A Static Analysis of Value Passing CCS with Application to Workflows

Sergiu Zavrotschi

Kongens Lyngby 2007

Introduction

The growth of web services in recent years attracts a lot of attention to their interaction, choreography and orchestration. A number of technologies have been developed for defining, executing and managing workflows targeted at web services. Examples of such technologies include the Business Process Execution Language (BPEL)[16] and the Windows Workflow Foundation (WWF)[17]. However, technologies used for implementing workflows focus mainly on their functional aspects; there is a clearly identified need for formal foundations as well as techniques and tools for analysing their quantitative and qualitative properties.

We will look into expressing of workflows in process calculi and design analysis of one of calculi, namely value passing CCS, with application to workflows expressed with the help of this calculus.

Workflow is the movement of information, documents and task through the work processes. In order to analyse workflows, they should be represented in a calculus which possesses a number of properties. This calculus should support parallelism, synchronization between processes, exchanging data between processes and manipulation with data.

It is natural to draw attention to process calculi that implement formal modelling of concurrent systems. The most popular and widely used calculi include CSP[5], PEPA[4], Orc[9], CCS and π-calculus[8]. In the last years it has been shown that both CCS and π-calculus are capable of representing workflows and services ([3], [11], [13]). This fact as well as strong mathematical foundation of CCS led to its choice as an underlying language for implementation of analysis of workflows and services.

In recent paper "A Monotone Framework for CCS"[10] it was proposed a static analysis that approximates the control structure of concurrent systems models in CCS. In this thesis we will take a monotone framework as a starting point and extend it in order to meet requirements imposed by its application to workflow modelling.

The goal of the analysis introduced in [10] is to construct a finite automaton that approximates the behavior of processes. But that analysis does not pay attention to the communication of information over channels of the processes. We will improve the analysis so that it takes into account variables of the CCS program and creates a more precise automaton. In order to be able to modify analysis we need to extend the original CCS language. The result of the analysis of value passing CCS will consist in automaton represented as a graph, that describes the bahaviour of the CCS program. A number of analyses of the resulting graph will be also made.

Chapter 1 introduces syntax and semantics of CCS with a number of additional syntactic constructs intended to pass values between processes, what is aimed to better mapping of workflows. In chapters 2–4 present sets of exposed, generated and killed actions from "A Monotone Framework for CCS" that are needed for the developed analysis. Chapter 5 describes the compatible actions extended to handle the modified syntax of CCS. Chapter 6 introduces a notion of free names, that will later be used for determination of possible values of variables. Chapter 7 is devoted to the propagation of values through the CCS model. It describes a worklist algorithm created to capture the approximation of sets of values for all variables at different points the CCS program. Chapter 8 the automaton constructed by the monotone framework is extended in order to include the information about variables. The resulting automaton may become rather big, that's why chapter 8 describes also how the graph corresponding to the finite automaton may be modified in order to express not the behaviour of the whole system but of its part. In chapter 9 some analyses of the obtained automaton are shown. Chapter 10 describes the creation of a GUI tool that implements the whole process of editing, creation of automaton and its analysis. Chapter 11 gives a number of more complex examples of workflows and services expressed in CCS and analysed using the developed technique. Chapter 12 contains some concluding remarks and directions of future work.

The reader of this thesis is expected to be familiar with the basic principles of process calculi, what is equivalent to reading of Chapters 3–4 of Milner's "Communicating and Mobile Systems: The π-calculus"[8], and with main ideas of program analysis described in Chapters 1–2 of "Principles of Program Analyses"[2].

Contents

Chapter 1

Calculus of Communicating Systems

We have chosen to use Calculus of Communicating Systems (CCS) with value passing, that permits its application to model workflows. In this section syntax and semantics of CCS will be introduced and issues concerning implementation will be described.

1.1 Syntax

The main paradigm of the calculus is process P.

$$P \quad ::= \quad P_1|P_2 \mid \sum_{i\in I} \alpha_i^{l_i}.P_i \mid \texttt{new}\ x\ P \mid A \mid \emptyset$$

Composition $P_1|P_2$ is used to model concurrent execution of P_1 and P_2. Summation $\sum_{i\in I} \alpha_i^{l_i}.P_i$ is the exclusive choice among the finite number of guarded processes $\alpha_i.P_i$, where action α_i is called a guard of process P_i. $\texttt{new}\ x\ P$ restricts the scope of the name x to process P. A is a process identifier, defined with a equation of the form $A \triangleq P$. $\emptyset$ is inaction.

Processes in sums are guarded by actions of the form:

$$\begin{aligned} \alpha &::= \overline{x}\langle v_1, \ldots, v_k \rangle \mid x(z_1, \ldots, z_k) \mid \tau \mid \gamma \\ \gamma &::= [v = x] \mid [v \neq x] \end{aligned}$$

The action of the form $\overline{x}\langle v_1, \ldots, v_k \rangle$ sends the names $v_1, \ldots, v_k$ over the channel x.

The action of the form $x(z_1, \ldots, z_k)$ receives names $z_1, \ldots, z_k$ over the channel x. If this action guards process P, then after its execution model continues as P with $z_1, \ldots, z_k$ replaced by the received names (see Semantics).

The unobservable action is written by τ. It is used to model the internal action of the process.

There are two types of match actions γ: $[v = x]$ and $[v \neq x]$. $[v = x]$ checks values of v and x for equality and continues execution of the process guarded by it, only if v is equal to x. $[v \neq x]$ does the reverse: the execution of the process guarded by it continues only if v and x are different.

We will analyse programs of the form

$$\begin{aligned} &\texttt{let} \quad A_1 \triangleq P_1;\ \ldots;\ A_k \triangleq P_k; \\ &\texttt{in} \quad P_0 \end{aligned}$$

where P_0 is the main process of the program and $A_1 \triangleq P_1;\ \ldots;\ A_k \triangleq P_k;$ are definitions of the subprocesses with identifiers $A_1, \ldots A_k$. Identifiers can be used anywhere in the program, they must not duplicate.

1.2 Semantics

Semantics of CCS is based on a reduction relation $\rightarrow$ presented in Figure 1.1.

Equation (1.1) shows that internal action τ will always occur.

Equation (1.2) states that reaction between an action and its complement will always occur, if there is such a possibility. Process P_2 continues with $z_1, \ldots, z_k$ having values $v_1, \ldots, v_k$, respectively. Relations (1.3) and (1.4) show that whenever a matching action $[v_1 = v_2]^l P + Q \rightarrow_l P$ is true the program may continue

$$\tau^{l}.\, P + Q \rightarrow_{l} P \tag{1.1}$$

$$(\overline{x}\langle v_1, \ldots, v_k\rangle^{l_1}.\, P_1 + Q_1) \mid (x(z_1, \ldots, z_k)^{l_2}.\, P_2 + Q_2) \rightarrow_{l_2 l_1} P_1 \mid P_2\{\,{}^{v_1,\ldots,v_k}/_{z_1,\ldots,z_k}\} \tag{1.2}$$

$$[v_1 = v_2]^{l} P + Q \rightarrow_{l} P \text{ if } v_1 = v_2 \tag{1.3}$$

$$[v_1 = v_2]^{l} P + Q \rightarrow_{l} Q \text{ if } v_1 \neq v_2 \tag{1.4}$$

$$\frac{P \rightarrow_{\bar{l}} Q}{P \mid P' \rightarrow_{\bar{l}} Q \mid P'} \tag{1.5}$$

$$\frac{P \rightarrow_{\bar{l}} P'}{\texttt{new}\ x\ P \rightarrow_{\bar{l}} \texttt{new}\ x\ P'} \tag{1.6}$$

$$\frac{P' \rightarrow_{\bar{l}} Q'}{P \rightarrow_{\bar{l}} Q} \text{ if } P \equiv P' \text{ and } Q' \equiv Q \tag{1.7}$$

Figure 1.1: Reduction relation $\rightarrow$ for CCS

as P. Equations (1.5) and (1.6) show that reaction can occur inside a parallel composition or restriction. Equation (1.7) indicates that structural congruence described in [8] may be used.

Example 1.1 Let us consider a system with a server and a client working in parallel.

They communicate via two channels. Client receives two values x and y via these channels and checks them for equality. If they are equal, client sends response to the server and continues from the beginning. If they are different, client sends another response and terminates. Server sends either two different values or two identical values, waits for the response from the client and repeats this procedure forever.

This example may be written in CCS with value passing as:

```
let
```
$$Client \triangleq p(x,z)^{1}.q(y)^{2}.([x=y]^{3}\overline{r}\langle x\rangle^{4}.Client + [x\neq y]^{5}\overline{r}\langle z\rangle^{6}.0);$$
$$Server \triangleq \overline{p}\langle a,c\rangle^{7}.\overline{q}\langle a\rangle^{8}.r(w)^{9}.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle b\rangle^{11}.r(w)^{12}.Server;$$
```
in
```
$$Client \mid Server$$

There are two possible ways of reduction of the model:

$$
\begin{aligned}
Client \mid Server \quad &\equiv \quad p(x,z)^1.q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad (\overline{p}\langle a,c\rangle^7.\overline{q}\langle a\rangle^8.r(w)^9.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle b\rangle^{11}.r(w)^{12}. \\
&\quad Server) \\
&\rightarrow_{1\ 7} \quad q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad \overline{q}\langle a\rangle^8.r(w)^9.Server \\
&\rightarrow_{2\ 8} \quad ([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad r(w)^9.Server \\
&\rightarrow_{3} \quad \overline{r}\langle x\rangle^4.Client \mid .r(w)^9.Server \\
&\rightarrow_{4\ 9} \quad Client \mid Server
\end{aligned}
$$

and

$$
\begin{aligned}
Client \mid Server \quad &\equiv \quad p(x,z)^1.q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad (\overline{p}\langle a,c\rangle^7.\overline{q}\langle a\rangle^8.r(w)^9.Server + \\
&\quad \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle b\rangle^{11}.r(w)^{12}.Server) \\
&\rightarrow_{1\ 10} \quad q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad \overline{q}\langle b\rangle^{11}.r(w)^{12}.Server \\
&\rightarrow_{2\ 11} \quad ([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0) \mid \\
&\quad r(w)^{12}.Server \\
&\rightarrow_{5} \quad \overline{r}\langle z\rangle^6.0 \mid r(w)^{12}.Server \\
&\rightarrow_{6\ 12} \quad Server
\end{aligned}
$$

After the first variant of reduction system returns to its initial state and continues execution from the beginning. The second variant leads to termination of *Client* process, *Server* process remains active. ■

1.3 Implementation

First, it is important to define a programming language for CCS. Then we need a system that converts program written in this programming language to the internal representation suitable for analysis. It was chosen to write a lexical analyser using *JLex: A Lexical Analyzer Generator for Java* (described in [1])

and parser with the help of *CUP: LALR Parser Generator for Java* (described in [12]).

The scheme of generation of syntax tree from the source code is shown in Figure 1.2.

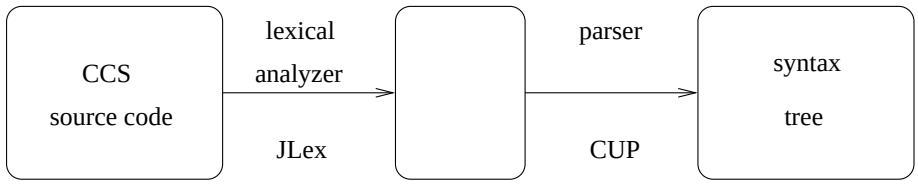


Figure 1.2: Syntax tree generation

Definition of the syntax of CCS with value passing used for building of parser is presented in the Appendix A.1.

Program is labelled during parsing. Labels are assigned incrementally beginning with $''1''$. Internal actions (τ), send and receive actions and match actions are annotated with unique labels $l \in \mathbf{Lab}$. Comment lines begin with symbol #.

Example 1.2 *Client-Server program from Example 1.1 may be written as:*

```
let
  Client ::= (p(x, z).q(y).([x = y] r<x>.Client + [x != y] r<z>.0));
  Server ::= (p<a, c>.q<a>.r(w).Server + p<a, d>.q<b>.r(w).Server);
in
  Client | Server
```

■

The next chapters will be devoted to the sets of actions that are used in the algorithm for the propagation of values through the CCS program and for the construction of the automaton.

Chapter 2

Exposed actions

Exposed actions are actions, that may participate in the next step of the process. Processes may contain several occurrences of the same action. Due to possibility of recursive definition of the processes there may be infinitely many exposed actions. For example, process $A \triangleq \overline{a}^1\langle x\rangle.0 \mid A$ contains infinite number of action $\overline{a}^1\langle x\rangle$, and all these occurrences are exposed.

In order to represent exposed actions we will introduce the notion of an *extended multiset* M as an element of:

$$\mathfrak{M} = \mathbf{Lab} \to \mathbb{N} \cup \{\infty\}$$

$M(l)$ is the number of occurrences of the label l. It may be either a natural number or infinity. The partial ordering $\leq_{\mathfrak{M}}$ is defined as:

$$M \leq_{\mathfrak{M}} M' \qquad \text{iff} \qquad \forall l : M(l) \leq M'(l) \vee M'(l) = \infty$$

The domain $(\mathfrak{M}, \leq_{\mathfrak{M}})$ is a complete lattice with leas element $\bot_{\mathfrak{M}}$, given by $\forall l : \bot_{\mathfrak{M}}(l) = 0$ and largest element $\top_{\mathfrak{M}}$ given by $\forall l : \top_{\mathfrak{M}}(l) = \infty$. For calculation

of the exposed actions we need to define operation $+_{\mathfrak{M}}$, addition operation on extended multisets:

$$(M +_{\mathfrak{M}} M')(l) = \begin{cases} M(l) + M'(l) & \text{if } M(l) \in \mathbb{N} \wedge M'(l) \in \mathbb{N} \\ \infty & \text{otherwise} \end{cases}$$

Given a process

$$\texttt{let } A_1 \triangleq P_1; \ldots; A_k \triangleq P_k \texttt{ in } P_0$$

our object of interest is the exposed function $\mathcal{E}_\star$ that maps process P_0 to the extended multiset, corresponding to actions that are exposed in the initial process:

$$\mathcal{E}_\star : \mathbf{Proc} \to \mathfrak{M}$$

As P_0 uses definitions of the processes $A_1, \ldots, A_k$, we need to introduce the environment $env_{\mathcal{E}}$ that maps process names $A_1, \ldots, A_k$ to the extended multisets $(\mathbf{PN} \to \mathfrak{M})$. We can now define $\mathcal{E}_\star[\![P]\!] = \mathcal{E}[\![P]\!]env_{\mathcal{E}}$, where

$$\mathcal{E} : \mathbf{Proc} \to (\mathbf{PN} \to \mathfrak{M}) \to \mathfrak{M}$$

$\mathcal{E}$ takes two parameters – process P and environment $env_{\mathcal{E}}$. We define a functional $\mathcal{F}_{\mathcal{E}}(env) = [A_1 \mapsto \mathcal{E}[\![P_1]\!], \ldots, A_k \mapsto \mathcal{E}[\![P_k]\!]env]$ and initial environment $env_{\perp_{\mathfrak{M}}} = [A_1 \mapsto \perp_{\mathfrak{M}}, \ldots, A_k \mapsto \perp_{\mathfrak{M}}]$ that maps process names to the empty multisets. We can now define $env_{\mathcal{E}} = \sqcup_{j \geq 0} \mathcal{F}^j_{\mathcal{E}}(env_{\perp_{\mathfrak{M}}})$, where j is the number of unfoldings of functional $\mathcal{F}_{\mathcal{E}}$.

Now we may introduce equations for calculation of the exposed actions for different syntactic entities of CCS. They are presented in the Figure 2.1:

$$\begin{aligned} \mathcal{E}[\![\texttt{new } x\ P]\!]env &= \mathcal{E}[\![P]\!]env \\ \mathcal{E}[\![P \mid P']\!]env &= \mathcal{E}[\![P]\!] +_{\mathfrak{M}} \mathcal{E}[\![P']\!]env \\ \mathcal{E}[\![A]\!]env &= env(A) \\ \mathcal{E}[\![\sum_{i \in I} \alpha_i^{l_i}.P_i]\!]env &= \sum_{i \in I}{}_{\mathfrak{M}} \perp_{\mathfrak{M}}[l_i \mapsto 1] \end{aligned}$$

Figure 2.1: Exposed actions

Exposed actions of the `new` construct are equal to the exposed actions of the process P it is applied to. Exposed actions of the parallel composition $P \mid P'$ are equal to the sum of the exposed actions of two subproceses P and P'. As for the exposed actions of a sum, they are equal to the sum of multisets that have form $\bot_{\mathfrak{M}}[l_i \mapsto 1]$, because in the i-th component of the sum only one action with label i_i is exposed. To calculate exposed actions of a process name, the environment needs to be queried.

2.1 Implementation

Extended multiset M, $\mathfrak{M}$ and $\mathfrak{T}$ were implemented as a number of classes. Class `M` contains all operation over the elements of extended multisets ($+_{\mathfrak{M}}$, $-_{\mathfrak{M}}$, $\geq_{\mathfrak{M}}$, $\sqcup_{\mathfrak{M}}$, $\sqcap_{\mathfrak{M}}$, $\bowtie_{\mathfrak{M}}$, $\triangledown_{\mathfrak{M}}$). Classes `Multiset` and `Multisets` contain pointwise extensions of these action to the level of $\mathfrak{M}$ and $\mathfrak{T}$.

After parsing of CCS source code we received syntax tree with nodes being objects of classes corresponding to all syntactic categories of the language. Each class has a function `Multiset exp (int num)` used for recursive calculation of function $\mathcal{E}$, described in the previous section.

The initial environment $env_{\bot_{\mathfrak{M}}}$ is calculated at the parse time. To ensure termination $env_{\mathcal{E}}$ is calculated according to the formula:

$$env_{\mathcal{E}} = \mathcal{F}_{\mathcal{E}}^{k}(env_{\bot_{\mathfrak{M}}}) \bowtie \mathcal{F}_{\mathcal{E}}^{2k}(env_{\bot_{\mathfrak{M}}})$$

where k is the number of definitions in the program and $\bowtie$ is the pointwise extension of the operation $\bowtie_{\mathfrak{M}}$ defined by

$$(M \bowtie_{\mathfrak{M}} M')(l) = \begin{cases} M(l) & if M(l) = M'(l) \\ \infty & \text{otherwise} \end{cases}$$

This formula ensures that we take into consideration the effect from unfoldings of the recursively defined k processes. We use only operation $+_{\mathfrak{M}}$ in calculation of $\mathcal{E}$ in the Figure 2.1, thus numbers in the extended multisets may only grow. If they are equal after k and $2k$ unfoldings, then operation $\bowtie_{\mathfrak{M}}$ returns the same number. Otherwise it is obvious, that they will grow indefinitely and $\bowtie_{\mathfrak{M}}$ returns ∞.

Using $env_{\mathcal{E}}$ we easily calculate exposed actions $\mathcal{E}_\star$ for the whole program and save it in a table in class `Env` for the later use.

Example 2.1 For the program from Example 1.1 we have:

$$\begin{aligned} env_{\mathcal{E}} &= [Client \mapsto \bot_{\mathfrak{M}}[1 \mapsto 1], \\ &\quad\ Server \mapsto \bot_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]] \\ \mathcal{E}_\star[\![Client \mid Server]\!] &= \bot_{\mathfrak{M}}[1 \mapsto 1, 7 \mapsto 1, 10 \mapsto 1] \end{aligned}$$

■

Chapter 3

Generated actions

For construction of automaton with the help of worklist algorithm we need to introduce *generated actions* and *killed actions*. This chapter will describe how generated actions are defined and calculated and the next chapter will do the same for the killed actions.

Generated actions are such actions, that become exposed after executing an action. It will always be safe to generate more actions, than will be actually generated at the run-time, therefore we will calculate an over-approximation.

Function $\mathcal{G}_\star$ that approximates information about generated actions will work with elements of:

$$\mathfrak{T} = \mathbf{Lab} \to \mathfrak{M}$$

that maps labels of the program of interest to the extended multisets $\mathfrak{M}$. We will define ordering $\leq_{\mathfrak{T}}$ on $\mathfrak{T}$ as the pointwise extension of the ordering $\leq_{\mathfrak{M}}$, described in Chapter 2. The domain $(\mathfrak{T}, \leq_{\mathfrak{T}})$ is a complete lattice and operator $\sqcup_{\mathfrak{T}}$ is defined as the pointwise extension of the operation $\sqcup_{\mathfrak{M}}$ defined as:

$$(M \sqcup_{\mathfrak{M}} M')(l) = \begin{cases} \max\{M(l), M'(l)\} & \text{if } M(l) \in (N) \wedge M'(l) \in \mathbb{N} \\ \infty & \text{otherwise} \end{cases}$$

In analogy with exposed actions we are interested in function $\mathcal{G}_\star$ that maps

process P_0 of the program of interest to the function $\mathfrak{T}$:

$$\mathcal{G}_\star : \mathbf{Proc} \to \mathfrak{T}$$

$\mathcal{G}_\star$ may be defined as $\mathcal{G}_\star[\![P]\!] = \mathcal{G}[\![P]\!]env_\mathcal{G}$, where env_G is the environment that maps process names $A_1, \ldots, A_k$ to $\mathfrak{T}$ and $\mathcal{G}[\![P]\!]$ is defined as:

$$\mathcal{G} : \mathbf{Proc} \to (\mathbf{PN} \to \mathfrak{T}) \to \mathfrak{T}$$

To calculate $env_\mathcal{G}$ a monotonic functional $\mathcal{F}_\mathcal{G}(env) = [A_1 \mapsto \mathcal{G}[\![P_1]\!], \ldots, A_k \mapsto \mathcal{G}[\![P_k]\!]env]$ and empty environment $env_{\perp_\mathfrak{T}} = [A_1 \mapsto \perp_\mathfrak{T}, \ldots, A_k \mapsto \perp_\mathfrak{T}]$ are needed. Thus, $env_\mathcal{G} = \sqcup_{j \geq 0} \mathcal{F}^j_\mathcal{G}(env_{\perp_\mathcal{G}})$, where j is the number of unfoldings of functional $\mathcal{F}_\mathcal{G}$.

Functional $\mathcal{F}_\mathcal{G}$ uses $\mathcal{G}[\![P]\!]env$ that needs to be defined for all syntactic entities of CCS. This is shown in Figure 3.1.

$$\begin{aligned}
\mathcal{G}[\![\texttt{new } x\ P]\!]env &= \mathcal{G}[\![P]\!]env \\
\mathcal{G}[\![P \mid P']\!]env &= \mathcal{G}[\![P]\!] \sqcup_\mathcal{G} \mathcal{G}[\![P']\!]env \\
\mathcal{G}[\![A]\!]env &= env(A) \\
\mathcal{G}[\![\sum_{i \in I} \alpha_i^{l_i}.P_i]\!]env &= \bigsqcup_{i \in I}{}_\mathfrak{T}(\perp_\mathfrak{T}[l_i \mapsto \mathcal{E}_\star[\![P_i]\!]] \sqcup_\mathfrak{T} \mathcal{G}[\![P_i]\!]env)
\end{aligned}$$

Figure 3.1: Generated actions

Generated actions of the `new` construct are equal to the generated actions of the process P it is applied to. Generated actions of the parallel composition $P \mid P'$ are equal to the least upper bound $\sqcup_\mathfrak{T}$ of the generated actions of two subproceses P and P', where $\sqcup_\mathfrak{T}$ is the pointwise extension of the operation $\sqcup_\mathfrak{M}$ defined earlier. To calculate generated actions of a process name, the environment needs to be queried. Generated actions of a sum are equal to the least upper bound $\sqcup_{\mathfrak{T}_{i \in I}}$ of all components of the sum. All components of summation in CCS have form $\alpha_i^{l_i}.P_i$. Generated actions for this case are defined as $\mathcal{G}[\![\alpha_i^{l_i}.P_i]\!]env = \perp_\mathfrak{T}[l_i \mapsto \mathcal{E}_\star[\![P_i]\!]] \sqcup_\mathfrak{T} \mathcal{G}[\![P_i]\!]env$. The usage of $\perp_\mathfrak{T}[l_i \mapsto \mathcal{E}_\star[\![P_i]\!]]$ is based on the fact that execution of action $\alpha_i^{l_i}$ makes actions from P_i exposed. The least upper bound with $\mathcal{G}[\![P_i]\!]env$ is needed to cover the situation when label l_i is used inside process P_i.

3.1 Implementation

The initial environment $env_{\perp_{\mathfrak{T}}}$ is initialised at the parse time. Calculation of $\mathcal{G}_\star$ may be invoked only after calculation of $\mathcal{E}_\star$, because $\mathcal{G}[\![\sum_{i\in I}\alpha_i^{l_i}.P_i]\!]env$ uses information from it.

k iterations are needed in order to calculate $env_{\mathcal{G}}$:

$$env_{\mathcal{G}} = \mathcal{F}_{\mathcal{G}}^{k}(env_{\perp_{\mathfrak{T}}})$$

where k is the number of recursively defined processes in the program of interest. k iterations are needed to make sure that all process names are unfolded at least once, and hence no additional information may be added by the operation $\sqcup_{\mathfrak{T}}$ used for the calculation of $\mathcal{G}$.

The calculation of generated actions is realised via recursive calls of function `Multisets gen (int num)` of the class `CcsTree` that implements syntax tree. The result is saved as table in class `Env`.

Example 3.1 For the program from Example 1.1 we have the following environment $env_{\mathcal{G}}$:

l	*Client*
1	$\perp_{\mathfrak{M}}[2 \mapsto 1]$
2	$\perp_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
3	$\perp_{\mathfrak{M}}[4 \mapsto 1]$
4	$\perp_{\mathfrak{M}}[1 \mapsto 1]$
5	$\perp_{\mathfrak{M}}[6 \mapsto 1]$
6	$\perp_{\mathfrak{M}}$

l	*Server*
7	$\perp_{\mathfrak{M}}[8 \mapsto 1]$
8	$\perp_{\mathfrak{M}}[9 \mapsto 1]$
9	$\perp_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
10	$\perp_{\mathfrak{M}}[11 \mapsto 1]$
11	$\perp_{\mathfrak{M}}[12 \mapsto 1]$
12	$\perp_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$

and $\mathcal{G}_\star$:

l	$\mathcal{G}_\star[\![Client \mid Server]\!]$
1	$\perp_{\mathfrak{M}}[2 \mapsto 1]$
2	$\perp_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
3	$\perp_{\mathfrak{M}}[4 \mapsto 1]$
4	$\perp_{\mathfrak{M}}[1 \mapsto 1]$
5	$\perp_{\mathfrak{M}}[6 \mapsto 1]$
6	$\perp_{\mathfrak{M}}$
7	$\perp_{\mathfrak{M}}[8 \mapsto 1]$
8	$\perp_{\mathfrak{M}}[9 \mapsto 1]$
9	$\perp_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
10	$\perp_{\mathfrak{M}}[11 \mapsto 1]$
11	$\perp_{\mathfrak{M}}[12 \mapsto 1]$
12	$\perp_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$

■

Chapter 4

Killed actions

Killed actions are such actions that were exposed before an action was executed and became not exposed after execution of this action. It will always be safe to kill fewer actions than will be actually killed at the run-time, therefore under-approximation of killed actions will be computed.

Function $\mathcal{K}_\star$ that approximates information about killed actions will like function $\mathcal{G}_\star$ work with elements of:

$$\mathfrak{T} = \mathbf{Lab} \to \mathfrak{M}$$

that maps labels of the program of interest to the extended multisets $\mathfrak{M}$. Need of under-approximation leads to the usage of the least upper bound operator $\sqcap_{\mathfrak{T}}$ over $\mathfrak{T}$ defined as the pointwise extension of the operator $\sqcap_{\mathfrak{M}}$:

$$(M \sqcap_{\mathfrak{M}} M')(l) = \begin{cases} \min\{M(l), M'(l)\} & \text{if } M(l) \in (N) \wedge M'(l) \in \mathbb{N} \\ M(l) & \text{if } M'(l) = \infty \\ M'(l) & \text{if } M(l) = \infty \end{cases}$$

We are again interested in function $\mathcal{K}_\star$ that maps process P_0 of the analysed program to the function $\mathfrak{T}$:

$$\mathcal{K}_\star : \mathbf{Proc} \to \mathfrak{T}$$

$\mathcal{K}_\star$ is be defined as $\mathcal{K}_\star[\![P]\!] = \mathcal{K}[\![P]\!]env_\mathcal{K}$, where $env_\mathcal{K}$ is the environment that maps process names $A_1, \ldots, A_k$ to $\mathfrak{T}$ and $\mathcal{K}[\![P]\!]$ is defined as:

$$\mathcal{K} : \mathbf{Proc} \to (\mathbf{PN} \to \mathfrak{T}) \to \mathfrak{T}$$

For calculation of $env_\mathcal{K}$ we define as monotonic functional $\mathcal{F}_\mathcal{K}(env) = [A_1 \mapsto \mathcal{K}[\![P_1]\!], \ldots, A_k \mapsto \mathcal{K}[\![P_k]\!]env]$ and empty environment
$env_{\top_\mathfrak{T}} = [A_1 \mapsto \top_\mathfrak{T}, \ \ldots, A_k \mapsto \top_\mathfrak{T}]$. Thus, $env_\mathcal{K} = \sqcap_{j \geq 0} \mathcal{F}^j_\mathcal{K}(env_{\top_\mathfrak{T}})$, where j is the number of unfoldings of functional $\mathcal{F}_\mathcal{K}$.

Functional $\mathcal{F}_\mathcal{K}$ uses $\mathcal{K}[\![P]\!]env$ that needs to be defined for all syntactic entities of CCS. This is shown in Figure 4.1.

$$\begin{array}{rcl}
\mathcal{K}[\![\texttt{new } x \ P]\!]env & = & \mathcal{K}[\![P]\!]env \\
\mathcal{K}[\![P \mid P']\!]env & = & \mathcal{K}[\![P]\!] \sqcap_\mathcal{K} \mathcal{K}[\![P']\!]env \\
\mathcal{K}[\![A]\!]env & = & env(A) \\
\mathcal{K}[\![\sum_{i \in I} \alpha_i^{l_i}.P_i]\!]env & = & \prod_{i \in I}{}_\mathfrak{T} (\top_\mathfrak{T}[l_i \mapsto M] \sqcap_\mathfrak{T} \mathcal{K}[\![P_i]\!]env) \\
 & & \text{where } M = +_{\mathfrak{M}_{j \in I}} \bot_\mathfrak{M}[l_j \mapsto 1]
\end{array}$$

Figure 4.1: Killed actions

Killed actions of the `new` construct are equal to the killed actions of the process P it is applied to. Killed actions of the parallel composition $P \mid P'$ are equal to the greatest lower bound $\sqcap_\mathfrak{T}$ of the killed actions of two subproceses P and P', where $\sqcap_\mathfrak{T}$ is the pointwise extension of the operation $\sqcap_\mathfrak{M}$ defined earlier. To calculate generated actions of a process name, the environment needs to be queried.

Killed actions of a sum are equal to the greatest lower bound $\sqcap_{\mathfrak{T}_{i \in I}}$ of all components of the sum that have form $\alpha_i^{l_i}.P_i$. After execution of one action from the sum, all actions of the sum that are exposed will be killed. Therefore each label l_i should be mapped to $M = +_{\mathfrak{M}_{j \in I}} \bot_\mathfrak{M}[l_j \mapsto 1]$. The greatest lower bound with $\mathcal{K}[\![P_i]\!]env$ is needed to cover the situation when label l_i occurs inside process P_i.

4.1 Implementation

The initial environment $env_{\top_\mathfrak{T}}$ is initialised at the parse time. Calculation of $\mathcal{K}_\star$ may be invoked at any time, because it does not use information from the

exposed actions $\mathcal{E}_\star$ or generated actions $\mathcal{G}_\star$.

The key operation used in calculation of $\mathcal{K}_\star$ i the greatest lower bound $\sqcap_{\mathfrak{T}}$ and the monotonic functional $\mathcal{F}_{\mathcal{K}}$ on a complete lattice $(\mathfrak{T}, \leq_{\mathfrak{T}})$, that leads to termination after some number of unfoldings of $\mathcal{F}_{\mathcal{K}}$.

The calculation of killed actions is realised via recursive calls of function `Multisets kill (int num)` of the class `CcsTree` that implements syntax tree. The result is saved as table in class `Env`.

Example 4.1 For the program from Example 1.1 we have the following environment $env_{\mathcal{K}}$:

l	*Client*
1	$\bot_{\mathfrak{M}}[1 \mapsto 1]$
2	$\bot_{\mathfrak{M}}[2 \mapsto 1]$
3	$\bot_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
4	$\bot_{\mathfrak{M}}[4 \mapsto 1]$
5	$\bot_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
6	$\bot_{\mathfrak{M}}[6 \mapsto 1]$

l	*Server*
7	$\bot_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
8	$\bot_{\mathfrak{M}}[8 \mapsto 1]$
9	$\bot_{\mathfrak{M}}[9 \mapsto 1]$
10	$\bot_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
11	$\bot_{\mathfrak{M}}[11 \mapsto 1]$
12	$\bot_{\mathfrak{M}}[12 \mapsto 1]$

and $\mathcal{K}_\star$:

l	$\mathcal{K}_\star[\![\mathit{Client} \mid \mathit{Server}]\!]$
1	$\bot_{\mathfrak{M}}[1 \mapsto 1]$
2	$\bot_{\mathfrak{M}}[2 \mapsto 1]$
3	$\bot_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
4	$\bot_{\mathfrak{M}}[4 \mapsto 1]$
5	$\bot_{\mathfrak{M}}[3 \mapsto 1, 5 \mapsto 1]$
6	$\bot_{\mathfrak{M}}[6 \mapsto 1]$
7	$\bot_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
8	$\bot_{\mathfrak{M}}[8 \mapsto 1]$
9	$\bot_{\mathfrak{M}}[9 \mapsto 1]$
10	$\bot_{\mathfrak{M}}[7 \mapsto 1, 10 \mapsto 1]$
11	$\bot_{\mathfrak{M}}[11 \mapsto 1]$
12	$\bot_{\mathfrak{M}}[12 \mapsto 1]$

■

Chapter 5

Compatible actions

Another important notion is the set of *compatible actions*. Compatible actions are the pairs of actions that may interact in parallel processes.

We introduce a new datatype — *tuple*, which will be used by the function that computes the set of compatible actions for the program of interest. The tuple has form (L, C) and contains two components:

- $L \in \wp(\mathbf{Lab})$ is the set of labels of all actions of the process
- $C \in \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})$ is the set consisting of the pairs of labels of actions that may interact and labels that may be executed without interaction with other labels. There are two types of such actions:
  - τ - internal action of the process,
  - and $\gamma ::= [v = x] \mid [v \neq x]$ — match actions that check whether two names are equal or not.

Function $\mathcal{C}_\star$ will return such tuple and it may be defined as:

$$\mathcal{C}_\star : \mathbf{Proc} \to \wp(\mathbf{Lab}) \times \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})$$

In analogy with generated and killed actions $\mathcal{C}_\star$ may be defined as $\mathcal{C}_\star[\![P]\!] = \mathcal{C}[\![P]\!]env_\mathcal{C}$, where env_C is the environment that maps process names $A_1, \ldots, A_k$ to $\mathfrak{T}$ and $\mathcal{C}[\![P]\!]$ is defined as:

$$\begin{aligned}\mathcal{C} \quad : \quad & \mathbf{Proc} \to (\mathbf{PN} \to \wp(\mathbf{Lab}) \times \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})) \\ & \to \wp(\mathbf{Lab}) \times \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})\end{aligned}$$

The domain used in this definition $(\wp(\mathbf{Lab}) \times \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab}))$ has partial ordering $\sqsubseteq$ because its components $\wp(\mathbf{Lab})$ and $\wp(\mathbf{Lab} \times \mathbf{Lab})$ also have this ordering and therefore is a complete lattice. We define functional $\mathcal{F}_C(env) = [A_1 \mapsto \mathcal{C}[\![P_1]\!], \ldots, A_k \mapsto \mathcal{C}[\![P_k]\!]env]$ and empty environment $env_\emptyset = [A_1 \mapsto (\emptyset, \emptyset), \ldots, A_k \mapsto (\emptyset, \emptyset)]$. Thus, $env_\mathcal{C} = \sqcup_{j \geq 0} \mathcal{F}_\mathcal{C}^j(env_\emptyset)$, where j is the number of unfoldings of monotonic functional $\mathcal{F}_\mathcal{C}$.

Function $\mathcal{C}[\![P]\!]env$ is defined for all syntactic categories of CCS in Figure 5.1.

$$\begin{aligned}\mathcal{C}[\![\texttt{new } x\ P]\!]env &= \mathcal{C}[\![P]\!]env \\ \mathcal{C}[\![P \mid P']\!]env &= \texttt{let } (L, C) = \mathcal{C}[\![P]\!] \ \&\& \ (L', C') = \mathcal{C}[\![P']\!]env \\ &\quad \texttt{in } (L \cup L', C \cup C' \cup \texttt{comp}(L, L')) \\ \mathcal{C}[\![A]\!]env &= env(A) \\ \mathcal{C}[\![\sum_{i \in I} \alpha_i^{l_i}.P_i]\!]env &= \texttt{let } (L_i, C_i) = \mathcal{C}[\![P_i]\!]env \\ &\quad \texttt{in } (\bigcup_{i \in I} L_i \cup \{l_i\}, \bigcup_{i \in I} \mathcal{C}_i)\end{aligned}$$

Figure 5.1: Compatible actions

For calculation of compatible actions of parallel composition $P \mid P'$ compatible actions of its two subprocesses are computed. The first component of the resulting tuple is set to the union of L and L', meaning that the set of labels of the composition is equal to the union of all labels in its subprocesses. The second element of the resulting tuple is set to the union of pairs of labels for actions that may interact inside the subprocesses (C and C') and pairs of labels for potential interacting actions, one of which is in P and another in P' ($\texttt{comp}(L, L')$).

Function $\texttt{comp}(L, L')$ may be defined with the help of *canonical name* associated

with each label l ($\partial(l)$) that is preserved by alpha-renaming.

$$\begin{aligned}\texttt{comp}(L, L') &= \{(l, l') \in L \times L' | \exists x : \partial(l) = \lfloor \overline{x} \rfloor \wedge \partial(l') = \lfloor x \rfloor\} \\ &\cup \{(l', l) \in L' \times L | \exists x : \partial(l') = \lfloor \overline{x} \rfloor \wedge \partial(l) = \lfloor x \rfloor\} \\ &\cup \{l \in L \cup L' | \exists \tau^l\} \\ &\cup \{l \in L \cup L' | \exists \gamma^l\}\end{aligned}$$

Each pair from the set returned by function `comp` has a label associated with a "send" action as the first and a label associated with a "receive" action as the second element.

As for the sums that have components in the form $\alpha_i^{l_i}.P_i$, the first element of the tuple returned by $\mathcal{C}$ is the union of labels L_i from P_i with labels l_i of actions $\alpha_i^{l_i}$. The second element is the union of C_i from P_i, because actions $\alpha_i^{l_i}$ cannot interact with each other.

5.1 Implementation

The initial environment $env_\emptyset$ is initialised at the parse time. Calculation of $\mathcal{C}_\star$ may be invoked at any time, because it doesn't use information from $\mathcal{E}_\star$ or any other function introduced before.

In analogy with env_G only k iterations are needed in order to calculate $env_\mathcal{C}$:

$$env_\mathcal{C} = \mathcal{F}_\mathcal{C}^k(env_\emptyset)$$

where k is the number of recursively defined processes in the program of interest. k iterations are needed to make sure that all processes are recursively unfolded at least once, and hence no additional information may be added.

The calculation of killed actions is realised via recursive calls of function `CompSet kill (int num)` of the class `CcsTree` that implements syntax tree. `CompSet` class is the realisation of tuple (L, C). The result is saved as table in class `Env`.

Example 5.1 For the program from Example 1.1 we have:

$$\begin{aligned}env_\mathcal{C} &= [Client \mapsto (\{1,2,3,4,5,6\}, \{3,5\}), \\ &\quad Server \mapsto (\{7,8,9,10,11,12\}, \emptyset)] \\ \mathcal{C}_\star &= (\{1,2,3,4,5,6,7,8,9,10,11,12\}, \\ &\quad \{(3),(5),(7,1),(10,1),(8,2),(11,2),(4,9),(4,12),(6,9),(6,12)\})\end{aligned}$$

■

Chapter 6

Free names

For taking care of values passing we need to to divide names used in the program into variables and constants. Variables are the names, that are bound to some values during the execution of the program – *bound names*. Constants are the names, that don't change – *free names*.

Given a program:

```
let
```
$$A_1 \triangleq P_1;\ \ldots;\ A_k \triangleq P_k;$$
```
in
```
$$P_0$$

we define function $\mathcal{FN}$, that returns free names, for all syntactic constructs of the CCS language described in Chapter 1.1. This function along with function `bn` for bound names is shown in Figure 6.1.

Free names of the parallel composition is equal to the union of free names of parallel processes. Free names of summation is equal to the union of sum components. Free names of a guarded process $\alpha.P$ is equal to the free names of process P without names, that are bound in α and including free names of α. For free names of a process definition environment is queried. Free names of sending action are equal to the names of variables, that are sent, but set of bound names is empty. As for the receiving action, set of free names is empty,

$$
\begin{aligned}
\mathcal{FN}[\![P \mid P']\!]env &= \mathcal{FN}[\![P]\!]env \cup \mathcal{FN}[\![P']\!]env \\
\mathcal{FN}[\![\sum_{i\in I} \alpha_i^{l_i}.P_i]\!]env &= \bigcup_{i\in I} \mathcal{FN}[\![\alpha_i^{l_i}.P]\!]env \\
\mathcal{FN}[\![\alpha.P]\!]env &= \mathcal{FN}[\![P]\!]env \setminus \mathtt{bn}(\alpha) \cup \mathcal{FN}[\![\alpha]\!]env \\
\mathcal{FN}[\![A_i]\!]env &= env(A_i) \\
\mathcal{FN}[\![\mathtt{new}\ x\ P]\!]env &= \mathcal{FN}[\![P]\!]env \setminus \{x\} \\
\mathcal{FN}[\![\overline{x}\langle v_1, \ldots, v_k\rangle]\!]env &= \{v_1, \ldots, v_k\} \\
\mathtt{bn}(\overline{x}\langle v_1, \ldots, v_k\rangle) &= \emptyset \\
\mathcal{FN}[\![x(z_1, \ldots, z_k)]\!]env &= \emptyset \\
\mathtt{bn}(x(z_1, \ldots, z_k)) &= \{z_1, \ldots, z_k\} \\
\mathcal{FN}[\![\tau]\!]env &= \emptyset \\
\mathtt{bn}(\tau) &= \emptyset \\
\mathcal{FN}[\![[v = x]]\!]env &= \{v, z\} \\
\mathtt{bn}([v = x]) &= \emptyset \\
\mathcal{FN}[\![[v \neq x]]\!]env &= \{v, z\} \\
\mathtt{bn}([v \neq x]) &= \emptyset
\end{aligned}
$$

Figure 6.1: Free names

but bound names are all the variables that are assigned in this action. Both free and bound names of an internal action τ are $\emptyset$. Free names of two matching constructs $[v = x]$ and $[v \neq x]$ are equal to the variable names that participate in these actions, and set of bound names is empty.

Our object of interest is the free names function $\mathcal{FN}_\star$ that maps process P_0 to the set of names that are free in this process:

$$\mathcal{FN}_\star : \mathbf{Proc} \to \mathfrak{N}$$

P_0 uses recursively defined process names $A_1, \ldots, A_k$. We introduce environment $env_{\mathcal{FN}}$ that maps process names $A_1, \ldots, A_k$ to the set of names ($\mathbf{PN} \to \mathfrak{N}$). Now it is possible to define $\mathcal{FN}_\star[\![P]\!] = \mathcal{FN}[\![P]\!]env_{\mathcal{FN}}$, where

$$\mathcal{FN} : \mathbf{Proc} \to (\mathbf{PN} \to \mathfrak{N}) \to \mathfrak{N}$$

$\mathcal{FN}$ takes two parameters – process P and environment $env_{\mathcal{FN}}$. We define a functional $\mathcal{F}_{\mathcal{FN}}(env) = [A_1 \mapsto \mathcal{FN}[\![P_1]\!], \ldots, A_k \mapsto \mathcal{FN}[\![P_k]\!]env]$ and initial

environment $env_{\perp_{\mathcal{FN}}} = [A_1 \mapsto \emptyset, \ldots, A_k \mapsto \emptyset]$ that maps process names to the empty sets of free names. We can now define $env_{\mathcal{FN}} = \sqcup_{j \geq 0} \mathcal{F}^j_{\mathcal{FN}}(env_{\perp_{\mathcal{FN}}})$, where j is the number of unfoldings of functional $\mathcal{F}_{\mathcal{FN}}$.

6.1 Implementation

The key operations over the set of names are union ($\cup$) and subtraction ($\setminus$) leading to the fact that recursion needs to be unfolded only once. Additional unfolding will have no effect. We need at most k iterations to calculate $env_{\mathcal{FN}}$:

$$env_{\mathcal{FN}} = \mathcal{F}^k_{\mathcal{FN}}(env_{\perp_{\mathfrak{N}}})$$

where k is the number of recursively defined processes.

The calculation of free names is realised via recursive calls of function `Vector <String> kill (int num)` of the class `CcsTree` that implements syntax tree. `Vector <String>` contains free names. The result is saved as table in class `Env`.

Example 6.1 For the program from Example 1.1 we have the following environment $env_{\mathcal{FN}}$:

$$env_{\mathcal{FN}} = [Client \mapsto \emptyset, Server \mapsto \{a, b, c, d\}]$$

and free names for the whole program:

$$\mathcal{FN}_\star = \{a, b, c, d\}$$

■

Chapter 7

Propagation of values

7.1 Introduction

The idea is to extend a finite automaton, that captures the control structure of a CCS model, in such a way that it will be capable to analyse values passed along the channels. This is essential for models of workflows, when it is important that processes exchange information and their actions depend on the information received.

In value passing CCS variables may be sent along channel x (i. e. $\overline{x}\langle v_1, \ldots, v_k\rangle$) or received along channel x (i. e. $x(v_1, \ldots, v_k)$). Variables may be later compared using constructs $[v_i = v_j]$ and $[v_i \neq v_j]$. We will construct an algorithm that for each label of the CCS model will determine a set of possible values for variables. Using this knowledge it will be possible to detect whether comparisons $[v_i = v_j]$ and $[v_i \neq v_j]$ are true or false. The procedure `enabled` of the automaton will then detect that a comparison is always false and can never be executed. Therefore all actions of the process after it become unreachable. Thus the number of enabled actions will be reduced. With fewer enabled actions automaton will become simpler, its number of states will be reduced. The automaton will also be more precise.

For the implementation of propagation of values we will construct an algorithm

based on worklist.

7.2 The worklist algorithm

Given a program

$$\texttt{let}\ \ A_1 \triangleq P_1; \ldots; A_k \triangleq P_k\ \texttt{in}\ \ P_0$$

and functions

$$\mathcal{E}_\star : \mathbf{Proc} \to \mathfrak{M}, \qquad \mathcal{G}_\star : \mathbf{Proc} \to (\mathbf{Lab} \to \mathfrak{M}) \qquad \mathcal{FN}_\star : \mathbf{Proc} \to \mathfrak{N}$$

$$\mathcal{C}_\star : \mathbf{Proc} \to \wp(\mathbf{Lab}) \times (\wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab}))$$

defined in the previous chapters, we can create a graph with the following components:

- A set of nodes $\mathtt{Q}$, with a node for each label $l \in \mathbf{Lab}$.
- A transition relation δ containing transitions of the form $l_s \Rightarrow l_t$ reflecting that the label l_t is generated by the label l_s, i. e. $l_t \in \mathcal{G}_\star(l_s)$.
- An additional node $l_0 \in \mathtt{Q}$ and a number of transitions of the form $l_0 \Rightarrow l_i$, where states l_i are the labels of the exposed actions from the function $\mathcal{E}_\star[\![P_0]\!]$.

The main data structures of the algorithm are:

- A set of labels $\mathtt{Q}$ and transaction relation δ introduced so far;
- a worklist $\mathtt{W}$ being a subset of $\mathtt{Q}$ containing those labels, that have yet to be processed;
- two functions V_{in} and V_{out} for each label. V_{in} maps each variable of the program to a set of possible values that it may take before execution of the action. V_{out} does the same thing, but after the execution of the action. $V_{in} : \mathbf{Lab} \to (\mathbf{Var} \to \mathcal{P}(\mathbf{Const}))$. Functions V_{in} and V_{out} were inspired by the *entry* and *exit* sets used for Data Flow Analysis described in [2].

All names of the program are divided into variables (**Var**) and constants (**Const**). This division is made with the help of *free names* described in Chapter 6. Constants are all names, that are elements of $\mathcal{FN}_\star$ and variables are all other names of the program. As we use CCS and not its more general form π-calculus that

permits exchange of channel names between processes, function $\mathcal{FN}_\star$ and therefore **Const** does not contain names of channels.

The overall algorithm is shown in the Figure 7.1.

```
(01) W = {l_i | l_i ∈ E_⋆};
(02) while W ≠ ∅ do
(03)     select l_s from W; W := W \ {l_s};
(04)     V_tmp = ∅
(05)     for each e_{l_f ⇒ l_s} do
(06)         V_tmp := V_tmp ∪ V_out(l_f)
(07)     V_in(l_s) := V_tmp
(08)     V_out(l_s) := V_in(l_s)
(09)     V_tmp := ∅
(10)     for each (l, l') ∈ C_⋆ do
(11)         if l' = l_s do
(12)                 V_tmp := V_tmp ∪ V_in(l)
(13)         if l = l_s do
(14)             W := W ∩ l'
(15)     replace(V_out(l_s), V_tmp)
(16)     for each e_{l_s ⇒ l_t} do
(17)         if V_in(l_t) ≠ V_out(l_s) do
(18)             W := W ∩ l_t
```

Figure 7.1: Worklist algorithm

The worklist algorithm begins with initialisation. First the graph Q is created as described before. Then the worklist is initialised to contain the initial labels l_i (line (01)). For the additional label l_0 in $V_{out}(0)$ all variables are mapped to $\top$ element of $\mathcal{P}(\mathbf{Const})$. All constants are mapped to themselves.

Algorithm proceeds while the worklist is not empty (line (02)).

A label l_s is selected and removed from the worklist in line (03). Lines (04-07) contain calculation of the union of $V_{out}(l_f)$ for all labels l_f, that are connected with the incoming edges to the current label l_s. This union is assigned to $V_{in}(l_s)$.

$V_{out}(l_s)$ of the current label l_s is assigned the value of $V_{in}(l_s)$ in line (08).

Cycle in lines (10-14) analyses all pairs of labels (l, l') that are compatible.

Lines (11-12) contain calculation of the possible values for variables updated in the action labeled with the current label. This is done by calculating a union of

$V_{in}(l)$ for all compatible actions of the form $(l, l') \in \mathcal{C}_\star$, where $l' = l_s$, i. e. for all "send" actions compatible with the current "receive" action.

Lines (13-14) update the worklist. If the current label is a label of the "send" action, then all the labels, that are compatible with it, are added to the worklist.

Procedure `replace` of line (15) consists of replacing of sets of possible values for variables in $V_{out}(l_s)$ by the sets from V_{tmp}. Sets of values for the variables of $V_{out}(l_s)$, that are not present in V_{tmp}, are left unchanged.

Lines (16-18) contain the update of the worklist. A label is added to the worklist, if it is a target label l_t of an edge, going out of the current label l_s and $V_{in}(t) \neq V_{out}(s)$, i. e. it is possible that new information will be added to the label.

7.3 Result

After execution of the worklist algorithm the function $V_{in} : \mathbf{Lab} \to (\mathbf{Var} \to \mathcal{P}(\{\mathbf{Const}\})$ which may be used in the `enabled` function of the worklist algorithm to construct a finite automaton.

7.4 Implementation

Algorithm for the propagation of values was implemented as an instance of class `ValList`, that fills tables for V_{in} and V_{out} in the class `Env`.

Example 7.1 Let us show how the worklist algorithm works for the Client-Server from Example 1.1.

`let`
$Client \triangleq p(x,z)^1.q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0);$
$Server \triangleq \overline{p}\langle a,c\rangle^7.\overline{q}\langle a\rangle^8.r(w)^9.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle b\rangle^{11}.r(w)^{12}.Server;$
`in`
$Client \mid Server$

$\mathcal{E}_\star$, $\mathcal{G}_\star$ and $\mathcal{C}_\star$ are calculated as shown in Examples 2.1, 3.1 and 5.1). Then graph based on the function $\mathcal{G}_\star$ is created with additional node "0" that is connected

to all nodes corresponding to labels from $\mathcal{E}_\star$. Graph is shown in Figure 7.2. Worklist is initialised with the same labels. $W = \{1, 6, 8\}$.

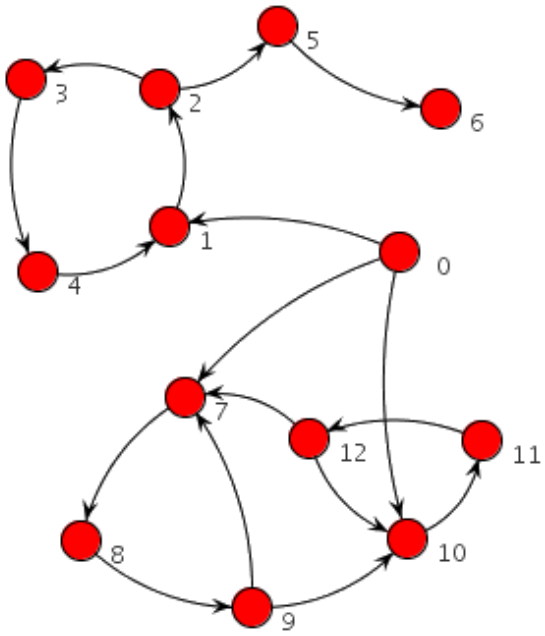


Figure 7.2: Graph based on function $\mathcal{G}_\star$

After execution of 26 rounds of the algorithm we receive tables for V_{in} and V_{out}, shown in Figure 7.3.

From these tables the points, where variables receive new values, may be detected. For variables x and z this is label 1, for variable y — label 2 and for w — labels 9 and 12. As the worklist algorithm shown in Figure 7.1 uses union in line (12), all possible values of variables are detected. For example, action with label 9 may interact with actions labelled by 4 and 6. Variable w may be assigned to x and z, which may take values of $\{a\}$ and $\{c, d\}$, correspondingly. Hence, $V_{out}(9)w = \{a, c, d\}$.

The usage of additional node $l_0 \in Q$ with all variables in $V_{out}(0)$ mapped to the $\top$ element of $\mathcal{P}(\mathbf{Const})$ ensures that recursion is handled correctly and that variables at the entry point of labels 1, 7 and 10 are still mapped to the $\top$ as the algorithm doesn't detect whether the recursion was already unfolded or not. ■

l	$V_{in}(l)$
1	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
2	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b,c,d\}]\!]$
3	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
4	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
5	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
6	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
7	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
8	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
9	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
10	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
11	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
12	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$

l	$V_{out}(l)$
1	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b,c,d\}]\!]$
2	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
3	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
4	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
5	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
6	$[\![x \mapsto \{a\}, w \mapsto \{a,b,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,b\}]\!]$
7	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
8	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
9	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
10	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
11	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,b,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$
12	$[\![x \mapsto \{a,b,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,b,c,d\}, y \mapsto \{a,b,c,d\}]\!]$

Figure 7.3: Tables for V_{in} and V_{out}

Example 7.2 Now we construct a modification of the Client-Server from Example 1.1 by changing constant b transmitted via channel p in the action with label "11" to a:

```
let
```
$$Client \triangleq p(x,z)^{1}.q(y)^{2}.([x=y]^{3}\overline{r}\langle x\rangle^{4}.Client + [x\neq y]^{5}\overline{r}\langle z\rangle^{6}.0);$$
$$Server \triangleq \overline{p}\langle a,c\rangle^{7}.\overline{q}\langle a\rangle^{8}.r(w)^{9}.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle a\rangle^{11}.r(w)^{12}.Server;$$
```
in
```
$$Client \mid Server$$

Functions $\mathcal{E}_\star$, $\mathcal{G}_\star$ and $\mathcal{C}_\star$ remain the same, but after execution of 23 rounds of the algorithm we receive new V_{in} and V_{out} shown in Appendix A.2. ■

Chapter 8

Automaton

Given a program

$$\texttt{let}\ A_1 \triangleq P_1; \ldots; A_k \triangleq P_k\ \texttt{in}\ P_0$$

and functions

$$\mathcal{E}_\star : \mathbf{Proc} \to \mathfrak{M}, \qquad \mathcal{G}_\star : \mathbf{Proc} \to (\mathbf{Lab} \to \mathfrak{M}) \qquad \mathcal{K}_\star : \mathbf{Proc} \to (\mathbf{Lab} \to \mathfrak{M})$$

$$\mathcal{C}_\star : \mathbf{Proc} \to \wp(\mathbf{Lab}) \times \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})$$

defined in the previous chapters, it is possible to create automaton $(\mathsf{Q}, q_0, \delta, \mathsf{E})$, where Q is a set of states with a number of states q_i, each of them associated with an extended multiset E, such that $\mathcal{E}_\star[\![P]\!] \leq_{\mathfrak{M}} \mathsf{E}[q]$. q_0 is the initial state associated with the exposed actions $\mathcal{E}_\star[\![P_0]\!]$ of the main process of the program. δ is a transition relation containing transitions of the form:

- $q_s \Rightarrow_{ll'} q_d$ meaning that in the state q_s two actions labeled l and l' with canonical actions $\partial(l)$ and $\partial(l')$ such that

  $$\exists x : (\partial(l) = \lfloor \overline{x} \rfloor \wedge \partial(l') = \lfloor x \rfloor) \vee (\partial(l') = \lfloor \overline{x} \rfloor \wedge \partial(l) = \lfloor x \rfloor)$$

  may interact and give rise to the state q_d.
- $q_s \Rightarrow_l q_d$ meaning that in the state q_s an internal action τ or a matching action γ with label l may occur and give rise to the state q_d.

The worklist algorithm is used to construct automaton. The main data structures of the algorithm include:

- a set of states Q with each state associated with extended multiset described above;
- a worklist $W \subseteq \mathsf{Q}$ of states that need to be processed;
- a set of edges δ with each edge of the form $(q_s, \tilde{l}, q_d)$, where q_s is the source state, q_d is the destination state and $\tilde{l} \in C_\star$ where $(L_\star, C_\star) = \mathcal{C}_\star[\![P_0]\!]$.

The worklist algorithm is shown in Figure 8.1.

(1) $\mathsf{Q} := \{q_0\}$; $W := \{q_0\}$; $\delta := \emptyset$; $\mathsf{E}[q_0] := \mathcal{E}_\star[\![P_0]\!]$;
(2) `while` $W \neq \emptyset$ `do`
(3) `select` q_s `from` W; $W := W \setminus \{q_s\}$;
(4) `for each` $\tilde{l} \in$ `enabled`$(\mathsf{E}[q_s])$ `do`
(5) `let` $E =$ `transfer`$_{\tilde{l}}(\mathsf{E}[q_s])$
(6) `in update`$(q_s, \tilde{l}, E, \mathsf{L})$;
(7) `squeeze`$(\mathsf{Q}, \delta, \mathsf{L})$;

Figure 8.1: The worklist algorithm

Input for the worklist algorithm is a set of labels of interest L. This set is used as a parameter for granularity function H and for `squeeze` function defined later.

Algorithm begins with initialisation in line (1), where initial state q_0 is added to the empty Q. Transition relation δ is initialised to the empty set. Worklist W is also initialised to contain only q_0. $\mathsf{E}[q_0]$ is assigned to the extended multiset, corresponding to the exposed actions of the initial process P_0.

The algorithm continues while worklist W is not empty (lines (2-6)). In line (3) a "source" state is selected and removed from the worklist W. Then for all $\tilde{l}$, corresponding to the enabled actions of the current state, returned by the `enabled`$(\mathsf{E}[q_s])$ function `transfer`$_{\tilde{l}}(\mathsf{E}[q_s])$ and `update`$(q_s, \tilde{l}, E, \mathsf{L})$ are called. Function `transfer` receives extended multiset of labels of exposed actions in at the entry point of state q_s and returns the extended multiset of labels of exposed actions after execution of actions in the current state. Function `update` modifies Q and δ in such a way the the loop in lines (2-6) terminates.

The worklist algorithm ends with function `squeeze` that modifies Q and δ in such a way that it contains only information about labels from L in order to track only some part of interactions occurring in the process.

8.1 The function `enabled`

The function `enabled` used in line (4) of the worklist algorithm from Figure 8.1 takes extended multiset associated with the current state q_s as its argument. It returns a set $\widetilde{L}$ of labels for actions that may interact or be executed by themselves. $\widetilde{L} \subseteq C_\star$, where $(L_\star, C_\star) = \mathcal{C}_\star[\![P_0]\!]$. Therefore, $\widetilde{L} \in \wp((\mathbf{Lab} \times \mathbf{Lab}) \cup \mathbf{Lab})$.

Function `enabled`(E) will add a label l or a pair of labels (l, l') to $\widetilde{L}$ in following cases:

- if $l \in \mathtt{dom}(E)$ is the label of a τ action, and therefore it is always is enabled;
- if $l \in \mathtt{dom}(E)$ and $l' \in \mathtt{dom}(E)$ are labels of matching "send" and "receive" actions and occur in parallel processes, and therefore (l, l') are enabled;
- if $l \in \mathtt{dom}(E)$ is the label of a *match* action γ (e. g. $[v_1 = v_2]$ or $[v_1 \neq v_2]$):
  - l is enabled if the action has form $[v_1 = v_2]$ and the following condition holds: $V_{in}(l)v_1 \cap V_{in}(l)v_2 \neq \emptyset$, i. e. intersection of $V_{in}(l)v_1$ and $V_{in}(l)v_2$ is not empty and v_1 and v_2 *may* be equal at the entry point of label l.
  - l is enabled if the action has form $[v_1 \neq v_2]$ and the following condition *doesn't* hold: $V_{in}(l)v_1 = V_{in}(l)v_2 \wedge |V_{in}(l)v_1| = |V_{in}(l)v_2| = 1$, i. e. when cardinality of $V_{in}(l)v_1$ and $V_{in}(l)v_2$ is equal to 1 and they are equal, which means that v_1 and v_2 *must* be equal.

The overall function `enabled`(E) may be expressed as:

$$
\begin{aligned}
\mathtt{enabled}(E) \quad = \quad & (C_\star \cap \mathtt{dom}(E)\{l \;\big|\; \partial(l) = \tau\}) \\
\cup \quad & (C_\star \cap (\mathtt{dom}(E) \times \mathtt{dom}(E))) \\
\cup \quad & (C_\star \cap \mathtt{dom}(E) \cap \\
& \quad \{l \;\big|\; \partial(l) = [v_1 = v_2] \wedge V_{in}(l)v_1 \cap V_{in}(l)v_2 \neq \emptyset\}) \\
\cup \quad & (C_\star \cap \mathtt{dom}(E) \cap \{l \;\big|\; \partial(l) = [v_1 = v_2]\} \setminus \\
& \quad \{l \;\big|\; V_{in}(l)v_1 = V_{in}(l)v_2 \wedge |V_{in}(l)v_1| = |V_{in}(l)v_2| = 1\})
\end{aligned}
$$

8.2 The function `transfer`

The function `transfer` is analogous to the transfer function for WHILE-language described in [2]. It is implemented as:

$$\texttt{transfer}_{\tilde{l}}(E) = (E -_{\mathfrak{M}} \mathcal{K}_\star[\![P]\!](\tilde{l})) +_{\mathfrak{M}} \mathcal{G}_\star[\![P]\!](\tilde{l})$$

This function takes the extended multiset E at the current point of the program and $\tilde{l} \in (\mathbf{Lab} \times \mathbf{Lab} \cup \mathbf{Lab})$ enabled at this point. Then it removes information "killed" by $\tilde{l}$ and adds information "generated" by $\tilde{l}$. Instead of operations on sets that are used in classical transfer function for the imperative languages here operations on the extended multiset $\mathfrak{M}$ are used.

8.3 The function `update`

The function `update` takes as its parameters the following:

- q_s — the state being currently analysed;
- $\tilde{l}$ — a pair of labels or a single label of action(s) that occurred in the state q_s;
- E — the extended multiset corresponding to the actions that became exposed after occurrence of $\tilde{l}$.

The overall function is shown in Figure 8.2.

(1) `if` $\exists q \in \mathsf{Q} \wedge H_{\mathsf{L}}(\mathsf{E}[q]) = H_{\mathsf{L}}(E)$
(2) `then` $q_d := q$;
(3) `else select` $q_d \notin \mathsf{Q}$
(4) $\mathsf{Q} := \mathsf{Q} \cup \{q_d\}$; $\mathsf{E}[q_d] := \bot_{\mathfrak{M}}$;
(5) `if` $\neg(\mathsf{E}[q_d] \geq_{\mathfrak{M}} E)$
(6) `then` $\mathsf{E}[q_d] := \mathsf{E}[q_d] \triangledown_{\mathfrak{M}} E; W := W \cup \{q_d\}$;
(7) $\delta := \delta \setminus \{(q_s, \tilde{l}, q) | q \in \mathsf{Q}\} \cup \{(q_s, \tilde{l}, q_d)\}$;
(8) `clean-up`(Q, W, δ)

Figure 8.2: The function `update`

This function checks in line (1) if there exists a state in Q that has the same *granularity* as E. If there is such state, the further work is performed with this

$\mathsf{Q}_{reach} := \{q_0\} \cup \{q \mid \exists n,\ \exists q_1, \ldots, q_n\ :\ (q_0, \ldots, q_1) \in \delta \wedge \ldots \wedge (q_n, \ldots, q) \in \delta)$
$\mathsf{Q} := \mathsf{Q} \cap \mathsf{Q}_{reach};\ \delta := \delta \cap (\mathsf{Q}_{reach} \times (\mathbf{Lab} \cup (\mathbf{Lab} \times \mathbf{Lab})) \times \mathsf{Q}_{reach});$
$W := W \cap Q_{reach};$

Figure 8.3: The function `clean-up`

state (line (2)). Otherwise a new state is added in lines (3-4) to Q and extended multiset associated with it is set to be empty.

In line (5) the test whether new extended multiset E contains additional information in comparison with extended multiset of the destination state. If it does, then the destination state q_d is added to the worklist and its extended multiset is updated using so called *widening operator* $\triangledown_{\mathfrak{M}}$ over extended multisets, defined by:

$$(M_1 \triangledown_{\mathfrak{M}} M_2)(l) = \begin{cases} M_1(l) & \text{if } M_2(l) \leq M_1(l) \\ M_2(l) & \text{if } M_1(l) = 0 \wedge M_2(l) > 0 \\ \infty & \text{otherwise} \end{cases}$$

In line (7) δ is updated with the new transition $(q_s, \tilde{l}, q_d)$ and all old transitions labeled with $\tilde{l}$ are removed.

In line (8) function `clean-up` is called. It performs reachability analysis of Q and removes all states that are unreachable from the initial state q_0 and all transitions that may have these states. The worklist is also updated to contain only the reachable states of Q. The overall implementation of the function `clean-up` is shown in Figure 8.3.

8.4 The granularity function

The granularity function is a function that allows to reuse some states of the automaton Q although they may have different extended multisets. This is useful, if we are interested in a specific part of the system. For example, only some labels of the program or labels independent of their counts may be taken into consideration.

Our choice of granularity function is defined as:

$$H_{\mathsf{L}}(E) = \{l, \infty \mid l \in \mathsf{L} \wedge (E(l) > 0 \vee E(l) = \infty)\}$$

(01) for each $q_s \in \mathsf{Q}$ do
(02) for each $(q_s, \tilde{l}, q_d) \in \delta$ do
(03) if $\tilde{l} \notin (\mathsf{L} \times \mathsf{L} \cup \mathsf{L}) \wedge q_s \neq q_d$ then
(04) for each $(q_d, \tilde{l}_\bullet, q)$ do
(05) $\delta := \delta \setminus (q_d, \tilde{l}_\bullet, q) \cap (q_s, \tilde{l}_\bullet, q)$
(06) for each $(q, \tilde{l}_\bullet, q_d)$ do
(07) $\delta := \delta \setminus (q, \tilde{l}_\bullet, q_d) \cap (q, \tilde{l}_\bullet, q_s)$
(08) $\mathsf{Q} := \mathsf{Q} \setminus q_d$;
(09) if $\tilde{l} \notin (\mathsf{L} \times \mathsf{L} \cup \mathsf{L}) \wedge q_s = q_d$ then
(10) $\delta := \delta \setminus (q_s, \tilde{l}_\bullet, q_s)$
(11) for each $(l \mapsto M) \in \mathsf{E}(q_s)$
(12) if$l \notin \mathsf{L}$ then
(13) $\mathsf{E}(q_s) := \mathsf{E}(q_s) \setminus (l \mapsto M)$

Figure 8.4: The behaviour of `squeeze` function

If $\mathsf{L} = \mathbf{Lab}$, then the function takes into consideration all labels of the program, but if $\mathsf{L} \subset \mathbf{Lab}$ information about some labels is ignored.

8.5 The function `squeeze`

The motivation for the function `squeeze` is that automaton obtained as a result of the worklist algorithm contains information about all labels of the CCS program. But it may be useful to have automaton especially its graphical representation in form of a graph that emphasizes only a part of the whole program. The decision is to "squeeze" automaton in such a way that it contains only the desired set of labels L.

Function `squeeze` takes $\mathsf{L} \subset \mathbf{Lab}$ as its argument. L was also given to the granularity function. But granularity function doesn't prevent information about labels outside of L from entering the automaton. For example, granularity function detects a situation, when in two states a label is mapped to different extended multisets. But when a transaction leads not only to the change in an extended multisets corresponding to some labels but also to introduction of new labels of the exposed action, granularity function permits creation of a new state. The need to merge these states explains the necessity of `squeeze` function.

The overall algorithm is shown in Figure 8.4.

Line (1) introduces cycle for each state q_s of Q all its outgoing edges (transitions)

are analysed (line (2)). If $\tilde{l}$ associated with an edge is not present in the set $\mathsf{L} \times \mathsf{L} \cup \mathsf{L}$ and destinations state q_d is not the same as the source state q_s (line (3)), then all incoming and outgoing edges of q_d are deleted from δ. Instead of them edges that contain q_s instead of q_d are added to δ (lines (4-7)). Then the state q_d is itself deleted from Q in line (8).

If $\wedge l$ associated with an edge is not present in the set $\mathsf{L} \times \mathsf{L} \cup \mathsf{L}$ and destinations state q_d is the same as the source state q_s (line (9)) then this edge is removed from δ in line 10.

Then all entries in $\mathsf{E}(q_s)$ where label l does not present in the set of labels of interest L are removed from $\mathsf{E}(q_s)$ in lines (11-13).

8.6 Implementation issues

Both algorithm for propagation of values and for construction of the automaton use information that may be presented as graphs. That was the reason for the choice to represent Q and δ in both algorithms as graphs with the help of a software library JUNG described in [15]. JUNG is an open-source library for Java that provides a number of functions for creation and manipulation of graphs as well as a visualization framework used to make graphical representation of the obtained automaton.

The algorithm itself was implemented as an instance of class `WL` that has functions `enabled`, `transfer`, `update` and `squeeze`. The main function of the algorithm is called `go`. The worklist algorithm uses functions $\mathcal{G}_\star$, $\mathcal{K}_\star$, $\mathcal{E}_\star$, $\mathcal{C}_\star$, V_{in} and V_{out} that were stored in the class `Env`.

8.7 Examples

Example 8.1 Let us now show an an automaton constructed for the workflow from the Example 1.1. Using V_{in} received by the propagation of values algorithm (see Figure 7.3) we may apply the algorithm for constructing an automaton. After 10 rounds of the algorithm the automaton shown in Figure 8.5 is obtained.

Loops ($V_0 \to V_{15} \to V_{17} \to V_{20} \to V_0$) and ($V_0 \to V_{14} \to V_{16} \to V_{18} \to V_0$) correspond to the choice of the first component of the sum $[x = y]^3\overline{r}\langle x\rangle^4.Client + [x \neq y]^5\overline{r}\langle z\rangle^6.0$) where $Client$ doesn't terminate. State V_{22} may occur if the

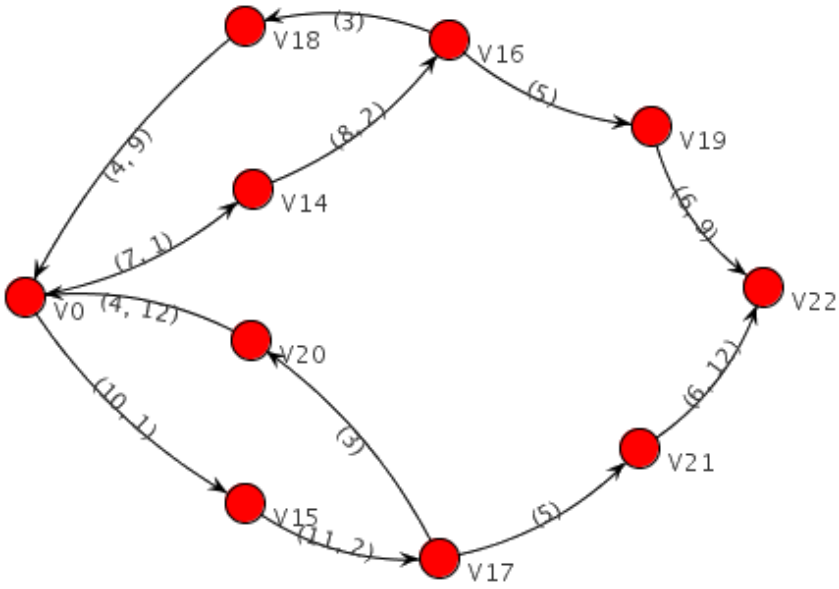


Figure 8.5: Automaton for the program from Example 1.1

second component of the sum was chosen and *Client* terminated leaving only *Server* process active.

If Client-Server program is modified by changing constant b transmitted via channel q in the action with label "11" to a:

```
let
```
$Client \triangleq p(x,z)^1.q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0);$
$Server \triangleq \overline{p}\langle a,c\rangle^7.\overline{q}\langle a\rangle^8.r(w)^9.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle a\rangle^{11}.r(w)^{12}.Server;$
```
in
```
$Client \mid Server$

the automaton shown in Figure 8.6 is received.

Now the algorithm for the propagation of values detected that at the entry points of labels 3 and 5 variables x and y will always take the same value "a" thus making it impossible for action with label 5 to be performed. The workflow algorithm that constructed automaton used this information and left only two loops ($V_0 \to V_{15} \to V_{17} \to V_{20} \to V_0$) and ($V_0 \to V_{14} \to V_{16} \to V_{18} \to V_0$) without adding states V_{19}, V_{21} and V_{22} that can not occur with this setup.

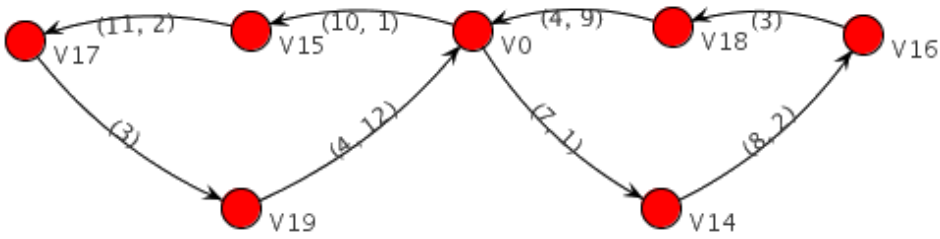


Figure 8.6: Automaton for the modified program from Example 1.1

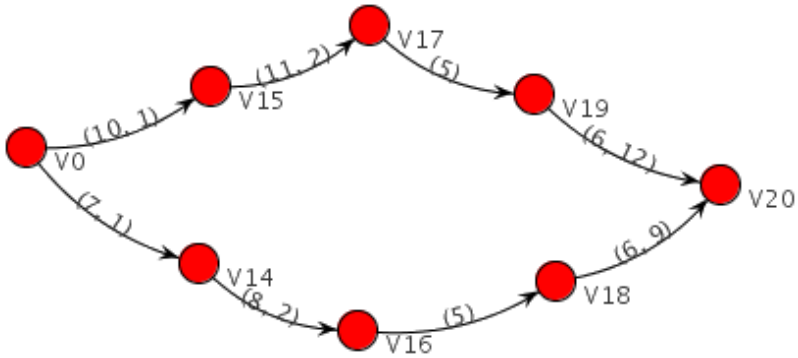


Figure 8.7: Automaton for the modified program from Example 1.1

Another modification of this program consists in transmission of constant b via channel q in actions with both label 8 and 11:

```
let
```
$Client \triangleq p(x,z)^1.q(y)^2.([x=y]^3\overline{r}\langle x\rangle^4.Client + [x\neq y]^5\overline{r}\langle z\rangle^6.0);$
$Server \triangleq \overline{p}\langle a,c\rangle^7.\overline{q}\langle b\rangle^8.r(w)^9.Server + \overline{p}\langle a,d\rangle^{10}.\overline{q}\langle b\rangle^{11}.r(w)^{12}.Server;$
```
in
```
$Client \mid Server$

Now the automaton shown in Figure 8.7 is received.

Here the fact that at the entry point of labels 3 and 5 variables x and y are always different was detected and the loop of "eternal" interaction between *Client* and *Server* was not included into automaton. ■

Example 8.2 Let us consider an example with two parallel processes. Process

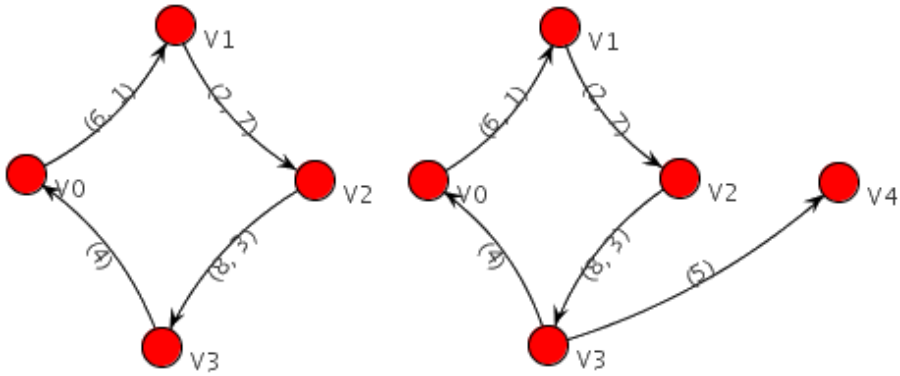


Figure 8.8: Automatons for Example 8.2

Q sends a variable a to process S, which sends it back. Then Q sends a received variable to S one more time. The idea is to check whether analysis detects, that variables recieved by S are the same.

CCS code for this example is the following:

```
let
```
$S \triangleq (g(x)^1.\overline{p}\langle x\rangle^2.m(z,y)^3.([x=z]^4 S + [x \neq z]^5 0));$
$Q \triangleq (\overline{g}\langle a\rangle^6.p(c)^7.\overline{m}\langle c,b\rangle^8.Q);$
```
in
```
$S \mid Q$

Algorithm for the propagation of values calculates $V_{out}(3)x = V_{out}(3)y = \{a\}$. Thus, V_{in} for "match" actions 4 and 5 are also equal. Action $[x = z]^4$ is enabled, but action $[x \neq z]$ is not enabled anymore. The automaton shown in Figure 8.8 (left) arises instead of more general case shown in Figure 8.8 (right).

■

Chapter 9

Analysis of the resulting automaton

When we are use value passing CCS with application to workflows, it is useful to know whether there are states where no further actions may be executed. This may be easily done by analysis of the automaton graph $(\mathsf{Q}, q_0, \delta, \mathsf{E})$. These states correspond to the nodes of this graph with no output edges. After analysis of the extended multisets $\mathsf{E}(q)$ corresponding to these states it is possible to detect whether the whole system has the desired behaviour, for example, parts of the system that according to the extended multiset remain enabled were meant to be enabled at the design time. This analysis makes it possible to detect design-defects of the workflows.

Another analysis is detection of states that have transactions leading only to themselves.

Example 9.1 Let us consider a system with a server and a client working in parallel. They communicate via two channels. Client receives two values x and y via these channels and checks them for equality. If they are equal, client and server continue their execution from the beginning. If they are different, client performs an internal action and terminates. Server sends either two different values or two identical values and repeats this procedure forever.

This example may be written in CCS with value passing as:

let
$Client \triangleq p(x)^1.q(y)^2.([x = y]^3 Client + [x \neq y]^4 \tau^5.0);$
$Server \triangleq \overline{p}\langle a\rangle^6.\overline{q}\langle a\rangle^7.Server + \overline{p}\langle a\rangle^8.\overline{q}\langle b\rangle^9.Server;$
in $Client \mid Server$

After execution of the workflow algorithm the graph, shown in Figure 9.1 was received.

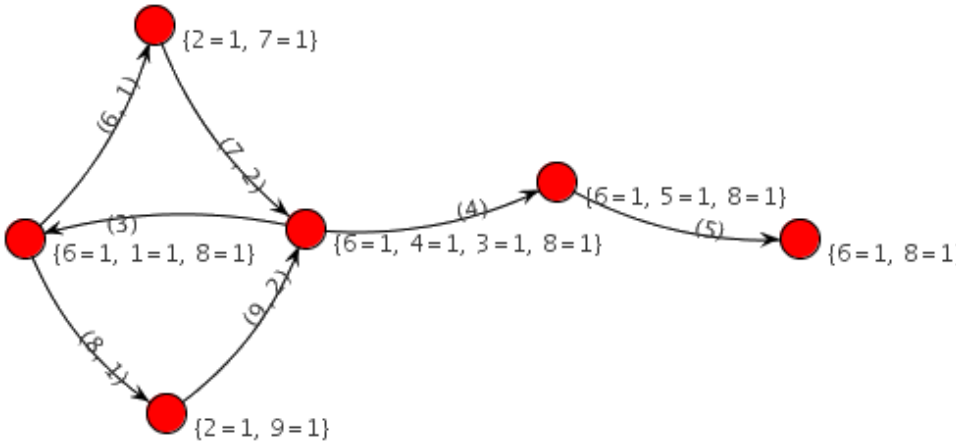


Figure 9.1: Automaton for the Example 9.1

After performing of the analysis state with the extended multiset $\perp_{\mathfrak{M}}[6 \mapsto 1, 8 \mapsto 1]$ was detected. It means that only actions $\overline{p}\langle a\rangle^6$ and $\overline{p}\langle a\rangle^8$ of the *Server* process are enabled, what corresponds to the correct performance of the program. ■

Worklist algorithm may show that some labels of the analysed program become unreachable. For example, automaton may show that some component $\alpha_i^l.P_i$ in a sum $\sum_{i\in I} \alpha_i^l.P_i$ will never be executed. This may happen if α_i^l doesn't have compatible actions from $(L_\star, C_\star) = \mathcal{C}_\star[\![P]\!]$ or if α_i^l is is a "match" action γ that can never be true due to the analysis of the propagation of values described in Chapter 7.

In this case P_i will never be executed. If it contains actions of the form $\overline{x}^l\langle v_1, \ldots, v_k\rangle$ then variables in actions compatible with $\overline{x}$ will never receive values $v_1, \ldots, v_k$. It means, that if we are interested in the values of variables at different points of the CCS program, functions V_{in} and V_{out} need to be re-calculated taking into consideration that some actions of the program became

unreachable.

To capture this the algorithm for the propagation of values must be executed once again. But its initialization must be changed so, that its graph (Q, δ) built on basis of the function $\mathcal{G}_\star$ doesn't contain labels that don't present in the automaton. This is done by the following code:

(1) $VL := \emptyset$;
(2) `for each` $q \in \mathsf{Q}$ `do`
(3) `for each` $[l \mapsto M] \in \mathsf{E}[q]$ `do`
(4) $VL := VL \cup \{l\}$;

Then at the initialization stage of the worklist algorithm for the propagation of values Q is modified so that $\mathsf{Q} := \mathsf{Q} \cap VL$. We also modify line 10 of the algorithm from Figure 7.1 so that it contains an extra condition:

(10) `for each` $(l, l') \in \mathcal{C}_\star$ `where` $l \in LV \wedge l' \in LV$ `do` ...

After execution of the modified algorithm V_{in} and V_{out} will contain more tight approximation of the values of all variables in the program.

Example 9.2 To illustrate the benefit of recalculation of V_{in} and V_{out} let us again consider program from the Example 7.2 V_{in} and V_{out} for which are shown in Appendix A.2 and automaton in Figure 8.7.

Originally $V_{out}(9)w = V_{out}(12)w = \{a, c, d\}$ but after construction of the automaton $VL = \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12\}$ is received. The algorithm for the propagation of values modifies V_{out} which now gives $V_{out}(9)w = V_{out}(12)w = \{a\}$. ■

Usually workflows are designed in such a way that all actions should sometime become active and be either be executed by themselves or interact with other action. If an action was never active, it signals a possible error in the design of the workflow or in its expression in CCS. An analysis that detects unused label was created. It looks through all the edges of the automaton and creates a set $\tilde{L}$ that contains all labels the $\tilde{l}$ associated with the edges. Then subset $\mathbf{Lab} \setminus \tilde{L}$, where **Lab** is a set of all labels of the CCS program, is calculated. This set contains unused labels.

Chapter 10

GUI front-end

To illustrate the work of the algorithms and analyses described in previous chapters and for the testing purpose a GUI front-end was developed. This application permits opening and editing of programs in CCS language with value passing.

Front-end allows to invoke the procedure for creation of the automaton for the currently edited program and displays the obtained graph. It is also possible to create automaton that emphasises only a part of labels in the program by specifying the set of labels L that is given as an argument go the granularity and "squeeze" functions.

For the obtained graph the front-end may show the information about the sets of values that variables of the analyzed program may take at the entry and exit points of all labels of the program (V_{in} and V_{out}). This information is shown after the second pass of the workflow algorithm for the propagation of values, thus it has a better approximation of the actual behaviour of the program.

It is also possible to display information about the exposed labels associated with each state of the automaton.

The resulting graph may be displayed using several layouts, including circle layout, when vertices are arranged in the circle, 3D layout, when the nodes are

placed in a 3-dimensional grid, and self-organising graph layout [7]. The graph may be saved as PNG file.

Information about variables, exposed actions in different states and labelled version of the program are shown in HTML format, as textual format doesn't allow showing of all special symbols used in notation of this thesis. There is a possibility to export information from the front-end to HTML, LaTeX or simple ASCII files, with LaTeX format giving the best representation corresponding to the notation of this thesis.

Based on the obtained graph the next analyses were implemented: the search for the states that don't have outgoing transactions and/or have only self-loops and search for the unused labels.

Screenshot of the application is provided in Appendix A.3. Source code and binaries of the developed program along with examples used in this thesis are included on CD-ROM.

Chapter 11

Worked examples

11.1 How To Become a Recording Star

Let us consider an example of workflow "How To Become a Recording Star" described in [14]. The proposed behaviour is shown in Figure 11.1.

The CCS code, that corresponds to the workflow is shown below.

```
let
```

$Phase1 \triangleq ChooseOne1 \mid WorkYourWayUp \mid TryToGetLucky \mid Sync1;$

$ChooseOne1 \triangleq (\overline{callWork}^{1}.0 + \overline{callTry}^{2}.0);$

$Sync1 \triangleq (retWork^{3}.\overline{next}^{4}.0 + retTry^{5}.\overline{next}^{6}.0);$

$WorkYourWayUp \triangleq (callWork^{7}.\tau^{8}.\overline{retWork}^{9}.0);$

$TryToGetLucky \triangleq (callTry^{10}.\tau^{11}.\overline{retTry}^{12}.0);$

$Phase2 \triangleq (next^{13}.\overline{callRec}^{14}.retRec^{15}.\overline{next2}^{16}.0) \mid MakeRec;$

$MakeRec \triangleq (callRec^{17}.\tau^{18}.\overline{retRec}^{19}.0);$

$Phase3 \triangleq (next2^{20}.(Tour \mid SubPhase3 \mid Sync3));$

$Tour \triangleq (\overline{callRehearse}^{21}.retRehearse^{22}.\overline{callDo}^{23}.retDo^{24}.SendSync)$
$\mid RehearseTour \mid DoTour;$

$RehearseTour \triangleq (callRehearse^{25}.\tau^{26}.\overline{retRehearse}^{27}.0);$

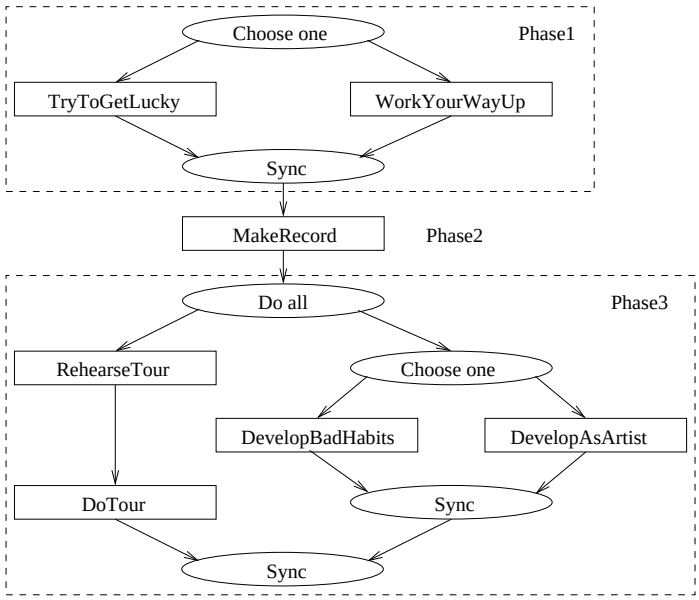


Figure 11.1: How To Become a Recording Star

$DoTour \triangleq (callDo^{28}.\tau^{29}.\overline{retDo}^{30}.0);$
$SubPhase3 \triangleq ChooseOne2 \mid DevelopBadHabits \mid DevelopAsArtist \mid Sync2;$
$ChooseOne2 \triangleq (\overline{callHabits}^{31}.0 + \overline{callArtist}^{32}.0);$
$Sync2 \triangleq (retHabits^{33}.SendSync + retArtist^{34}.SendSync);$
$SendSync \triangleq (\overline{sync}^{35}.0);$
$DevelopBadHabits \triangleq (callHabits^{36}.\tau^{37}.\overline{retHabits}^{38}.0);$
$DevelopAsArtist \triangleq (callArtist^{39}.\tau^{40}.\overline{retArtist}^{41}.0);$
$Sync3 \triangleq (sync^{42}.sync^{43}.0);$
`in` $Phase1 \mid Phase2 \mid Phase3$

The whole workflow is divided into three phases. The first phase implements choosing and executing only one of two processes. It is done with the help of summation and sending signals to the processes ($ChooseOne1$). Synchronization is implemented as a summation as well ($Sync1$), but now it receives signals from the processes. The second phase executes a subprocess and proceeds to the next phase. The third phase consists in parallel execution of two branches. The first branch is a sequence of two processes and the second branch executes one of two processes. The phase ends with synchronization.

Building of the automaton resulted in a graph shown in Appendix A.4. The

branches V_0, V_1, V_3, V_5, V_7 and V_0, V_2, V_4, V_6, V_7 correspond to the first phase, sequence V_7, V_8, V_9, V_{10} corresponds to the second phase. The rest of the graph corresponds to the third phase. As the third phase consists in parallel execution of processes, this part of the graph is a grid, that captures all possible interleavings of actions. Labels on the horizontal edges correspond to the actions from the branch $RehearseTour$; $DoTour$;. Labels on the vertical edges correspond to the actions from the branch $ChooseOne2$. The two last rows of vertices model the synchronization ($Sync3$).

The analysis that searches for the unused labels shows that all labels of the initial programs were used in the automaton. The analysis of deadlocks and termination states shows that there is only one termination state, namely V_{65} with no exposed actions associated with it ($\mathsf{E}[V_{65}] = \bot_{\mathfrak{M}}$). These facts confirm the correctness of the model.

11.2 Car repair

Let us consider a part of a state machine from [6]. $CarRepair$ process requests from $Reasoner$ process a services: credit, and then garage. In case of timeout modelled by a signal via channel $after$ the $CarRepair$ process stops normal execution and requests compensation. It should be checked, that the previous reservation is also revoked.

The code for this model is shown below:

$$
\begin{array}{l}
\texttt{let}\ CarRepair \triangleq (\overline{getService}\langle id, the\_credit\rangle^{1}. \\
\quad (after^{2}.\overline{compensate}\langle null\rangle^{3}.0 + offer(id, credit)^{4}. \\
\quad\quad (after^{5}.\overline{compensate}\langle credit\rangle^{6}.0 + \overline{getService}\langle id, the\_garage\rangle^{7}. \\
\quad\quad\quad offer(id, garage)^{8}.(after^{9}.\overline{compensate}\langle garage\rangle^{10}.0)))); \\
\quad Reasoner \triangleq (getService(xid, name)^{11}. \\
\quad\quad (compensate(what)^{12}.Compensator + [name = the\_credit]^{13} \\
\quad\quad \overline{offer}\langle xid, credit\rangle^{14}.compensate(what)^{15}.Compensator \\
\quad\quad + [name = the\_garage]^{16}\overline{offer}\langle xid, garage\rangle^{17}. \\
\quad\quad compensate(what)^{18}.Compensator + [name = the\_truck]^{19} \\
\quad\quad \overline{offer}\langle xid, truck\rangle^{20}.compensate(what)^{21}.Compensator)) \\
\quad\quad \mid Reasoner; \\
\quad Compensator \triangleq ([what = null]^{22}0 \\
\quad\quad + [what = credit]^{23}\overline{revoke}\langle credit\rangle^{24}.0 \\
\quad\quad + [what = garage]^{25}\overline{revoke}\langle garage\rangle^{26}.\overline{revoke}\langle credit\rangle^{27}.0); \\
\quad Revoker \triangleq (revoke(smth)^{28}.0) \mid Revoker; \\
\quad Deadline \triangleq (\overline{after}^{29}.Deadline);
\end{array}
$$

The obtained automaton contains 37 states, but in order to capture the behaviour of the program we may simplify it using "squeeze" function and eliminating information concerning communication via channel $after$. We get the automaton with 27 states shown in Figure 11.2.

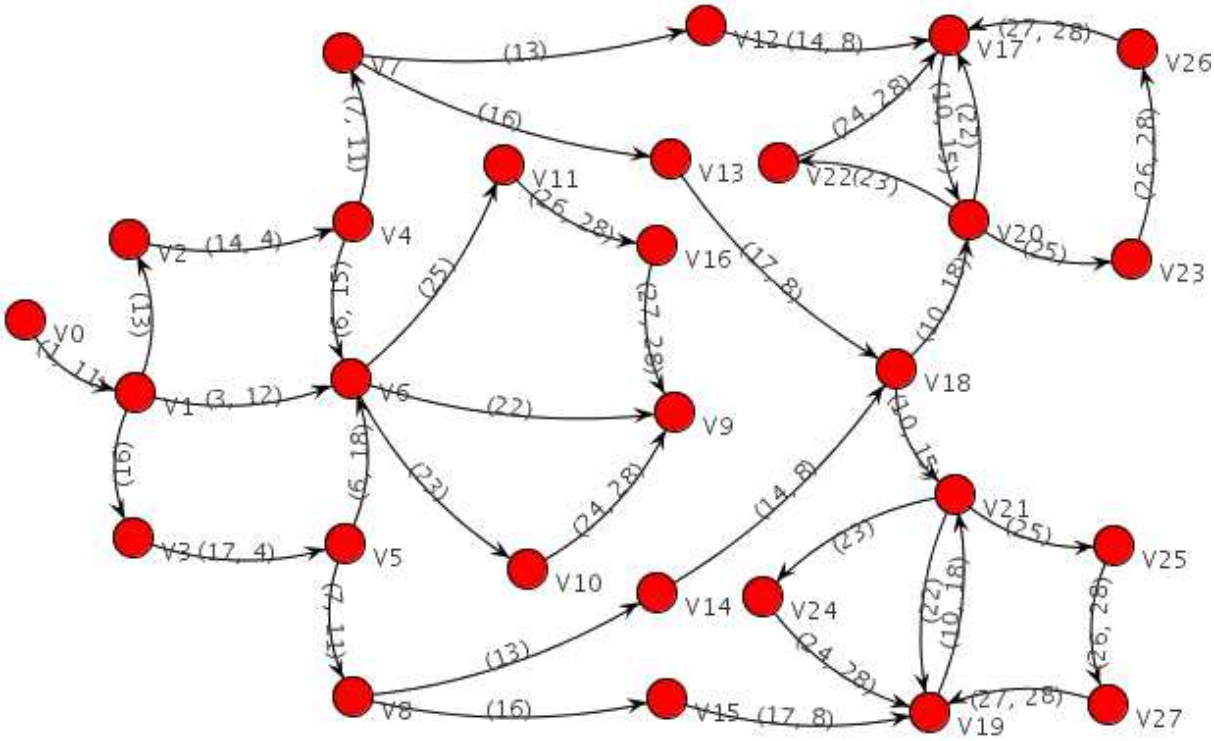


Figure 11.2: Car Repair Workflow

As our model is an over-approximation, it contains more states than the actual execution would result into. For example, there is a transaction $(V_1, (16), V_3)$, while only $(V_1, (13), V_2)$ and $(V_1, (3, 12), V_6)$ would be really possible during the actual execution. But nevertheless, it can be seen from the graph, that every time after compensation of the garage (transactions $(V_{11}, (26, 28), V_{16})$, $(V_{23}, (26, 28), V_{26})$ and $(V_{25}, (26, 28), V_{27})$) the compensation of the credit follows (transaction $(V_{16}, (27, 28), V_9)$, $(V_{26}, (27, 28), V_{17})$ and $(V_{27}, (27, 28), V_{19})$).

11.3 Traveller

The next example models the workflow between a traveller, travel agent, plane company and bank. A traveller requests travel agent about destination, the agent queries a plane company about the possibility of flight reservation to this destination. In case of negative response it sends "cancel" message to the traveller goes into initial state. In case of positive response, travel agent asks traveller for confirmation. If a confirmation is sent along with bank details, travel agent makes reservation in the plane company. If reservation succeeds, travel agent connects with bank, gets the money from traveller's account and transfers it to the plane company. The plane company returns reservation details, which are sent by the travel agent to the traveller.

The resulting code in CCS is shown below:

$$
\begin{aligned}
&\texttt{let } Traveller \triangleq (\overline{request}\langle destination\rangle^{1}.(cancel^{2}.Traveller + approve(det)^{3}.\tau^{4}.\\
&\quad (\overline{no}^{5}.Traveller + \overline{yes}\langle bankDetails\rangle^{6}.\\
&\quad\quad (cancel^{7}.Traveller + reply(resDet)^{8}.Traveller))));\\
&\quad TravelAgent \triangleq (request(d)^{9}.\overline{query}\langle d\rangle^{10}.response(isav, det)^{11}.\\
&\quad\quad ([isav = false]^{12}\overline{cancel}^{13}.TravelAgent + [isav = true]^{14}\overline{approve}\langle det\rangle^{15}.\\
&\quad\quad\quad (no^{16}.TravelAgent + yes(bankDet)^{17}.\overline{reserve}\langle details\rangle^{18}.\\
&\quad\quad\quad\quad reserveok(flag)^{19}.([flag = false]^{20}\overline{cancel}^{21}.TravelAgent +\\
&\quad\quad\quad\quad\quad [flag = true]^{22}\overline{connectBank}\langle taID, bankD\rangle^{23}.transferMoney^{24}.\\
&\quad\quad\quad\quad\quad\quad \overline{moneyToPlane}\langle det\rangle^{25}.done(reserveDet)^{26}.\overline{reply}\langle reserveDet\rangle^{27}.\\
&\quad\quad\quad\quad\quad\quad\quad TravelAgent))));\\
&\quad PlaneCompany \triangleq (query(dd)^{28}.\tau^{29}.(\overline{response}\langle true, details\rangle^{30}.0 +\\
&\quad\quad \overline{response}\langle false, details\rangle^{31}.0)) \mid (reserve(details)^{32}.\tau^{33}.\\
&\quad\quad\quad (\overline{reserveok}\langle true\rangle^{34}.0 + \overline{reserveok}\langle false\rangle^{35}.0))\\
&\quad\quad \mid (moneyToPlane(ds)^{36}.\tau^{37}.\overline{done}\langle resDetails\rangle^{38}.0) \mid PlaneCompany;\\
&\quad Bank \triangleq (connectBank(travelAgentID, bankDs)^{39}.\tau^{40}.\\
&\quad\quad \overline{transferMoney}^{41}.Bank);\\
&\texttt{in } Traveller \mid TravelAgent \mid PlaneCompany \mid Bank
\end{aligned}
$$

The created automaton is presented in Figure 11.3. The interesting property of this automaton is that there are two cases, when two edges lead to the same state: $(V_3, (30, 11), V_4)$, $(V_3, (31, 11), V_4)$ and $(V_{11}, (34, 19), V_{12})$, $(V_{11}, (35, 19), V_{12})$. These edges correspond to sending of different values via channels. Let's analyse the second case. Here plane company sends "true" or "false" constant via channel *reserveok*. Travel agent receives a variable $flag$ via this channel and analyses its value with the help of "match" actions $[flag = false]$ and $[flag = true]$. Here the fork $(V_{12}, (20), V_{13})$, $(V_{12}, (22), V_{13})$ arises leading to the different further execution.

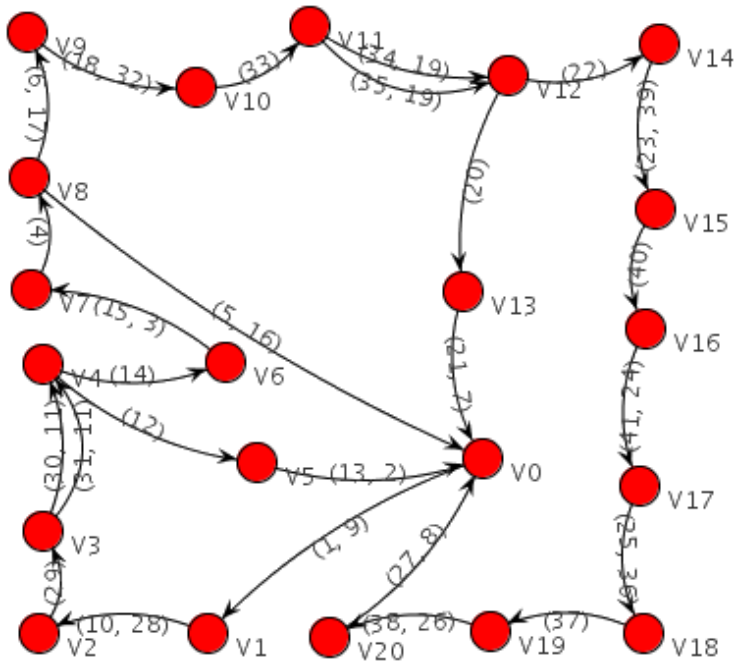


Figure 11.3: Traveller example

The system was designed in such a way that it should enter the initial state both after the correct execution and after all failures. The analysis detected that graph is circular and does not contain any deadlocks. The automaton also uses all the labels of the CCS model. This confirms the correctness of the intended design.

If we modify this example so that the traveller process terminates after the positive reply from the travel agent by adding $\emptyset$ action after $reply(resDet)$[8], but still continues execution from the beginning in case of all negative replies, the resulting automaton will no longer be circular. It will have one termination state V_{21} with exposed actions $\perp_{\mathfrak{M}}[39 \mapsto 1, 36 \mapsto \infty, 32 \mapsto \infty, 9 \mapsto 1, 28 \mapsto \infty]$. These exposed actions show that now processes $TravelAgent$, $PlaneCompany$ and $Bank$ are ready to begin interaction, but none of the actions from the process $Traveller$ is exposed and system cannot continue execution. The automaton for this case is shown in Appendix A.5.

Chapter 12

Conclusion

In the undergone work the extension of Calculus for Communicating Systems (CCS) with value passing was chosen for performing of a static analysis with application to workflos.

A Monotone Framework for CCS described in [10] was selected as a starting point for the analysis that captures transitions among configurations that arise during the running of CCS processes. As the original analysis doesn't focus on the values passed via channels to the different processes, what is crucial for the workflows, it was extended. The extension consists in approximations of the actual values that variables of CCS processes may take at execution time and strengthening of conditions for constructing a finite automaton.

The syntax of CCS was also extended to support additional operations on variables — comparison and conditional execution of the processes.

The analysis was presented as a transition graph that approximates execution states of the CCS program.

A graphical front-end was developed to include the overall process of editing CCS programs, parsing them, performing the analysis, visualising the results in form of the graph and showing information about properties of the transition graph, variables and different states of the CCS program. During the work,

different parts of developed analysis were tested on a number of small examples, and in the end several more complex examples were studied.

The version of CCS used in analysis described in this thesis does not support specification of arguments for the definitions `let` $A_i(x_1, \ldots, x_m) \triangleq P_i$;. Further work may consist in adding this feature to the language, what leads to the more complex analysis. Another possibility is to extend CCS with value passing to π-calculus, that permits transfer of channel names. Another direction of the further work is the deeper investigation of the structure of the resulting automaton. As transfer of real-world workflows directly to CCS may lead to the great size of models and and is a time-consuming task, the further research may concern the development of automatic tools for this purpose.

Appendix A

Appendices

A.1 Syntax of CCS with value passing

let	::=	`let in` *proc_list*
	|	`let` *def_list* `in` *proc_list*
def_list	::=	*def* `;`
	|	*def_list def* `;`
def	::=	*process* `::=` *proc_list*
proc_list	::=	*proc*
	|	*proc_list* `|` *proc*
	|	`(` *proc_list* `)`
proc	::=	`new` *CHANNEL* `(` *proc_list* `)`
	|	`(` *sum* `)`
	|	*process*
	|	`0`
sum	::=	*seq*
	|	*sum* `+` *seq*

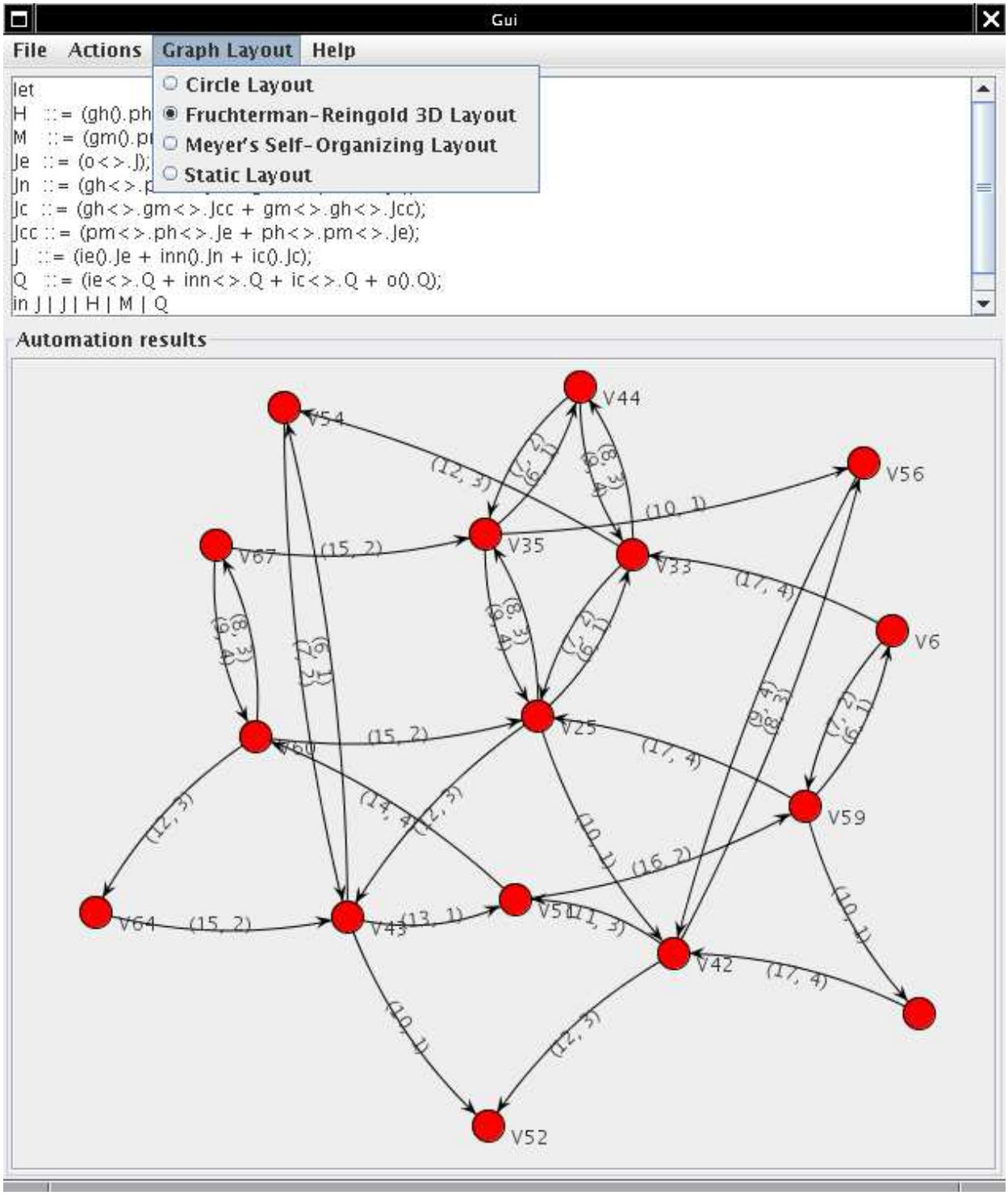
$$
\begin{array}{rcl}
seq & ::= & prefix\ proc\_list \\
 & | & prefix\ seq \\
prefix & ::= & \tau\texttt{.} \\
 & | & CHANNEL\ \texttt{(}\ param\_list\ \texttt{).} \\
 & | & CHANNEL\ \texttt{<}\ param\_list\ \texttt{>.} \\
 & | & \texttt{[}\ VAR\ \texttt{=}\ VAR\ \texttt{]} \\
 & | & \texttt{[}\ VAR\ \texttt{!=}\ VAR\ \texttt{]} \\
param\_list & ::= & \texttt{/* empty */} \\
 & | & param \\
 & | & param\_list\ \texttt{,}\ param \\
param & ::= & VAR
\end{array}
$$

A.2 V_{in} and V_{out} for Example 7.2

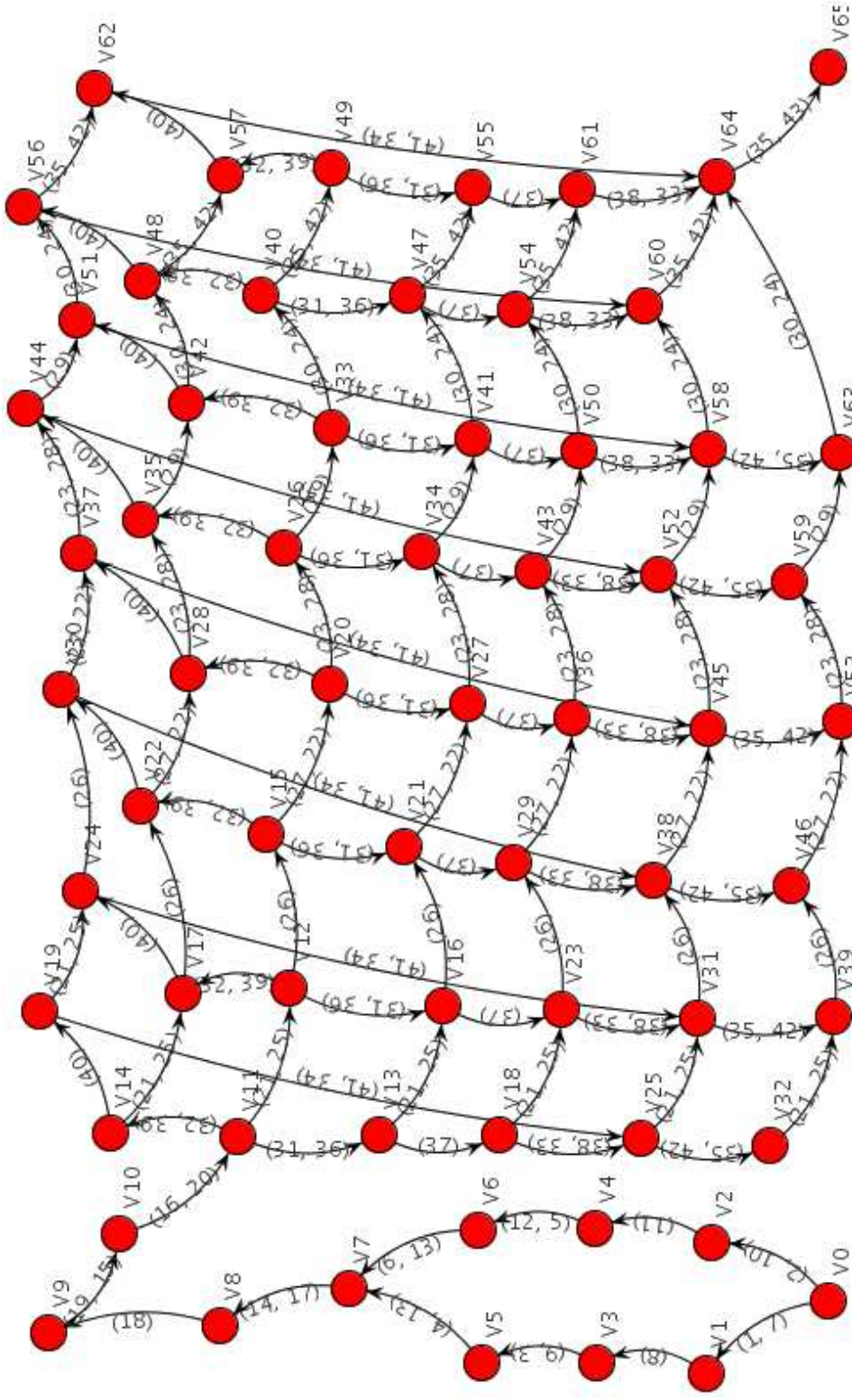
l	$V_{in}(l)$
1	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
2	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,c,d\}]\!]$
3	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
4	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
5	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
6	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
7	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
8	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
9	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
10	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
11	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
12	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$

l	$V_{in}(l)$
1	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a,c,d\}]\!]$
2	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
3	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
4	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
5	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
6	$[\![x \mapsto \{a\}, w \mapsto \{a,c,d\}, z \mapsto \{c,d\}, y \mapsto \{a\}]\!]$
7	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
8	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
9	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
10	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
11	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$
12	$[\![x \mapsto \{a,c,d\}, w \mapsto \{a,c,d\}, z \mapsto \{a,c,d\}, y \mapsto \{a,c,d\}]\!]$

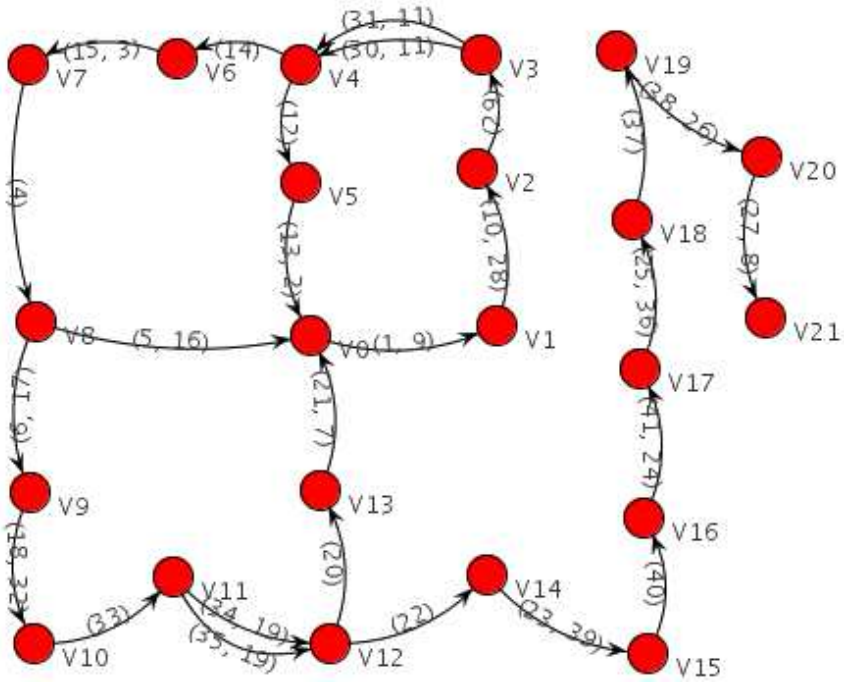
A.3 Screenshot of GUI



A.4 How to Become a Recording Star workflow



A.5 Modified example of Traveller workflow



Bibliography

[1] Elliot Joel Berk and C. Scott Ananian. *JLex: A Lexical Analyzer Generator for Java*. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[2] C. L. Hankin F. Nielson, H. Riis Nielson. *Principles of Program Analysis*. Springer, second edition, 2005.

[3] Frank Puhlmann Hagen Overdick and Mathias Weske. *Towards a Formanl Model for Agile Service Discovery and Integration*. Hasso-Plattner-Institute for IT Systems Engineering at the University of Potsdam.

[4] J. Hillston. Process algebras for quantitative analysis. In *Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 239–248, Chicago, June 2005. IEEE Computer Society Press.

[5] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[6] S. Gilmore M. Wirsing, A. Clark and other. *Semantic-Based Development of Service-Oriented Systems*. `http://homepages.inf.ed.ac.uk/stg/publications/forte2006.pdf`.

[7] Bernd Meyer. Self-organizing graphs — a neural network perspective of graph layout. *Lecture Notes in Computer Science*, 1547:246–262, 1998.

[8] Robin Milner. *Communicating and Mobile Systems: The $\pi-$calculus*. Cambridge University Press, 1999.

[9] Jayadev Misra and William R. Cook. *Computation Orchestration. A Basis for Wide-Area Computing*. The University of Texas at Austin.

[10] Hanne Riis Nielson and Flemming Nielson. *A Monotone Framework for CCS*. Informatics and Mathematical Modelling, Technical University of Denmark, 2006.

[11] Weske M. Puhlmann F. Using the pi-calculus for formalizing workflow patterns. In *van der Aalst, W. Benatallah, B., Casati, F., eds.: BPM 2005, volume 3649 of LNCS*, pages 153–168, Berlin, 2005. Springer-Verlag.

[12] C. Scott Ananian Scott Hudson, Frank Flannery. *CUP: LALR Parser Generator for Java*. `http://www2.cs.tum.edu/projects/cup/`.

[13] Christian Stefansen. *Expressing Workflow Patterns in CCS*. Department of Computer Science. University of Copenhagen, 2005. `http://www.stefansen.dk/papers/workflowpatterns.pdf`.

[14] Christian Stefansen. *SMAWL: A SMAll Workflow Language Based on CCS*. CAiSE Forum, 2005. `http://www.stefansen.dk/papers/smawl.pdf`.

[15] The JUNG Framework Development Team. *JUNG: Java Universal Network/Graph Framework*. `http://jung.sourceforge.net/index.html`.

[16] Hitesh Dholakia Tony Andrews, Francisco Curbera. Business process execution language for web services. `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`, 2003.

[17] Windows workflow foundation official site. `http://wf.netfx3.com/`.