implements the construction of the CSCFG. Nodes and control and loop edges are built following the algorithm of Definition 3.1.4. For synchronization edges we use an algorithm based on the approach by Naumovich et al. [NA98]. For efficiency reasons, the implementation of the CSCFG does some simplifications which reduces the size of the graph. For instance, "*start*" and "*end*" nodes are not present in the graph.

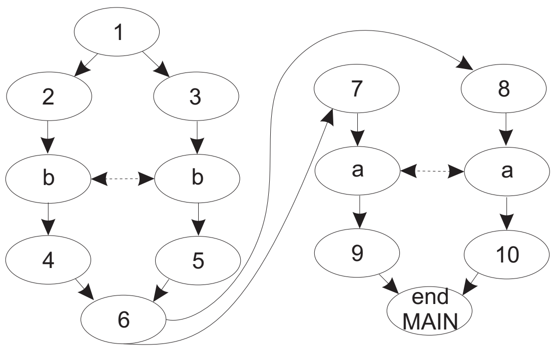


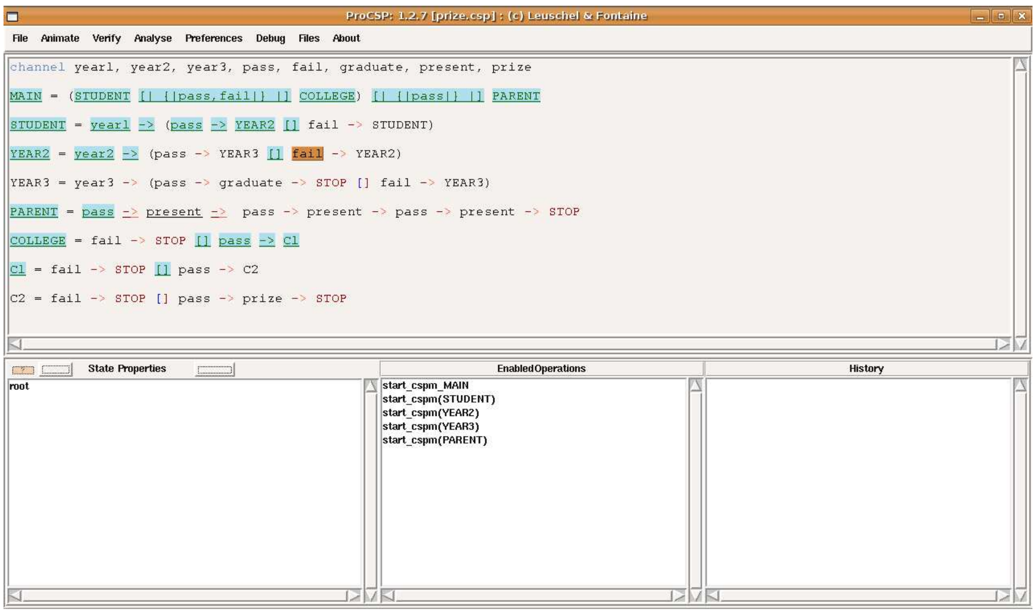
Figure 3.4: Compacted version of the CSCFG in Fig. 3.1

Another simplification to reduce the size of the graph is the graph compactation which joins together all the nodes defining a sequential path (i.e., without nodes with more than one output edge). For instance, the graph of Fig. 3.4 is the compacted version of the CSCFG in Fig. 3.1 (right). Here, e.g., node `6` accounts for the sequence of nodes `P` $\rightarrow$ `start P`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process. This is made in the second module.

The third module performs the MEB and CEB analyses by implementing Algorithm 3.2.3 and Algorithm 3.2.5. Finally, a module allows the tool to communicate with the ProB interface in order to get the slicing criterion and show the highlighted code.

We have integrated our tool into the graphical user interface of ProB. This allows us to use features such as text coloring in order to highlight the final slices. Figure 3.5 shows a screenshot of the tool showing a slice of a CSP specification.

In the following sections we give an detailed description of the tool. In Section 3.3.1 we show the applications of this tool and an example of use. In Section 3.3.2 we describe the architecture of *SOC*. In Section 3.3.3 we show how is used the tool by means of an example. Finally, in Section 3.3.4 we show a summary of some experiments which show the speedup and performance



```
channel year1, year2, year3, pass, fail, graduate, present, prize

MAIN = (STUDENT [| {|pass,fail|} |] COLLEGE) [| {|pass|} |] PARENT

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present ->  pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 3.5: Slice of a CSP specification produced by *SOC*

of our tool.

3.3.1 *SOC* in Practice

In this section, we describe the purpose of our tool and how it can be used to extract slices from CSP specifications. Let us consider the following example to show the usefulness of the technique.

Example 3.3.1 *Consider the CSP specification of Figure 3.5. It is representing the case of example 1.1.1. Our slicing technique automatically extracts the slice composed by the highlighted parts. This is what we have called MEB analysis. Therefore,* SOC *is a powerful tool for program comprehension. Note, for instance, that in order to fail in the second year, the student has necessarily passed the first year. But, the parent could or could not give a present to his son (indeed if he passed the first year) because this specification does not force the parent to give a present to his son until he has passed the* *second year. This is not so obvious from the specification, and* SOC *can help to understand the real meaning of the specification.*

We can additionally be interested in knowing what parts could be executed before the same event. This is what we have called CEB analysis. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event. This can be useful, e.g., for debugging. If the slicing criterion is an event that executed incorrectly (i.e., it should not happen in the execution), then the slice produced contains all the parts of the specification which could produce the wrong behavior.

A third application of our tool is program specialization. SOC *is able to extract executable slices with a program transformation applied to the generated slices. The specialized specification contains all the necessary parts of the original specification whose execution leads to the slicing criterion (and then, the specialized specification finishes).*

Note that, in the slices produced by both analyses in Figure 3.5, the slice produced could be made executable by replacing the removed parts by "`STOP`" or by "$\rightarrow$ `STOP`" if the removed expression has a prefix.

As described in the previous example, the slicing process is completely automatic. Once the user has loaded a CSP specification, she can select (with the mouse) the event or process call she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the source position of interest. Then, the tool internally generates an internal data structure which represents all possible computations, and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event.

There is another application of *SOC* which was our original aim when we developed this tool. ProB is able to perform different static analyses over CSP specifications. However, due to the complexity of the specifications and to the parallel and non-deterministic execution of processes, these analyses usually become too costly as to be used with real programs. *SOC* can be used as a preprocessing stage of these analyses in order to reduce the size of the specification and, thus, the size of the data structures used in, and the complexity of, the static analyses.

3.3.2 Architecture

SOC has been implemented in Prolog and it has been integrated in ProB. Therefore, *SOC* can take advantage of ProB's graphical features to show slices to the user. In order to be able to color parts of the code, it has been necessary to implement the source code positions detection; in such a way that ProB can color every subexpression which is sliced by *SOC*. Apart from the interface module for the communication with ProB, *SOC* has three main modules which we describe in the following:

Graph Generation

The first task of the slicer is to build a CSCFG. The module which generates the CSCFG from the source program is the only module which is ProB dependent. This means that *SOC* could be used in other systems by only changing the graph generation module.

Graph Compactation

The original definition of the CSCFG is inaccurate from an implementation point of view. Therefore, we have implemented a module which reduces the size of the CSCFG by removing unnecessary nodes and by joining together

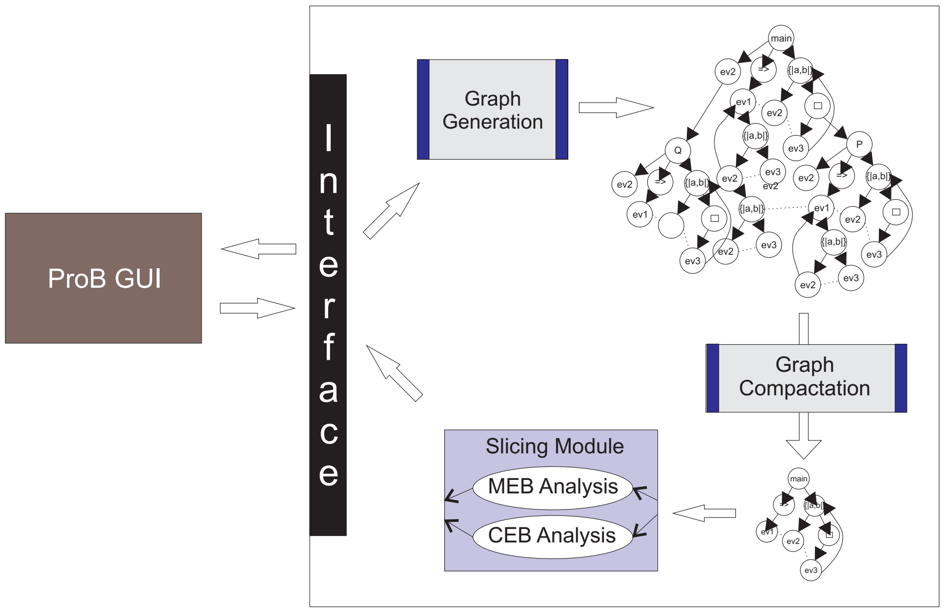
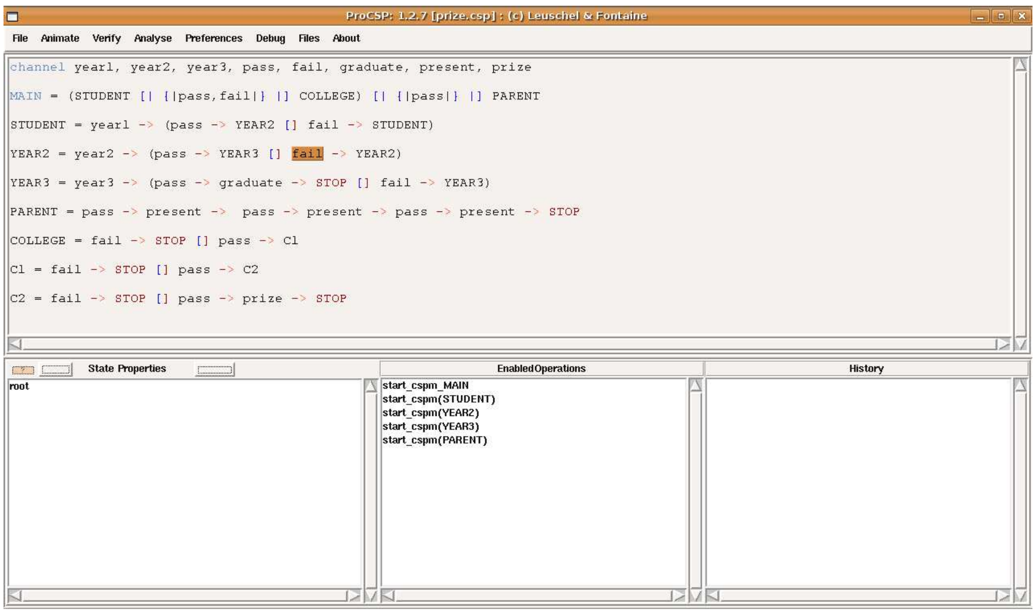


Figure 3.6: Slicer's Architecture

those nodes that form paths that the slicing algorithms must traverse in all cases. This compactation not only reduces the size of the stored CSCFG, but it also speeds up the slicing process due to the reduced number of nodes to be processed.

Slicing Module

This is the main module of the tool. It is further composed of two submodules which implement the algorithms to perform the MEB and CEB analyses on the compacted CSCFGs. Depending on the analysis selected by the user this module extracts a subgraph from the compacted CSCFG using either MEB or CEB. Then, it extracts from the subgraph the part of the source code which forms the slice. If the user has selected to produce an executable slice, then the slice is transformed to become executable (it mainly fills gaps in the produced slice in order to respect the syntax of the language). The final result is then returned to ProB in such a way that ProB can either highlight



```
channel year1, year2, year3, pass, fail, graduate, present, prize

MAIN = (STUDENT [| {|pass,fail|} |] COLLEGE) [| {|pass|} |] PARENT

STUDENT = year1 -> (pass -> YEAR2 [] fail -> STUDENT)

YEAR2 = year2 -> (pass -> YEAR3 [] fail -> YEAR2)

YEAR3 = year3 -> (pass -> graduate -> STOP [] fail -> YEAR3)

PARENT = pass -> present ->  pass -> present -> pass -> present -> STOP

COLLEGE = fail -> STOP [] pass -> C1

C1 = fail -> STOP [] pass -> C2

C2 = fail -> STOP [] pass -> prize -> STOP
```

```
root
```

```
start_cspm_MAIN
start_cspm(STUDENT)
start_cspm(YEAR2)
start_cspm(YEAR3)
start_cspm(PARENT)
```

Figure 3.7: Selecting the slicing criterion

the final slice or save a new CSP executable specification in a file.

Figure 3.6 summarizes the internal architecture of *SOC*. Note that both the graph compactation module and the slicing module take a CSCFG as input, and hence, they are independent of ProB.

3.3.3 Using the slicer

Let us consider the example 3.3.1 to show how *SOC* can be used to extract slices from CSP specifications. The user has two possibilities: to edit the specification directly in the top pane or to load it from a file (if it exists as a previously edited file). Once the program is loaded, the user can slice it with a process which is fully automatic. After this, the user has to select what will be the slicing criterion using the mouse. In our example, the slicing criterion is event `fail` of process `YEAR2`, as we can see in Figure 3.7.

Then, she selects command `Highlight Slice` from `Analyse` → `Slicing` menu. The tool internally generates the CSCFG of the specification (saved in a file .dot) and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must (respectively could) be executed before the specified event. Figure 3.5 shows a screenshot of ProB showing a slice of our CSP specification example w.r.t. the slicing criterion `(YEAR2,fail)`. We can observe highlighted in green the MEB slice and underlined the CEB slice which coincide with the expected results.

Finally, the user can view the generated CSCFG opening the corresponding .dot file. In Figure 3.8 is shown the CSCFG generated when `Analyse` → `Slicing` is selected. The nodes of the MEB slice are darker.

3.3.4 Benchmarking the slicer

In order to measure the performance and the slicing capabilities of our tool, we conducted some experiments over a subset of the examples listed in `http://www.dsic.upv.es/~jsilva/soc/examples`.

The benchmarks selected for the experiments are the following:

- `ATM.csp`. This specification represents an Automated Teller Machine. The slicing criterion is `(Menu,getmoney)`, i.e., we are interested in determining what parts of the specification must be executed before the menu option `getmoney` is chosen in the ATM.

- `RobotControlling.csp`. This example describes a game in which four robots move in a maze. The slicing criterion is `(Referee,winner2)`, i.e., we want to know what parts of the system could be executed before the second robot becomes the winner.

- `Buses.csp`. This example describes a bus service with two buses running in parallel. The slicing criterion is `(BUS37, pay90)`, i.e., we are interested in determining what could and could not happen before the user payed at bus 37.

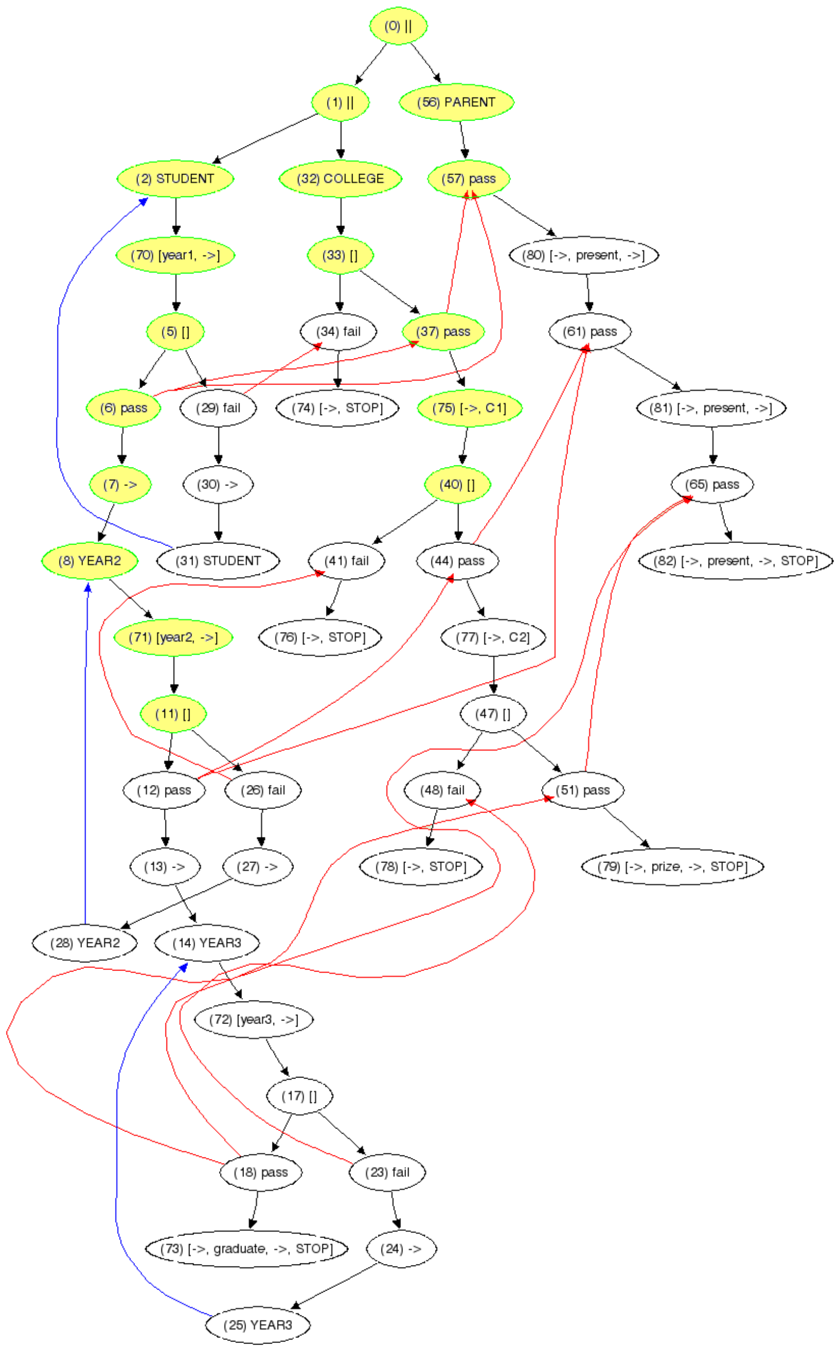


Figure 3.8: CSCFG of Example 3.3.1

- `Prize.csp`. This is the specification of Example 3.3.1. The slicing criterion is `(YEAR2,fail)`, i.e., we are interested in determining what parts of the specification must be executed before the student fails in the second year.

- `Phils.csp`. This is a simple version of the dining philosophers problem. In this example, the slicing criterion is `(PHIL221,DropFork2)`, i.e., we want to know what happened before the second philosopher dropped the second fork.

- `TrafficLights.csp`. This specification defines two cars driving in parallel on different streets. The first car can only circulate in two streets. The second car can only circulate in a third street. Each street has one traffic light for cars controlling. The slicing criterion is `(STREET3,park)`, i.e., we are interested in determining what parts of the specification must be executed before the second car parks on the third street.

- `Processors.csp`. This example describes a system that, once connected, receives data from two machines. The slicing criterion is `(MACH1,datreq)` to know what parts of the example must be executed before the first machine requests data.

- `ComplexSynchronization.csp`. This specification defines five routers working in parallel. Router i can only send messages to router i+1. Each router can send a broadcast message to all routers. The slicing criterion is `(Process3,keep)`, i.e., we want to know what parts of the system could be executed before router 3 keeps a message.

- `Computers.csp`. This benchmark describes a system in which a user can surf internet and download files. The computer can control if files are infected by virus. The slicing criterion is `(USER,consult_file)`,

Table 3.1: Benchmark time results

benchmark	CSCFG	MEB	CEB	Total
ATM.csp	1239 ms.	7083 ms.	438 ms.	8760 ms.
RobotControlling.csp	586 ms.	923 ms.	2175 ms.	3684 ms.
Buses.csp	11 ms.	17 ms.	2 ms.	30 ms.
Prize.csp	23 ms.	116 ms.	17 ms.	156 ms.
Phils.csp	39 ms.	11 ms.	152 ms.	202 ms.
TrafficLights.csp	245 ms.	115 ms.	631 ms.	991 ms.
Processors.csp	7 ms.	7 ms.	7 ms.	21 ms.
ComplexSynchronization.csp	1365 ms.	98107 ms.	250 ms.	99722 ms.
Computers.csp	40 ms.	426 ms.	11 ms.	477 ms.
Highways.csp	4555 ms.	92 ms.	40 ms.	4687 ms.

i.e., we are interested in determining what parts of the specification must be executed before the user consults a file.

- `Highways.csp`. This specification describes a net of spanish highways. The slicing criterion is `(HW6,Toledo)`, i.e., we want to determine what cities must be traversed in order to reach Toledo from the starting point.

For each benchmark, Table 3.1 summarizes the time spent to generate the compacted CSCFG (this includes the generation plus the compactation phases), to produce the MEB and CEB slices, and the total time. Table 3.2 summarizes the size, in nodes, of the graphs participating in the slicing process: Column `Ori_CSCFG` shows the size of the CSCFG of the original program. Column `Com_CSCFG` shows the size of the compacted CSCFG. Last column shows the percentage of the original program that represents the compacted version. Table 3.3, which follows the previous one, summarizes the size of the graphs when they are reduced by the slicing process: Columns `MEB Slice`

Table 3.2: Benchmark size graphs results

benchmark	Ori_CSCFG	Com_CSCFG	(%)
ATM.csp	163 nodes	110 nodes	67.48 %
RobotControlling.csp	339 nodes	123 nodes	36.28 %
Buses.csp	36 nodes	24 nodes	66.67 %
Prize.csp	70 nodes	49 nodes	70.0 %
Phils.csp	181 nodes	57 nodes	31.49 %
TrafficLights.csp	197 nodes	112 nodes	56.85 %
Processors.csp	30 nodes	15 nodes	50.0 %
ComplexSynchronization.csp	179 nodes	122 nodes	68.16 %
Computers.csp	53 nodes	34 nodes	64.15 %
Highways.csp	103 nodes	60 nodes	58.25 %

and `CEB Slice` show respectively the size of the MEB and CEB slices. Finally, column `(%)` shows the difference in size between the MEB and CEB slices. Clearly, CEB slices are always equal or greater than their MEB counterparts.

The CSCFG compactation technique has not been published. We have implemented it in our tool and the experiments show that the size of the original specification is substantially reduced using this technique. The size of both MEB and CEB slices obviously depends on the slicing criterion selected. This table compares both slices with respect to the same criterion and, therefore, gives an idea of the difference between them.

All the information related to the experiments, the source code of the benchmarks, the slicing criteria used, the source code of the tool and other material can be found at:
`http://www.dsic.upv.es/~jsilva/soc`.

Table 3.3: Benchmark size slice results

benchmark	MEB Slice	CEB Slice	(%)
ATM.csp	48 nodes	61 nodes	27.08 %
RobotControlling.csp	36 nodes	109 nodes	202.78 %
Buses.csp	11 nodes	11 nodes	0.0 %
Prize.csp	17 nodes	18 nodes	5.88 %
Phils.csp	7 nodes	44 nodes	528.57 %
TrafficLights.csp	17 nodes	72 nodes	323.53 %
Processors.csp	8 nodes	9 nodes	12.5 %
ComplexSynchronization.csp	100 nodes	116 nodes	16.0 %
Computers.csp	28 nodes	28 nodes	0.0 %
Highways.csp	14 nodes	20 nodes	42.85 %

Chapter 4

Slicing Petri Nets

In the context of Petri nets, computing a net slice can be seen as a graph reachability problem. In this chapter, we propose two slicing techniques for Petri nets that can be useful to reduce the size of the considered net, thereby simplifying subsequent analysis and debugging tasks by standard Petri net techniques.

Part of the material of this chapter has been presented in the 2nd Workshop on Reachability Problems (RP 2008) held in Liverpool (UK) in 2008 [LOSTV08].

4.1 Dynamic Slicing of Petri Nets

In this section, we introduce our first approach to dynamic slicing of Petri nets. We say that our slicing technique is *dynamic* since an initial marking is taken into account (in contrast to previous approaches, e.g., [CW86, LCKK00, Rak07, Rak08]).

Using an initial marking can be useful, e.g., in debugging. Consider for instance that the user is analyzing a particular trace for a marked Petri net (using a simulation tool [Dat], which we assume correct), so that an erroneous state is reached. Here, by *erroneous* state, we mean a marking in which some places have an incorrect number of tokens. In this case, we are interested in extracting the set of places and transitions (more formally, a subnet) that may erroneously contribute tokens to the places of interest, so that the user can more easily locate the bug.

Therefore, our first notion of *slicing criterion* is formalized as follows:

Definition 4.1.1 *(Slicing Criterion) Let $\mathcal{N} = (P, T, F)$ be a Petri net. A* slicing criterion *for $\mathcal{N}$ is a pair $\langle M_0, Q\rangle$ where M_0 is an initial marking for $\mathcal{N}$ and $Q \subseteq P$ is a set of places.*

Roughly speaking, given a slicing criterion $\langle M_0, Q\rangle$ for a Petri net $\mathcal{N}$, we are interested in extracting a subnet with those places and transitions of $\mathcal{N}$ which can contribute to change the marking of Q in any execution starting in M_0.

Our notion of *dynamic* slice is defined as follows. In the following, we say that σ' is a *subsequence* of a firing sequence σ w.r.t. a set of transitions T if σ' contains all transitions of σ that belong to T and in the same order.

Definition 4.1.2 *(Slice) Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. Given a Petri net $\mathcal{N}' = (P', T', F')$, we say that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, Q\rangle$ if the following conditions hold:*

- *the Petri net $\mathcal{N}'$ is a subnet of $\mathcal{N}$ and*
- *for each firing sequence $\sigma = t_1 \ldots t_n$, for $\mathcal{N}$, with $M_0 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n$ such that $M_{n-1}(p) < M_n(p)$ for some $p \in Q$, there exists a firing sequence σ' for $(\mathcal{N}', M'_0)$, with $M'_0 = M_0|_{P'}$, such that*
  - *σ' is a subsequence of σ w.r.t. T',*
  - *$M'_0 \xrightarrow{\sigma'} M'_m$, $m \leq n$, and*
  - *M'_m covers $M_n|_{P'}$ (i.e., $M'_m \geq M_n|_{P'}$).*

Intuitively speaking, a Petri net $\mathcal{N}'$ is a slice of another Petri net $\mathcal{N}$ if $\mathcal{N}'$ is a subnet of $\mathcal{N}$ (i.e., no additional places nor transitions are added) and the behaviour of $\mathcal{N}$ is preserved in $\mathcal{N}'$ for the restricted sets of places and transitions. In order to formalize this second condition, we require that, for all firing sequences $\sigma = t_1 \ldots t_n$ that may *move* tokens to the places of the slicing criterion, i.e.,

$$M_0 \xrightarrow{t_1} \ldots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n \text{ and } M_{n-1}(p) < M_n(p),\ p \in Q$$

the *restriction* of this firing sequence can also be performed on the slice $\mathcal{N}'$, i.e.,

$$M'_0 \xrightarrow{\sigma'} M'_m \text{ and } M'_m \geq M_n$$

Trivially, given a Petri net $\mathcal{N}$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 4.1.3 *Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. First, we compute a* backward slice *similar to that of [Rak07]. This is obtained from* $\mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\})$*, where function* $\mathsf{b\_slice}_{\mathcal{N}}$ *is defined as follows:*

$$\mathsf{b\_slice}_{\mathcal{N}}(W, W_{done}) = \begin{cases} \{\,\} \quad \mathit{if}\ W = \{\,\} \\ T \cup {}^{\bullet}T \cup \mathsf{b\_slice}_{\mathcal{N}}(W \setminus W'_{done}, W'_{done}) \\ \qquad \mathit{if}\ W \neq \{\,\},\ \mathit{where}\ T = {}^{\bullet}p, \\ \qquad\qquad\qquad \mathit{and}\ W'_{done} = W_{done} \cup \{p\} \\ \qquad\qquad\qquad \mathit{for\ some}\ p \in P \end{cases}$$

Now, we compute a forward slice *from*

$$\mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

where function $\mathsf{f\_slice}_{\mathcal{N}}$ *is defined as follows:*

$$\mathsf{f\_slice}_{\mathcal{N}}(W, R, V) = \begin{cases} W \cup R \quad \textit{if } V = \{\,\} \\ \mathsf{f\_slice}_{\mathcal{N}}(W \cup V^{\bullet}, R \cup V, V') \\ \qquad \textit{if } V \neq \{\,\}, \\ \qquad \textit{where } V' = \{t \in T \setminus (R \cup V) \mid {}^{\bullet}t \subseteq W \cup V^{\bullet}\} \end{cases}$$

Then, the dynamic slice is finally obtained from the intersection of the backward and forward slices. Formally, let

$$P' \cup T' = \mathsf{b\_slice}_{\mathcal{N}}(Q, \{\,\}) \cap \\ \mathsf{f\_slice}_{\mathcal{N}}(\{p \in P \mid M_0(p) > 0\}, \{\,\}, \{t \in T \mid M_0 \xrightarrow{t}\})$$

with $P' \subseteq P$ *and* $T' \subseteq T$*, the computed slice is*

$$\mathcal{N}' = (P', T', F|_{(P',T')})$$

Algorithm 4.1.3 describes our method to extract a dynamic slice from a Petri net. Intuitively speaking, Algorithm 4.1.3 constructs the slice of a Petri net (P, T, F) for a set of places $Q \subseteq P$ as follows. The key idea is to capture a possible token flow relevant for places in Q. For this purpose,

- we first compute the possible paths which lead to the slicing criterion,
- then we also compute the paths that may be followed by the tokens of the initial marking.

This can be done by taking into account that (i) the marking of a place p depends on its input and output transitions, (ii) a transition may only be fired if it is enabled, and (iii) the enabling of a transition depends on the marking of its input places. The algorithm is divided in three steps:

- The first step is a backward slicing method (which is similar to the *basic slicing algorithm* of [Rak07]) that obtains a slice $\mathcal{N}_1 = (P_1, T_1, F_1)$ defined as the subnet of $\mathcal{N}$ that includes all input places of all transitions connected to any place p in P_1, starting with $Q \subseteq P_1$.
  - The core of this method is the auxiliary function $\mathsf{b\_slice}_{\mathcal{N}}$, which is initially called with the set of places Q of the slicing criterion together with an empty set of places.
  - For a particular non-empty set of places W and a particular place $p \in W$, function $\mathsf{b\_slice}_{\mathcal{N}}$ returns the transitions T in ${}^\bullet p$ and the input places of these transitions ${}^\bullet T$. Then, function $\mathsf{b\_slice}_{\mathcal{N}}$ moves backwards adding the place p to the set W_{done} and removing from W the updated set W_{done} until the set W becomes empty.
- The second step is a forward slicing method that obtains a slice $\mathcal{N}_2 = (P_2, T_2, F_2)$ defined as the subnet of $\mathcal{N}$ that includes all transitions initially enabled in M_0 as well as those transitions connected as output transitions of places in P_2, starting with $p \in P$ such that $M_0(p) > 0$.
  - We define an auxiliary function $\mathsf{f\_slice}_{\mathcal{N}}$, which is initially called with the places that are marked at M_0, an empty set of transitions and the enabled transitions in M_0.
  - For a particular set of places W, a particular set of transitions R and a particular non-empty set of transitions V, function $\mathsf{f\_slice}_{\mathcal{N}}$ moves forwards adding the places in $V^\bullet$ to W, adding the transitions in V to R and replacing the set of transitions V by a new set V' in which are included the transitions that are not in $R \cup V$ and whose input places are in $W \cup V^\bullet$.
  - Finally, when V is empty, function $\mathsf{f\_slice}_{\mathcal{N}}$ returns the accumulated set of places and transitions $W \cup R$.

- Finally, the third step obtains the slice $\mathcal{N}' = (P', T', F')$ defined as the subnet of $\mathcal{N}$ where P' is the intersection of P_1 and P_2, T' is the intersection of T_1 and T_2, and F' is the restriction of F over P' and T', i.e., the intersection of backward and forward slices.

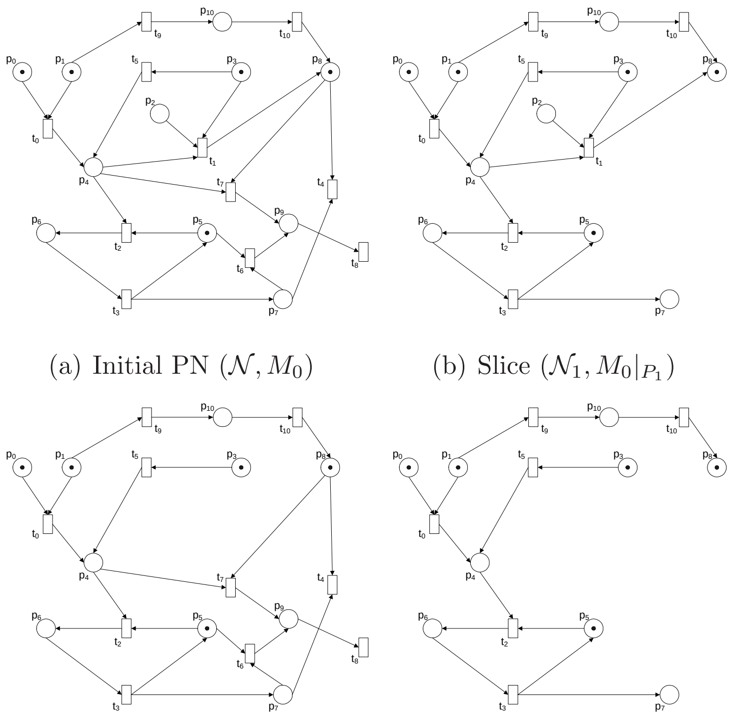
The following result states the completeness of our algorithm for computing Petri net slices. The proof of this result follows easily by induction on the length of the firing sequences considered in Definition 4.1.2.

Theorem 4.1.4 *(Correctness) Let $\mathcal{N}$ be a Petri net and $\langle M_0, Q\rangle$ be a slicing criterion for $\mathcal{N}$. The dynamic slice $\mathcal{N}'$ computed in Algorithm 4.1.3 is a correct slice according to Definition 4.1.2.*

We will now show the usefulness of the technique with a simple example.

Example 4.1.5 *Consider the Petri net $\mathcal{N}$ of Fig. 4.1(a) where the user wants to produce a slice w.r.t. the slicing criterion $\langle M_0, \{p_5, p_7, p_8\}\rangle$. Figure 4.1(b) shows the slice $\mathcal{N}_1$ obtained in the first part of Algorithm 4.1.3. Figure 4.1(c) shows the slice $\mathcal{N}_2$ obtained in the second part of Algorithm 4.1.3. The subnet shown in Fig. 4.1(d) is the final result of Algorithm 4.1.3 (the intersection of $\mathcal{N}_1$ and $\mathcal{N}_2$). This slice contains all the places and transitions of the original Petri net which can transmit tokens to the slicing criterion.*

Clearly, using an initial marking allows us to produce smaller slices. Surprisingly, previous approaches completely ignored the marking of the net, and thus their slices are often rather big. For instance, the slice of Fig. 4.1(b) is a subset of the slice produced by Rakow's algorithm [Rak07] (this algorithm would also include transitions t_4, t_6 and t_7). Clearly, this slice contains parts of the Petri net that cannot be reached with the given initial marking (e.g., transition t_1 which could never be fired because place p_2 is empty). Rakow's algorithm computes all the parts of the Petri net which could transmit tokens to the slicing criterion and, thus, the associated slicing criterion is just $\langle Q\rangle$,



(a) Initial PN $(\mathcal{N}, M_0)$

(b) Slice $(\mathcal{N}_1, M_0|_{P_1})$

(c) Slice $(\mathcal{N}_2, M_0|_{P_2})$

(d) Slice result of Algorithm 4.1.3

Figure 4.1: Example of an application of Algorithm 4.1.3

where $Q \subseteq P$ is a set of places. In contrast, we compute all the parts of the Petri net which could transmit tokens to the slicing criterion from the initial marking. Therefore, our technique is essentially a generalization of Rakow's technique because the slice produced with Rakow's algorithm w.r.t. $\langle Q \rangle$ is the same as the slice produced w.r.t. $\langle M_0, Q \rangle$ if $M_0(p) > 0$ for all $p \in P$ and all $t \in T$ are enabled transitions at M_0.

Our slicing technique is more general than Rakow's technique but, at the same time, it keeps its simplicity and efficiency because we still use the Petri net structure to produce the slice. Therefore, our first approach can be considered *lightweight* because its cost is bounded by the number of transitions T of the original Petri net; namely, the cost of our algorithm is $\mathcal{O}(2T)$.

4.2 Extracting Slices from Traces

In this section, we present an alternative approach to dynamic slicing that generally produces smaller slices by also considering a particular firing sequence.

In principle, Algorithm 4.1.3 should consider all possible executions of the Petri net starting from the initial marking. This approach can be useful in some contexts but it is too imprecise for debugging when a particular simulation has been performed. Therefore, in our second approach, we refine the notion of slicing criterion so as to also include the firing sequence that represents the erroneous simulation. By exploting this additional information, the new slicing algorithm will usually produce smaller slices. Formally,

Definition 4.2.1 *(Slicing Criterion) Let $\mathcal{N} = (P, T, F)$ be a Petri net. A* slicing criterion *for $\mathcal{N}$ is a triple $\langle M_0, \sigma, Q \rangle$ where M_0 is a marking for $\mathcal{N}$, σ is an initial firing sequence (i.e., starting from M_0) and $Q \subseteq P$ is a set of places.*

Roughly speaking, given a slicing criterion $\langle M_0, \sigma, Q\rangle$ for a Petri net, we are interested in extracting a subnet with those places and transitions which are necessary to move tokens to the places in Q.

Our notion of *dynamic* slice is defined as follows:

Definition 4.2.2 *(Dynamic Slice) Let $\mathcal{N} = (P, T, F)$ be a Petri net. Let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$, with $\sigma = t_1 t_2 \ldots t_n$. Given a Petri net $\mathcal{N}' = (P', T', F')$, we say that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$ if the following conditions hold:*

- *the Petri net $\mathcal{N}'$ is a subnet of $\mathcal{N}$,*
- *the set of places Q appears in P' (i.e., $Q \subseteq P'$), and*
- *there exists a firing sequence σ' for $(\mathcal{N}', M'_0)$, with $M'_0 = M_0|_{P'}$, such that*
  - *σ' is a subsequence of σ w.r.t. T',*
  - *$M'_0 \xrightarrow{\sigma'} M'_m$, $m \leq n$, and*
  - *M'_m covers $M_n|_{P'}$ (i.e., $M'_m \geq M_n|_{P'}$).*

Trivially, given a marked Petri net $(\mathcal{N}, M_0)$, the complete net $\mathcal{N}$ is always a correct slice w.r.t. any slicing criterion. The challenge then is to produce a slice as small as possible.

Algorithm 4.2.3 *Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$, with $\sigma = t_1 t_2 \ldots t_n$. Then, we compute a dynamic slice $\mathcal{N}'$ of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$ as follows:*

- *We have $\mathcal{N}' = (P', T', F')$, where $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$, $P' \cup T' = \mathsf{slice}(M_n, \sigma, Q)$, $P' \subseteq P$, $T' \subseteq T$, and $F' = F|_{(P',T')}$. Auxiliary function* slice *is defined as follows:*

$$\mathsf{slice}(M_i, \sigma, W) = \begin{cases} W & \\ \qquad \textit{if } i = 0 & \\ \mathsf{slice}(M_{i-1}, \sigma, W) & \\ \qquad \textit{if } \forall p \in W.\ M_{i-1}(p) \geq M_i(p),\ i > 0 & \\ \{t_i\} \ \cup\ \mathsf{slice}(M_{i-1}, \sigma, W \cup {}^\bullet t_i) & \\ \qquad \textit{if } \exists p \in W.\ M_{i-1}(p) < M_i(p),\ i > 0 & \end{cases}$$

- *The initial marking M'_0 is the restriction of M_0 over P', i.e., $M'_0 = M_0|_{P'}$.*

Intuitively speaking, given a slicing criterion $\langle M_0, \sigma, Q\rangle$, the slicing algorithm proceeds as follows:

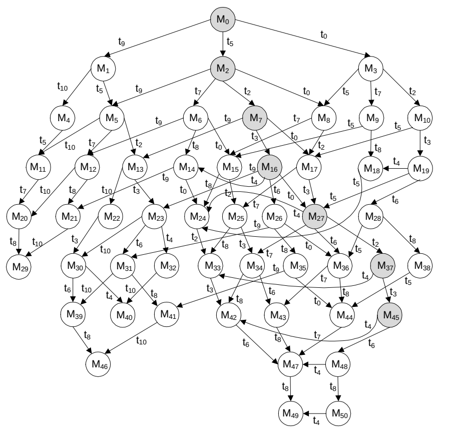
- The core of the algorithm lies in the auxiliary function slice, which is initially called with the marking M_n which is reachable from M_0 through σ, together with the firing sequence σ and the set of places Q of the slicing criterion.

- For a particular marking M_i, $i > 0$, a firing sequence σ and a set of places W, function slice just moves "backwards" when no place in W increased its tokens by the considered firing.

- Otherwise, the fired transition t_i increased the number of tokens of some place in W. In this case, function slice already returns this transition t_i and, moreover, it moves backwards also adding the places in ${}^\bullet t_i$ to the previous set W.

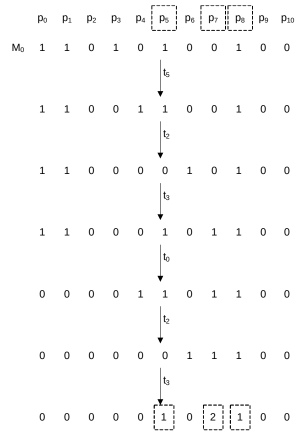
- Finally, when the initial marking is reached, function slice returns the accumulated set of places (which includes the initial places in Q).

We will now show the utility of the technique with a simple example.

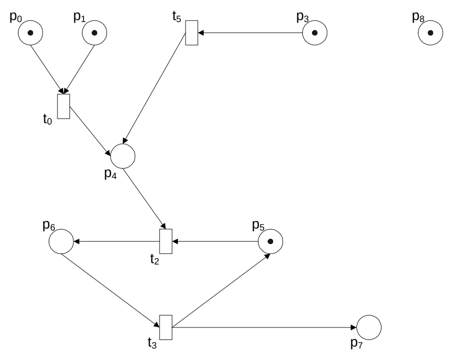


M_0	1 1 0 1 0 1 0 0 1 0 0	M_{17}	0 0 0 0 1 0 1 0 1 0 0	M_{34}	0 0 0 0 0 1 0 1 0 1 0
M_1	1 0 0 1 0 1 0 0 1 0 1	M_{18}	0 0 0 1 0 1 0 0 0 0 0	M_{35}	1 1 0 0 0 0 0 0 1 0 0
M_2	1 1 0 0 1 1 0 0 1 0 0	M_{19}	0 0 0 1 0 1 0 1 1 0 0	M_{36}	0 0 0 0 1 0 0 0 1 1 0
M_3	0 0 0 1 1 1 0 0 1 0 0	M_{20}	1 0 0 0 0 1 0 0 1 1 0	M_{37}	0 0 0 0 0 0 1 1 1 0 0
M_4	1 0 0 1 0 1 0 0 2 0 0	M_{21}	1 0 0 0 0 1 0 0 0 0 1	M_{38}	0 0 0 1 0 0 0 0 1 0 0
M_5	1 0 0 0 1 1 0 0 1 0 1	M_{22}	1 0 0 0 0 0 1 0 2 0 0	M_{39}	1 0 0 0 0 0 0 0 2 1 0
M_6	1 1 0 0 0 1 0 0 0 1 0	M_{23}	1 0 0 0 0 1 0 1 1 0 1	M_{40}	1 0 0 0 0 1 0 0 1 0 0
M_7	1 1 0 0 0 0 1 0 1 0 0	M_{24}	0 0 0 0 1 1 0 0 0 0 0	M_{41}	1 0 0 0 0 0 0 0 1 0 1
M_8	0 0 0 0 2 1 0 0 1 0 0	M_{25}	0 0 0 0 0 0 1 0 0 1 0	M_{42}	0 0 0 0 0 1 0 1 0 0 0
M_9	0 0 0 1 0 1 0 0 0 1 0	M_{26}	1 1 0 0 0 0 0 0 1 1 0	M_{43}	0 0 0 0 0 0 0 0 0 2 0
M_{10}	0 0 0 1 0 0 1 0 1 0 0	M_{27}	0 0 0 0 1 1 0 1 1 0 0	M_{44}	0 0 0 0 1 0 0 0 1 0 0
M_{11}	1 0 0 0 1 1 0 0 2 0 0	M_{28}	0 0 0 1 0 0 0 0 1 1 0	M_{45}	0 0 0 0 0 1 0 2 1 0 0
M_{12}	1 0 0 0 0 1 0 0 0 1 1	M_{29}	1 0 0 0 0 1 0 0 1 0 0	M_{46}	1 0 0 0 0 0 0 0 2 0 0
M_{13}	1 0 0 0 0 0 1 0 1 0 1	M_{30}	1 0 0 0 0 1 0 1 2 0 0	M_{47}	0 0 0 0 0 0 0 0 0 1 0
M_{14}	1 1 0 0 0 1 0 0 0 0 0	M_{31}	1 0 0 0 0 0 0 0 1 1 1	M_{48}	0 0 0 0 0 0 0 1 1 1 0
M_{15}	0 0 0 0 1 1 0 0 0 1 0	M_{32}	1 0 0 0 0 1 0 0 0 0 1	M_{49}	0 0 0 0 0 0 0 0 0 0 0
M_{16}	1 1 0 0 0 1 0 1 1 0 0	M_{33}	0 0 0 0 0 0 1 0 0 0 0	M_{50}	0 0 0 0 0 0 0 1 1 0 0

(a) Reachability graph



(b) Firing sequence σ



(c) Slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$

Figure 4.2: Example of an application of Algorithm 4.2.3

Example 4.2.4 *Consider the Petri net $\mathcal{N}$ of Example 4.1.5 shown in Fig. 4.1(a), together with the firing sequence σ shown in Fig. 4.2(b). The firing sequence $\sigma = t_5t_2t_3t_0t_2t_3$ corresponds to the branch of the reachability graph shown in Fig. 4.2(a) that goes from the root to the node M_{45}. Then, the user can define the slicing criterion $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$ for $\mathcal{N}$; where M_0 is the initial marking for $\mathcal{N}$ defined in Fig 4.1(a).*

Clearly, this slicing criterion focus on a particular execution and thus the slice produced is more precise than the one produced by Algorithm 4.1.3. In *this case, the slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, \{p_5, p_7, p_8\}\rangle$ is the Petri net shown in Fig. 4.2(c).*

The following result states the completeness of our algorithm for computing Petri net slices.

Theorem 4.2.5 *(Correctness) Let $\mathcal{N} = (P, T, F)$ be a Petri net and let $\langle M_0, \sigma, Q\rangle$ be a slicing criterion for $\mathcal{N}$. The dynamic slice $\mathcal{N}'$ computed in Algorithm 4.2.3 is a correct slice according to Definition 4.2.2.*

Proof. We prove the claim by induction on the number n of transitions in σ.

If $n = 0$, then $\mathsf{slice}(M_0, \sigma, Q) = \bigcup_{p \in Q} \mathsf{slice}(M_0, \sigma, \{p\}) = Q$ and the claim follows trivially for $\mathcal{N}' = (Q, \{\}, \{\})$ and $M_0' = M_0|_Q$.

If $n > 0$, then we distinguish two cases:

- If $M_{n-1}(p) \geq M_n(p)$ for all $p \in Q$, then $\mathsf{slice}(M_n, \sigma, Q) = \mathsf{slice}(M_{n-1}, \sigma, Q)$ and the claim follows by induction.
- Otherwise, there exists some $p \in Q$ with $M_{n-1}(p) < M_n(p)$ and, therefore, $\mathsf{slice}(M_n, \sigma, Q) = \{t_n\} \cup \mathsf{slice}(M_{n-1}, \sigma, Q \cup {}^\bullet t_n)$. Let $\mathcal{N}' = (P', T', F')$, $F' = F|_{(P',T')}$, and $M_0' = M_0|_{P'}$. Now, we prove that $\mathcal{N}'$ is a slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma, Q\rangle$:
  - Trivially, $\mathcal{N}'$ is a subnet of $\mathcal{N}$, M_0' is a restriction of M_0 and $Q \subseteq P'$.
  - Let $\mathcal{N}''$ be the slice of $\mathcal{N}$ w.r.t. $\langle M_0, \sigma_{n-1}, Q \cup {}^\bullet t_n\rangle$, with $\sigma_{n-1} = M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots M_{n-1}$ and $\mathcal{N}'' = (P'', T'', F'')$.

    By the inductive hypothesis, there exists a firing sequence σ'' for $(\mathcal{N}'', M_0'')$, with $M_0'' = M_0|_{P''}$, such that

    * σ'' is a subsequence of σ_{n-1} w.r.t. T'',
    * $M_0'' \xrightarrow{\sigma''} M_k''$, $k \leq n-1$, and

 * M''_k covers M_{n-1} (i.e., $M''_k \geq M_{n-1}$).

Now, we consider a firing sequence σ' for $(\mathcal{N}', M'_0)$ that mimicks σ'' (which is safe since $P'' = P'$ and $T'' \subseteq T'$) and then adds one more firing depending on whether $t_n \in T'$ or not. If $\sigma' = \sigma''$ then the claim follows by induction. Otherwise, it follows trivially by the inductive hypothesis and the fact that M''_k covers M_n.

Chapter 5

Conclusions

This thesis has presented new slicing techniques for two different languages, CSP and Petri Nets. Both languages provide a means for modelling the behavior of concurrent systems. This made these slice techniques very useful to reduce the size of some specifications in order to help program understanding, debugging,etc.

As for CSP we define two new static analyses that can be applied to languages with explicit synchronizations such as CSP. In particular, we introduce a method to slice CSP specifications, in such a way that, given a CSP specification and a slicing criterion we produce a slice that (i) is a subset of the specification (i.e., it is produced by deleting some parts of the original specification); (ii) contains all the parts of the specification which must be executed (in any execution) before the slicing criterion (MEB analysis); and (iii) can also produce an augmented slice which also contains those parts of the specification that could be executed before the slicing criterion (CEB analysis).

We have presented two algorithms to compute the MEB and CEB analyses which are based on a new data structure, the CSCFG, that has shown to be more precise than the previously used SCFG. The advantage of the CSCFG is that it cares about contexts, and it is able to distinguish between different contexts in which a process is called.

We have built a prototype which implements all the data structures and algorithms defined in chapter 3; and we have integrated it into the system ProB. Preliminary experiments has demonstrated the usefulness of the technique.

As for future work, we are interested in adapt these analyses to full CSP, with communication between events, processes with parameters, etc. Other interesting works may rely on using CSCFGs for slice other languages similar to CSP. Finally, we plan to improve the algorithm that calculates synchronizations of CSCFG, in order to reduce their number, reducing by this way the size of the slices.

As for Petri Nets we have introduced two different techniques for dynamic slicing of Petri nets. To the best of our knowledge, this is the first approach to dynamic slicing for Petri nets. The first approach takes into account the Petri net and an initial marking, but produces a slice w.r.t. any possibly firing sequence. The second approach further reduces the computed slice by fixing a particular firing sequence. In general, our slices are smaller than previous (static) approaches where no initial marking nor firing sequence were considered.

As for future work, we plan to carry on an experimental evaluation of our slicing techniques in order to test its viability in practice. We also find it useful to extend our slicing technique to other kind of Petri nets (e.g., coloured Petri nets [Jen97] and marked-controlled reconfigurable nets [LO04]).

Bibliography

[BH05] A. Bell and B.R. Haverkort. Sequential and distributed model checking of Petri nets. *Int. Journal on Software Tools for Technology Transfer*, 7(1):43–60, 2005.

[BG96] D. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996.

[BL05] M. Butler and M. Leuschel. Combining CSP and B for specification and property verification. In *Proceedings of Formal Methods 2005*, LNCS 3582, pages 221–236, Newcastle upon Tyne, 2005. Springer-Verlag.

[Brü04] I. Brückner. Slicing CSP-OZ Specifications. In P. Pettersson and W. Yi, editors, *Proc. of the 16th Nordic Workshop on Programming Theory*, number 2004-041 in Technical Reports of the Department of Information Technology, pages 71–73. Uppsala University, Sweden, 2004.

[BW05] I. Brückner and H. Wehrheim. Slicing Object-Z Specifications for Verification. In *Proc. of the 4th Int'l Conf. of B and Z Users (ZB 2005)*, pages 414–433. Springer LNCS 3455, 2005.

[CW86] C.K. Chang and H. Wang. A Slicing Algorithm of Concurrency Modeling Based on Petri Nets. In *Proc. of the Int'l*

Conf. on Parallel Processing (ICPP'86), pages 789–792. IEEE Computer Society Press, 1986.

[CR94] J. Chang and D. Richardson. Static and dynamic specification slicing. In *Proc. of the Fourth Irvine Software Symposium*. Irvine, CA, 1994.

[CS88] D. Callahan and J. Sublok. Static analysis of low-level synchronization. In *In proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD'88)*, pages 100–111, New York, NY, USA, 1988. ACM.

[CGP00] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press: Cambridge, MA, 2000.

[Dat] Petri Nets Tool Database. Available at `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html`.

[DE95] J. Desel and J. Esparza. *Free choice Petri nets*. Cambridge University Press, New York, NY, USA, 1995.

[FOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[HRS98] M.J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *International Symposium on Software Testing and Analysis*, pages 11–20, 1998.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[HW97] M.P.E. Heimdahl and M.W. Whalen. Reduction and Slicing of Hierarchical State Machines. In M. Jazayeri and H. Schauer, editors, *Proc. of the 6th European Software Engineering Conference (ESEC/FSE'97)*, pages 450–467. Springer LNCS 1301, 1997.

[Jen97] K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts, 1992. Volume 2: Analysis Methods, 1994. Volume 3: Practical Use, 1997. Monographs in Theoretical Computer Science, Springer-Verlag.

[KSS95] K. M. Kavi, F. T. Sheldon, and B. Shirazi. Reliability analysis of CSP specifications using petri nets and markov processes. In *Proc. 28th Annual Hawaii Int. Conf. on System Sciences;: Software Technology, 3-6 January 1995, Wailea, HI*, volume 2, pages 516–524, 1995.

[Kri98] J. Krinke. Static slicing of threaded programs. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 35–42, 1998.

[Kri03] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proc. FSE/ESEC*, pages 178–187, 2003.

[LS95] P. Ladkin and B. Simons. Static deadlock analysis for csp-type communications, 1995.

[LCKK00] W.J. Lee, S.D. Cha, Y.R. Kwon, and H.N. Kim. A Slicing-based Approach to Enhance Petri Net Reachability Analysis. *Journal of Research and Practice in Information Technology*, 32(2):131–143, 2000.

[LB08] M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[LF08] Michael Leuschel and Marc Fontaine. Probing the depths of CSP-M: A new FDR-compliant validation tool. In *Proceedings ICFEM 2008*, pages 278–297. LNCS. Springer LNCS, 2008.

[LLOST08] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. Static Slicing of CSP Specifications. *Proc. of the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'08)*, pages 141–150, 2008.

[LLOST08b] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. The MEB and CEB Static Analysis for CSP Specifications. Logic-Based Program Synthesis and Transformation (revised and selected papers from LOPSTR 2008). Springer LNCS. To appear.

[LLOST09] M. Leuschel, M. Llorens, J. Oliver, J. Silva, and S. Tamarit. SOC: a Slicer for CSP Specifications. Proc. of the ACM SIGPLAN 2009 Workshop on Partial Evaluation and Program Manipulation (PEPM'09). To appear.

[LO04] M. Llorens and J. Oliver. Introducing Structural Dynamic Changes in Petri Nets: Marked-Controlled Reconfigurable Nets. In Farn Wang, editor, *Proc. of the 2nd Int'l Conf. on Automated Technology for Verification and Analysis (ATVA'04)*, pages 310–323. Springer LNCS 3299, 2004.

[LOSTV08] M. Llorens and J. Oliver and J. Silva and S. Tamarit and G. Vidal. Dynamic Slicing Techniques for Petri Nets. Proc. of the 2nd Workshop on Reachability Problems (RP 2008), To appear in Electronic Notes in Theoretical Computer Science.

[Mur89] T. Murata. Petri Nets: Properties, Analysis and Applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[NA98] G. Naumovich and G.S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. *SIGSOFT Softw. Eng. Notes*, 23(6):24–34, 1998.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Rak07] A. Rakow. Slicing Petri Nets. Technical report, Department für Informatik, Carl von Ossietzky Universität, Oldenburg, 2007.

[Rak08] A. Rakow. Slicing Petri Nets with an Application to Workflow Verification. In *Proc. of the 34th Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, pages 436–447. Springer LNCS 4910, 2008.

[Rau90] M. Rauhamaa. *A Comparative Study of Methods for Efficient Reachability Analysis.* Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, 1990.

[RGGHJS95] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *First International Workshop Tools and Algorithms for Construction and Analysis of Systems (TACAS '95).*

[SH96] A.M. Sloane and J. Holdsworth. Beyond traditional program slicing. In *Proc. of the Int'l Symp. on Software Testing and Analysis*, pages 180–186, San Diego, CA, 1996. ACM Press.

[Tip95] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[Wei79] M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, 1979.

[Wei84] M.D. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.