Runtime Systems and Behavioural Abstractions

Wolf Zimmermann

Martin-Luther-University Halle-Wittenberg

Contents

Objectives

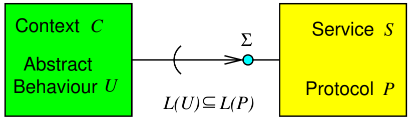
- Understanding the Principles of Runtime Systems
- Understanding the Principles Abstraction
- Understanding the Principles of Software Model Checkibng
- Understanding the use of Rewrite Systems for Abstractions

1. Introduction
2. While-Languages
3. Procedures
4. Concurrency
5. Synchronization
6. Summary

1. Introduction

Idea of Protocol Conformance Checking

- Let S be a service with an interface providing services Σ and protocol P
- Let C be a context using S
- ⇒ Model behaviour of C as a rewrite system U specifying the set $L(U)$ of sequences of service calls to S
- Check whether $L(C) \subseteq L(S)$
- If the answer is no, present a sequence $\sigma \in L(C) \setminus L(S)$.
- ☞ U should be automatically constructed from C.



To Do

- Runtime systems
- Concepts of Abstraction
- ☞ also used for Software Model Checkings (is it possible to avoid undesired situations?)

2. While Languages

Our Example Language

$\langle \mathit{Prog}\rangle$	$\rightarrow$ { $\langle \mathit{Decls}\rangle\langle \mathit{Stats}\rangle$ }	A program is executes its declarations followed by executing its statements.
$\langle \mathit{Decls}\rangle$	$\rightarrow (\langle \mathit{Decl}\rangle$`;`$)^*$	Declarations are executed in its order
$\langle \mathit{Decl}\rangle$	$\rightarrow \langle \mathit{type}\rangle$ `identifier`	Allocates a variable of the type
$\langle \mathit{type}\rangle$	$\rightarrow$ `int (const)`	integers of a given size
$\langle \mathit{Stats}\rangle$	$\rightarrow (\langle \mathit{Stat}\rangle)^*$	Statements are executed in its order
$\langle \mathit{Stat}\rangle$	$\rightarrow \langle \mathit{Assign}\rangle \mid \langle \mathit{If}\rangle \mid \langle \mathit{While}\rangle \mid \langle \mathit{Block}\rangle$	
$\langle \mathit{Assign}\rangle$	$\rightarrow$ `identifier=`$\langle \mathit{Expr}\rangle$`;`	The value of the RHS is stored at the variable at the LHS
$\langle \mathit{If}\rangle$	$\rightarrow$ `if (`$\langle \mathit{expr}\rangle$`)` $\langle \mathit{Stat}\rangle$ `else` $\langle \mathit{Stat}\rangle$	
	If the value of the condition is $\neq 0$ the first statement is executed. Otherwise the first statement is executed	
$\langle \mathit{While}\rangle$	$\rightarrow$ `while (`$\langle \mathit{expr}\rangle$`)` $\langle \mathit{Stat}\rangle$	
	The statement is executed while the value of the condition is $\neq 0$	
$\langle \mathit{Block}\rangle$	$\rightarrow$ { $\langle \mathit{Stats}\rangle$ }	The execution of a block executes its statement

- Integers are represented by 2-complement and can be coerced to integer of larger sizes
- Expressions are evaluated as usual (without overflows or underflows). The program execution aborts if a division by zero happens.
- The value of variable *res* is the output, the initial values of the other variables are the input

Example 1: A While-Program

```
    { int x(4);
      int y(4);
      int res(4);
q0 :  if (x>0&&y>0) {
q1 :  res=0;
q2 :  while (x!=y)
q3 :    if (x>y)
q4 :      x=x-y;
q5 :    else y=y-x;
q6 :  res=x;
q7 :  }
q8 :  else res=-1;
q9 :}
```

$$
\begin{array}{lll}
Q & \triangleq \{(q_i,x,y,r) : 0 \le i \le 9, -8 \le x,y,r \le 7\} & \\
I & \triangleq \{(q_0,x,y,r) : -8 \le x,y,r \le 7\} & \\
F & \triangleq \{(q_9,x,y,r) : -8 \le x,y,r \le 7\} & \\
R & \triangleq & \{(q_0,x,y,r) \rightarrow (q_1,x,y,r) : 0 \le x,y \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_0,x,y,r) \rightarrow (q_8,x,y,r) : -8 \le x,y \le 0, -8 \le r \le 7\} \\
 & \cup & \{(q_1,x,y,r) \rightarrow (q_2,x,y,0) : -8 \le x,y \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_2,x,y,r) \rightarrow (q_3,x,y,r) : -8 \le x,y \le 7, x \ne y, -8 \le r \le 7\} \\
 & \cup & \{(q_2,x,y,r) \rightarrow (q_6,x,y,r) : -8 \le x,y \le 7, x = y, -8 \le r \le 7\} \\
 & \cup & \{(q_3,x,y,r) \rightarrow \{(q_4,x,y,r) : -8 \le y < x \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_3,x,y,r) \rightarrow \{(q_5,x,y,r) : -8 \le x \le y \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_4,x,y,r) \rightarrow (q_2,x-y,y,r) : -8 \le x,y \le 7, x \ne y, -8 \le r \le 7\} \\
 & \cup & \{(q_5,x,y,r) \rightarrow (q_2,x,y-x,r) : -8 \le x,y \le 7, x \ne y, -8 \le r \le 7\} \\
 & \cup & \{(q_6,x,y,r) \rightarrow (q_7,x,y,x) : -8 \le x,y \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_7,x,y,r) \rightarrow (q_9,x,y,r) : -8 \le x,y \le 7, -8 \le r \le 7\} \\
 & \cup & \{(q_8,x,y,r) \rightarrow (q_9,x,y,-1) : -8 \le x,y \le 7, -8 \le r \le 7\}
\end{array}
$$

Principles for Construction of Finite State Machines

- For each statement and block end there is a program point
- State $\triangleq$ values of variables and program point
- State transition rules formally define the semantics of the statement
- Execution according to the execution order
- Several initial states (for each value of variable)
- Possible alphabet could be the statements (here not considered)
- Final state is the state at the block end of the program

Discussion

Observations

- Program semantics of while-programs can be represented as a finite state machine, if all data types are finite types
- ☞ Holds for all programs without procedures, concurrency, exceptions
- However, number of states is horribly large (**state explosion problem**)

Software Model Checking

Does the finite state machine A defining the program semantics satisfy a certain property?

- Let G be the graph representation for A.
- Is the final state always being reached?
- ☞ Is G acyclic?
- ⇒ If not the program may not terminate.
- Can the final state be reached?
- ☞ Is there a path from an initial state to a final state in G?
- ⇒ If not, the program never terminates.
- Other properties φ can be checked by jumps:

$\vdots$
$\hat{q} : \cdots$ //property φ must hold
q : `if ( ` $\neg\varphi$ ` ) goto ` q
q' :$\cdots$

- ⇒ Is there a path from the initial state to q'?

Abstraction

Objective

Reduce the number of states

Idea

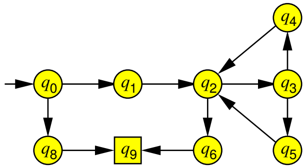
Define a finite state machine $A \triangleq (Q', I', F', \rightarrow')$ such that $|Q'| \ll |Q|$ and there is mapping $\alpha : Q \rightarrow Q'$ satisfying the following properties:

- $\alpha(I) \subseteq I'$ and $\alpha(F) \subseteq \alpha(F')$
- If $q \rightarrow \hat{q}$ then $\alpha(q) \rightarrow \alpha(\hat{q})$:

$$\begin{tikzcd}
\alpha(q) \arrow[r] & \alpha(\hat{q}) \\
q \arrow[u, "\alpha"] \arrow[r] & \hat{q} \arrow[u, "\alpha"']
\end{tikzcd}$$

Example 2: Abstraction of the FSM in Example 1

```
     { int x(4);
       int y(4);
       int res(4);
q0 :   if (x>0&&y>0) {
q1 :   res=0;
q2 :   while (x!=y)
q3 :     if (x>y)
q4 :       x=x-y;
q5 :     else y=y-x;
q6 :   res=x;
q7 :   }
q8 :   else res=-1;
q9 :}
```



Observation: Graph representation has cycles?

⇒ According to the abstraction the answer to termination is no? (**false negative**)

- Without the loop, the answer would still be correct.

Reason: There are more paths in the abstraction than in the original finite state machine

⇒ Negative answers to questions to the absence of path properties may be false

⇒ Positive answers to the absence of path properties are still correct

$Q' \triangleq \{q_0, \ldots, q_9\}$

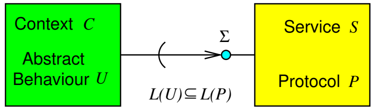
$\alpha : Q \to Q'$ is defined by $\alpha(q_i, x, y, r) \triangleq q_i,\ 0 \le i \le 9, -8 \le x, y, r, \le 7$

$F' \triangleq \{q_9\}$ and $I' \triangleq \{q_0\}$

$R' \triangleq \{\ q_0 \to q_1, q_0 \to q_8, q_1 \to q_2, q_2 \to q_3, q_2 \to q_6, q_3 \to q_4, q_3 \to q_5,$
$\quad q_4 \to q_2, q_5 \to q_2, q_6 \to q_7, q_7 \to q_9, q_8 \to q_9\}$

Discussion

- Abstraction may increase feasability of model checking
- However, the price to pay are false negatives
- Application to protocol conformance checking:



  - Let Σ be the alpabet of the finite state machine U specifying the behaviour of C
  - For a service call $q : A.f(\cdots);\ q'$: add a transition $q \xrightarrow{f} q'$

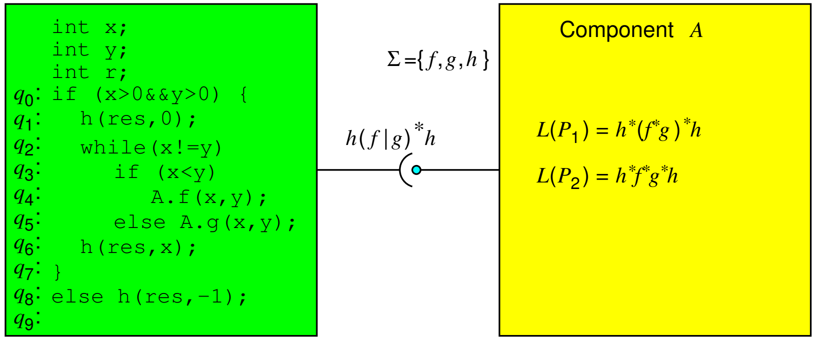
☞ We write $q \xrightarrow{f} q'$ instead of $qf \to q'$

⇒ $L(U)$ is a superset of the sequence of service calls being executed

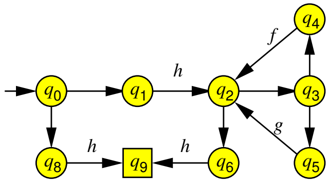
⇒ Protocol conformance checking $L(U) \subseteq L(P)$ for two finite state machines

☞ False negative are possible because of abstraction.

Example 3: Protocol Conformance Checking



```
    int x;
    int y;
    int r;
q0: if (x>0&&y>0) {
q1:   h(res,0);
q2:   while(x!=y)
q3:      if (x<y)
q4:          A.f(x,y);
q5:      else A.g(x,y);
q6:   h(res,x);
q7: }
q8: else h(res,-1);
q9:
```



- $L(U) = L(h(f|g)^*h) \subseteq L(h^*(f^*g)^*h^*) = L(P_1)$
- $hffh \in L(U) \setminus L(P_2) \Rightarrow$ Protocol Violation

---

2. Procedures

Objectives

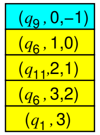
Add procedures to the while language:

$\langle \mathit{Prog}\rangle \to \langle \mathit{Decls}\rangle^*\langle \mathit{Proc}\rangle^*\{\langle \mathit{Decls}\rangle\langle \mathit{Stats}\rangle\}$
$\langle \mathit{Proc}\rangle \to \langle \mathit{type}\rangle$ `identifier` $((\langle \mathit{Pars}\rangle|\lambda))\{\ \langle \mathit{Decls}\rangle\langle \mathit{Stats}\rangle\}$
$\langle \mathit{Pars}\rangle \to \langle \mathit{Par}\rangle(,\langle \mathit{Par}\rangle)^*$
$\langle \mathit{Par}\rangle \to \langle \mathit{type}\rangle$ `identifier`
$\langle \mathit{Stat}\rangle \to \cdots|\langle \mathit{Call}\rangle|\langle \mathit{Ret}\rangle$
$\langle \mathit{Call}\rangle \to$ `identifier` $((\langle \mathit{Args}\rangle|\lambda));$
$\langle \mathit{Ret}\rangle \to$ `return` $((\langle \mathit{Expr}\rangle|\lambda));$
$\langle \mathit{Args}\rangle \to \langle \mathit{Expr}\rangle(,\langle \mathit{Expr}\rangle)^*$

- A procedure with return type `void`$\triangleq$`int(0)` is called proper.
- Any other procedure is called a function.
- A procedure call allocates the parameters and local variables, passes the arguments by value to the corresponding parameters, and then executes the statements.
- A function call allocates in addition a return parameter. This must be last argument in a function call which must be a variable.
- If a return statement is being executed then the execution continues with the statement after the corresponding call.
- In case of a function, the return statement must have an expression. Its value is stored at the last argument.

Example 5: Procedure Calls

```
     void f(int(4) i) {
       int(4) j;
q2 :   j=i-1;
q3 :   if (j<0)
q4 :     return;
q5 :   else g(j);
q6 : }
     void g(int i) {
       int j;
q7 :   j=i-1;
q8 :   if (j<0)
q9 :     return;
q10 :  else f(j);
q11 :}
     {int(4) k;
q0 :   f(k);
q1 : }
```



$$
\begin{array}{ll}
Q & \triangleq \{z\} \\
I & \triangleq \{z\} \\
F & \triangleq \{z\} \\
S & \triangleq \{(q_i,k) : i \in \{0,1\}, -8 \le k \le 7\} \\
 & \cup \{(q_h,i,j) : 2 \le h \le 11, -8 \le i,j \le 7\} \\
R & \triangleq \quad \{(q_0,k)z \xrightarrow{f} (q_1,k)(q_2,k,j)z : -8 \le k,j \le 7\} \\
 & \cup \quad \{(q_2,i,j)z \longrightarrow (q_3,i,i-1)z : -8 \le i,j \le 7\} \\
 & \cup \quad \{(q_3,i,j)z \longrightarrow (q_4,i,j)z : -8 \le i \le 7, -8 \le j < 0\} \\
 & \cup \quad \{(q_3,i,j)z \longrightarrow (q_5,i,j)z : -8 \le i \le 7, 0 \le j \le 7\} \\
 & \cup \quad \{(q_4,i,j)z \longrightarrow \lambda z : -8 \le i,j \le 7\} \\
 & \cup \quad \{(q_5,i,j)z \xrightarrow{g} (q_6,i,j)(q_3,j,h)z : -8 \le i,j,h \le 7\} \\
 & \cup \quad \{(q_6,i,j)z \longrightarrow \lambda z : -8 \le i,j \le 7\} \\
 & \cup \quad \{(q_7,i,j)z \longrightarrow (q_8,i,i-1)z : -8 \le i,j \le 7\} \\
 & \cup \quad \{(q_8,i,j)z \longrightarrow (q_9,i,j)z : -8 \le i \le 7, -8 \le j < 0\} \\
 & \cup \quad \{(q_8,i,j)z \longrightarrow (q_{10},i,j)z : -8 \le i \le 7, 0 \le j \le 7\} \\
 & \cup \quad \{(q_9,i,j)z \longrightarrow \lambda z : -8 \le i,j \le 7\} \\
 & \cup \quad \{(q_{10},i,j)z \xrightarrow{f} (q_{11},i,j)(q_2,j,h)z : -8 \le i,j,h \le 7\} \\
 & \cup \quad \{(q_{11},i,j)z \longrightarrow \lambda z : -8 \le i,j \le 7\}
\end{array}
$$

- The state of the caller must be remembered
- The callee starts its execution with the first statement
- ⇒ Behaves as a stack
- ⇒ Runtime systems maintains call stack
- Transition rules should only be applied to the top of stack elements
- Introduce a new state z

Discussion

Observations

- Semantics defines a pushdown machine
- Only 1 state but a huge stack alphabet: $|S| = 2 \cdot 2^4 + 10 \cdot 2^4 \cdot 2^4 = 2592$ (for 32-bit integers $|S| \approx 1.8 \cdot 10^{20}$)
- The language accepted by the pushdown machine is the possible sequence of calls to f and g: $L(f((fg)^*|f(gf)^*))$

Problems

- How to model global variables?
- How to avoid the artifical state?

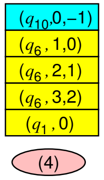
Global Variables $x_1, \ldots, x_n$

Idea: Include them into a global state.

- $Q \triangleq \{(x_1, \ldots, x_n) : x_i \in 2^{-b_i} \le x_i < 2^{b_i}$ where `int(`b_i`)` is the type of $x_i\}$
- Replaces the artificial state z
- There is only one global state, if the program has no global variables: ()

Example 5: Procedure Calls and Global Variables

```
     int(4) k;
     void f(int(4) i) {
       int(4) j;
q3 :   j=i-1;
q4 :   if (i>0) {
q5 :     f(j);
q6 :     k=k-1;
q7 :     g(j);
q8 :   }
q9 :   else k=k+1;
q10 :}
     void g(int(4) i) {
q11 :  k=k+i;
q12 :}
     { int(4) res;
q0 :   f(k);
q1 :   res=k;
q2: }
```



$$
\begin{array}{ll}
Q & \triangleq \{(k) : -8 \le k \le 7\} \\
I & \triangleq \{(k) : -8 \le k \le 7\} \\
F & \triangleq \{(k) : -8 \le k \le 7\} \\
S & \triangleq \{(q_i,x) : i \in \{0,1,2,11,12\}, -8 \le x \le 7\} \\
 & \cup \{(q_h,i,j) : 3 \le h \le 10, -8 \le i,j \le 7\} \\
R & \triangleq \quad \{(q_0,r)(k) \xrightarrow{f} (q_1,r)(q_3,k,j)(k) : -8 \le k,j,r \le z\} \\
 & \cup \quad \{(q_1,r)(k) \longrightarrow (q_2,k)(k) : -8 \le k,r \le 7\} \\
 & \cup \quad \{(q_2,r)(k) \longrightarrow (k) : -8 \le k,r \le 7\} \\
 & \cup \quad \{(q_3,i,j)(k) \longrightarrow (q_4,i,i-1)(k) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_4,i,j)(k) \longrightarrow (q_5,i,j)(k) : -8 \le i \le 0, -8 \le j,k \le 7\} \\
 & \cup \quad \{(q_4,i,j)(k) \longrightarrow (q_9,i,j)(k) : 0 < i \le 7, -8 \le j,k \le 7\} \\
 & \cup \quad \{(q_5,i,j)(k) \xrightarrow{f} (q_6,i,j)(q_3,j,h)(k) : -8 \le i,j,h,k \le 7\} \\
 & \cup \quad \{(q_6,i,j)(k) \longrightarrow (q_7,i,j)(k-1) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_7,i,j)(k) \xrightarrow{g} (q_8,i,j)(q_{11},j)(k) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_8,i,j)(k) \longrightarrow (q_{10},i,j)(k-1) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_9,i,j)(k) \longrightarrow (q_{10},i,j)(k+1) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_{10},i,j)(k) \longrightarrow (k) : -8 \le i,j,k \le 7\} \\
 & \cup \quad \{(q_{11},i)(k) \longrightarrow (q_{12},i)(k+i) : -8 \le i,k \le 7\} \\
 & \cup \quad \{(q_{12},i)(k) \longrightarrow (k) : -8 \le i,k \le 7\}
\end{array}
$$

- Semantics is a pushdown machine Π
- The language accepted is the set of sequences on calls on f and g $L(\Pi) = \{f^{n+1}g^n : n \in \mathbb{N}\}$
- ☞ $L(\Pi)$ is not a regular language, but it is context-free

---

Discussion

Observations

- Program semantics of while-programs with procedures and global variables can be represented as a pushdown machine, if all data types are finite types
- ☞ Holds for all programs without concurrency, exceptions
- However, number of states and the size of the stack alphabet is horribly large (**state explosion problem**)

Software Model Checking

Does the pushdown machine A defining the program semantics satisfy a certain property?

- Is the final state always being reached?
- ⇒ If not, the program may not terminate.
- Can the final state be reached?
- ⇒ If not, the program never terminates.
- Other properties φ can be checked by jumps:

$\vdots$
$\hat{q}$: ··· //property φ must hold
q : `if ( `$\neg\varphi$` ) goto` q
q' :···

- ⇒ Is there a path from the initial state to q'?

Abstraction

Objectives

- Reduce the number of states
- Reduce the size of the stack alphabet

Idea

Let $A \triangleq (T, Q, R, I, F, S, s_0)$ Define a pushdown machine $A' \triangleq (T', Q', R', I', F', S', s_0')$ such that $|Q'| \ll |Q|$, $|S'| \ll |S$ and there are mappings $\alpha : Q \to Q'$, $\beta : S \to S'$ satisfying the following properties:

- $\alpha(I) \subseteq I'$ and $\alpha(F) \subseteq \alpha(F')$
- If $sq \xrightarrow{a} \hat{s}\hat{q} \in R$, $a \in T \cup \{\lambda\}$ then $\beta^*(s)\alpha(q) \xrightarrow{a} \hat{s}\hat{q} \in R$

$$\begin{tikzcd}
\beta^*(s)\alpha(q) \arrow[r, "a"] & \beta^*(\hat{s})\alpha(\hat{q}) \\
sq \arrow[u, "\alpha,\beta^*"] \arrow[r, "a"] & \hat{s}\hat{q} \arrow[u, "\alpha,\beta^*"]
\end{tikzcd}$$

where $\beta^* : S^* \to S'^*$ is defined by $\beta^*(s_1 \cdots s_n) \triangleq \beta(s_1) \cdots \beta(s_n)$

Example 5: Abstraction of a Pushdown Machine

```
      int(4) k;
      void f(int(4) i) {
        int(4) j;
q3 :    j=i-1;
q4 :    if (i>0) {
q5 :      f(j);
q6 :      k=k-1;
q7 :      g(j);
q8 :    }
q9 :    else k=k+1;
q10 :}
      void g(int(4) i) {
q11 :   k=k+i; }
q12 :}
      { int(4) res;
q0 :    f(k);
q1 :    res=k;
q2 : }
```

$$q_0z \overset{f}{\Rightarrow} q_1q_3z \Rightarrow q_1q_4z \Rightarrow q_1q_5z \overset{f}{\Rightarrow} q_1q_6q_3z \Rightarrow q_1q_6q_4z$$
$$\Rightarrow q_1q_6q_5z \overset{f}{\Rightarrow} q_1q_6q_6q_3z \Rightarrow q_1q_6q_6q_4z \Rightarrow q_1q_6q_6q_5z$$
$$\overset{f}{\Rightarrow} q_1q_6q_6q_6q_3z \Rightarrow q_1q_6q_6q_6q_4z \Rightarrow q_1q_6q_6q_6q_9z \Rightarrow q_1q_6q_6q_6q_{10}z$$
$$\Rightarrow q_1q_6q_6q_6z \Rightarrow q_1q_6q_6q_7z \overset{g}{\Rightarrow} q_1q_6q_6q_8q_{11}z \Rightarrow q_1q_6q_6q_8q_{12}z$$
$$\Rightarrow q_1q_6q_6q_8z \Rightarrow q_1q_6q_6q_{10}z \Rightarrow q_1q_6q_6z \Rightarrow q_1q_6q_7z \overset{g}{\Rightarrow} q_1q_6q_8q_{11}z$$
$$\Rightarrow q_1q_6q_8q_{12}z \Rightarrow q_1q_6q_8z \Rightarrow q_1q_6q_{10}z \Rightarrow q_1q_6z \Rightarrow q_1q_7z$$
$$\overset{g}{\Rightarrow} q_1q_8q_{11}z \Rightarrow q_1q_8q_{12}z \Rightarrow q_1q_8z \Rightarrow q_1q_{10}z \Rightarrow q_1z \Rightarrow q_2z \Rightarrow z$$

$Q' \triangleq \{z\} \quad S' \triangleq \{q_0, \ldots, q_{12}\}$
$\alpha : Q \to Q', \alpha((k)) \triangleq z$
$\beta : S \to S',$
$\beta((q_i, x)) \triangleq q_i, i \in \{0, 1, 2, 11, 12\}$
$\beta((q_i, x, y)) \triangleq q_i, i \in \{3, \ldots, 10\}$

Observations: Let A' be the abstraction of the pushdown machine A defining the semantics

- $ffffggg \in L(A')$
- In this case it holds even $L(A) = L(A')$
- In general, it holds $L(A) \subseteq L(A')$
- There is only one state

Remark: A pushdown machine with one state is called a **context-free system**

$$R' \triangleq \{\, q_0z \xrightarrow{f} q_1q_3z, q_1z \to q_2z, q_2z \to z, q_3z \to q_4z, q_4z \to q_5z, q_4z \to q_9z, q_5z \xrightarrow{f} q_6q_3z,$$
$$q_6z \to q_7z, q_7z \xrightarrow{g} q_8q_{11}z, q_8z \to q_{10}z, q_9z \to q_{10}z, q_{10}z \to z, q_{11}z \to q_{12}z,$$
$$q_{12}z \to z\}$$

Getting Rid of the State z in context-free system Π

$$R' \triangleq \{\, q_0 \xrightarrow{f} q_3.q_1, q_1 \to q_2, q_2 \to \varepsilon, q_3 \to q_4, q_4 \to q_5, q_5 \xrightarrow{f} q_3.q_6, q_6 \to q_7,$$
$$q_7 \xrightarrow{g} q_{11}.q_8, q_8 \to q_{10}, q_9 \to q_{10}, q_{10} \to \varepsilon, q_{11} \to q_{12}, q_{12} \to \varepsilon\}$$

Other Definition of a Derivation Relation

- Make concatenation explicit by the operator . and let stack grow from right to left
- Specify the derivation relation $\Rightarrow$ by inference rules:

$$\frac{u \to v \in R}{u.s \Rightarrow v.s} \qquad \frac{}{u \overset{\lambda}{\Rightarrow} u} \qquad \frac{u \overset{x}{\Rightarrow} v \quad v \overset{y}{\Rightarrow} w}{u \overset{xy}{\Rightarrow} w}$$

- $L(\Pi) \triangleq \{x \in T^* : q_0 \overset{x}{\Rightarrow} \varepsilon\}$

$$q_0 \overset{f}{\Rightarrow} q_3.q_1 \Rightarrow q_4.q_1 \Rightarrow q_5.q_1 \overset{f}{\Rightarrow} q_3.q_6.q_1 \Rightarrow q_4.q_6.q_1 \Rightarrow q_5.q_6.q_1 \overset{f}{\Rightarrow} q_3.q_6.q_6.q_1 \Rightarrow q_4.q_6.q_6.q_1$$
$$\Rightarrow q_5.q_6.q_6.q_1 \overset{f}{\Rightarrow} q_3.q_6.q_6.q_6.q_1 \Rightarrow q_4.q_6.q_6.q_6.q_1 \Rightarrow q_9.q_6.q_6.q_6.q_1 \Rightarrow q_{10}.q_6.q_6.q_6.q_1 \Rightarrow q_6.q_6.q_6.q_1$$
$$\Rightarrow q_7.q_6.q_6.q_1 \overset{g}{\Rightarrow} q_{11}.q_8.q_6.q_6.q_1 \Rightarrow q_{12}.q_8.q_6.q_6.q_1 \Rightarrow q_8.q_6.q_6.q_1 \Rightarrow q_{10}.q_6.q_6.q_1 \Rightarrow q_6.q_6.q_1$$
$$\Rightarrow q_7.q_6.q_1 \overset{g}{\Rightarrow} q_{11}.q_8.q_6.q_1 \Rightarrow q_{12}.q_8.q_6.q_1 \Rightarrow q_8.q_6.q_1 \Rightarrow q_{10}.q_6.q_1 \Rightarrow q_6.q_1 \Rightarrow q_7.q_1 \overset{g}{\Rightarrow} q_{11}.q_8.q_1$$
$$\Rightarrow q_{12}.q_8.q_1 \Rightarrow q_8.q_1 \Rightarrow q_1 \Rightarrow q_2 \Rightarrow \varepsilon$$

Getting Rid of the State z in context-free system Π

Observation

The single state z is only be needed to avoid that the rewrite system changes stack symbols other than the top of the stack.

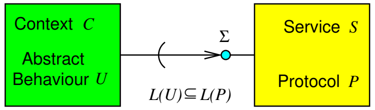
- Without z, Example 5 would allow the following derivation:
  $q_0 \Rightarrow q_1q_3 \Rightarrow q_2q_3 \Rightarrow q_3 \Rightarrow q_4 \Rightarrow q_9 \Rightarrow q_{10} \Rightarrow \varepsilon$

☞ It is necessary to introduce a empty string ε on the stack alphabet in order to distinguish it from λ (the empty string over the terminal symbols)

⇒ Change derivation relation such that only top of stack elements are considered

Discussion

- Abstraction may increase feasability of model checking
- However, the price to pay are false negatives
- Application to protocol conformance checking:



  - Let Σ be the alpabet of the pushdown machine U specifying the behaviour of C
  - For a service call $q : A.f(\cdots);\ q'$: add a transition $q \xrightarrow{f} q'$
  - Internal procedure calls are not labelled with the procedure name
  - ☞ We write $q \xrightarrow{f} q'$ instead of $qf \to q'$
  - ⇒ $L(U)$ is a superset of the sequence of service calls being executed
  - ⇒ Protocol conformance checking $L(U) \subseteq L(P)$ where $L(U)$ is a context-free language and $L(P)$ a regular language
  - ☞ False negatives are possible because of abstraction.

---

3. Concurrency

Asynchronous vs. Synchronous Procedures

Synchronous Procedures: Usual procedure execution

- Callee starts with its execution
- Caller waits until callee has been completed and returns

Aynchronous Procedures:
- Callee starts with its execution

Caller continues its execution concurrently to the callee

There is no synchronization except that the procedure contain the asynchronous procedure call only can return if the callee has been completed.

Problem

A stack cannot model the runtime behaviour of concurrent execution.

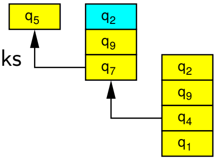
Remark

In the following, we abstract from the variable values and only consider the resulting abstract semantics.

Example 6: Asynchronous vs. Synchronous Procedures

- **Recursive Programs:** Behaviour can be modeled by pushdown machines
  - We only consider control-flow
  - ⇒ Stack frames contain program points
  - ⇒ State (stack) is a sequence of program points

Recursive and Concurrent Programs:
  - State is a cactus stack
  - ⇒ State transitions transform cactus stacks into cactus stacks



```
void main() {       async void b() {
q0: a();            q5: b();
q1: return;         q6: c();
}                   q7: return
sync void a() {     }
q2: b();            snyc void c() {
q3: c();            q8: a();
q4: return;         q9: return
```

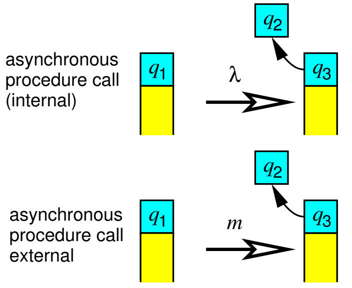
$$(q_5 \| ((q_5 \| (q_8.q_7) \| q_3).q_9.q_7) \| ((q_5 \| q_3).q_9.q_4)).q_1$$

Conclusions

- Any top of stack elements can perform state transition in any order (**interleaving semantics**)
- A cactus stack k can be represented as a process-algebraic expression
- ⇒ Behaviour of recursive and concurrent programs can be modeled by **Process Rewrite Systems**

Modelling Asynchronous Procedure Calls

Transition Rules



$q_1 \xrightarrow{\lambda} q_2 \| q_3$

$q_1 \xrightarrow{m} q_2 \| q_3$

Interleaving Semantics

Any applicable transitions rules can be executed in any order

Inference Rules: $\dfrac{u \overset{a}{\Rightarrow} v}{u \| w \overset{a}{\Rightarrow} v \| w}$ $\dfrac{u \overset{a}{\Rightarrow} v}{w \| u \overset{a}{\Rightarrow} w \| v}$

Returning From Asynchronous Procedures: $q \to \lambda$ and $q \| \varepsilon = \varepsilon \| q = q$

Modelling Semantics for Synchronous and Asynchronous Procedures

Transition Rules for Abstract Service Semantics

- Rewrite rules should include the names of services

⇒ Process rewrite rules

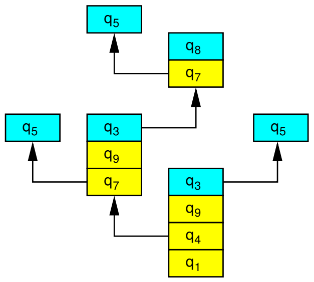
$q_i \xrightarrow{\lambda} q_j$	internal transition to next instruction
$q_i \xrightarrow{\lambda} q_k.q_j$	internal procedure call (q_k is first instruction)
$q_i \xrightarrow{a} q_j.q_k$	call of synchronous service a
$q_i \xrightarrow{\lambda} \varepsilon$	return from procedure/service
$q_i \xrightarrow{\lambda} q_k \parallel q_j$	internal call of asychronous procedure
$q_i \xrightarrow{a} q_j \parallel q_k$	call of asynchronous service a

Process Rewrite Systems (Mayr 1997)

Tuple $\Pi \triangleq (Q, \Sigma, I, \rightarrow, F)$ where

- Q is a finite set **atomic processes** (Here: program points of all components),
- Σ finite alphabet(Here: names of all services),
- $I \in Q$ **initial process**(Here: start of program execution),
- $F \subseteq PEXPR(Q)$ Menge **final Processes** (Here: ε),
- finite relation $\rightarrow \subseteq PEXPR(Q) \times (\Sigma \uplus \{\lambda\}) \times PEXPR(Q)$ (**process rewrite rules**), denoted as $e \xrightarrow{a} e'$.

Application of Process Rewrite Rules



Process Rewrite Rules can only be applied to a top stack frame of the cactus stack

Direct Derivation:
$\overset{a}{\Rightarrow} \in PEXPR(Q) \times (\Sigma \uplus \{\lambda\}) \times PEXPR(Q)$ ist defined by

$$\frac{q_i \xrightarrow{a} v \in \Pi}{q_i \overset{a}{\Rightarrow} v} \quad \frac{u \overset{a}{\Rightarrow} v}{u.w \overset{a}{\Rightarrow} v.w} \quad \frac{u \overset{a}{\Rightarrow} v}{u \parallel w \overset{a}{\Rightarrow} v \parallel w}$$

$$\frac{u \overset{a}{\Rightarrow} v}{w \parallel u \overset{a}{\Rightarrow} w \parallel v}$$

Derivation: $\overset{x}{\Rightarrow} \in PEXPR(Q) \times \Sigma^* \times PEXPR(Q)$ defined by:

$$\frac{}{u \overset{\lambda}{\Rightarrow} u} \quad \frac{u \overset{x}{\Rightarrow} v \quad v \overset{a}{\Rightarrow} w}{u \overset{xa}{\Rightarrow} w} q$$

$$(q_5 \parallel ((q_5 \parallel (q_8.q_7) \parallel q_3).q_9.q_7) \parallel ((q_5 \parallel q_3).q_9.q_4)).q_1$$

Discussion

- LHS of process rewrite rules are either atomic or have the form $q \parallel q'$ where q and q' are atomic
- ε is the identity of the sequentialn and the parallel operator
- If no parallel operator is used in the PRS, then the PRS corresponds to a pushdown system
- IF the LHS as well as the RHS of the process rewrite rules are atomic then the PRS corresponds to a finite state machine.
- $L(\Pi)$ contains all interleavings of asynchronous executions.
- Abstraction from programs to PRS can be mechanized.
- Checking conformance to a protocol p can be reduced to $L(\Pi) \overset{?}{\subseteq} L(p)$

---

4. Synchronization

Synchronization Statement `sync f;`

The execution of the synchronization statement `sync f;` continues only, if the previously called asynchronous procedure `f` has been completed.

Abstract Semantics

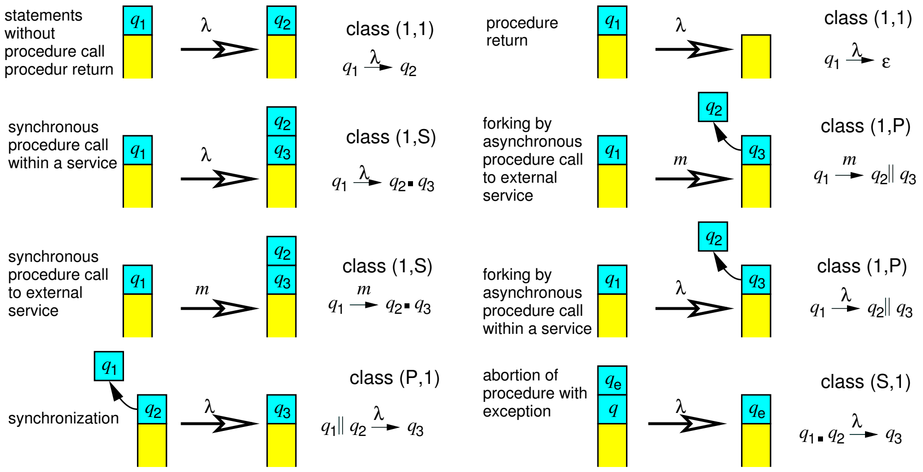
Let

- q_i : `sync f;` be a synchronization statement,
- q_{i+1} be the program point afte q_i,
- and q_j be any program point of a return statement in `f` or the last program point of `f`.

⇒ $q_i \parallel q_j \rightarrow q_{i+1}$ is an abstract semantics of the synchronization statement

5. Summary

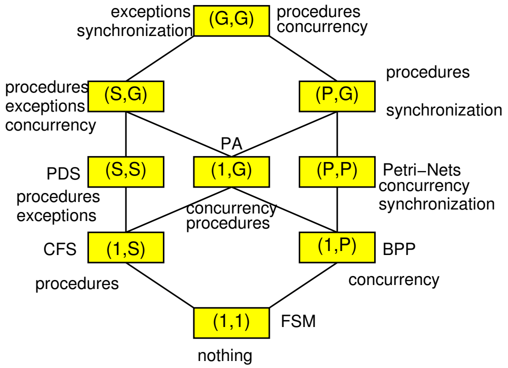
Classification of the Transition Rules within Service Implementations



The Hierarchy on Process Rewrite Systems

(x, y)-PRS:

- Each LHS belongs to x
- Each RHS belongs to y
- Class G if both S and P are allowed for x or y
- Yields a hierarchy with well-known correspondences (Mayr 1997)
- Correspondence to Programming Language Concepts (Both, Zimmermann, Franke, Heike (2010,2012))



Decidability Results

Theorem 1 (Reachability (Mayr 1997))

For PRS $\Pi = (Q, \Sigma, I, \Rightarrow, F)$ it is decidable

- whether $I \overset{x}{\Rightarrow} u$
- wether $x \in L(\Pi)$, or whether $L(\Pi) = \emptyset$

Theorem 2 (LTL-Model Checking (Mayr 1997))

Let φ be a propositional LTL-formula defining a language $L(\varphi) \subseteq \Sigma^*$.

- For $(1, 1)$-PRS, $(1, S)$-PRS,$(1, P)$-PRS, (S, S)-PRS and (P, P)-PRS it is decidable whether $L(\Pi) \subseteq L(\varphi)$
- For $(1, G)$-PRS, (S, G)-PRS, (P, G)-PRS and (G, G)-PRS it is undecidable whether $L(\Pi) \subseteq L(\varphi)$

Corollary (Protocol Conformance Checking)

For $(1, G)$-PRS, (S, G), (P, G)-PRS and (G, G)-PRS Π and a regular language L it is undecidable whether $L(\Pi) \subseteq L$