UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

Cool: A Control-flow Generator for System Analysis[1]

Volker Braun*, Jens Knoop†, and Dirk Koschützki*

* Fachbereich Informatik
Universität Dortmund
Baroper Straße 301
D–44221 Dortmund, Germany
{braun,koschuetzki}@ls5.cs.uni-dortmund.de

† Fakultät für Mathematik und Informatik
Universität Passau
Innstraße 33
D–94032 Passau, Germany
knoop@fmi.uni-passau.de

MIP-9801
Januar 1998

[1]This is an extended version of an article going to appear in Proceedings of the 7th International Conference on Compiler Construction (CC'98), Lisbon, Portugal, April 1 – 3, 1998 (cf. [4]).

Abstract

Cool is a unifying *control-flow analysis* (*CFA*) *generator* for *system analysis*, which is implemented within the MetaFrame system [17]. It uniformly supports the automatic generation of transition systems and flow graphs from terms and programs of process algebras and programming languages. Basically, it relies on "unrolling" its argument according to transition rules in the style of structural operational semantic rules. As a side-effect of the unifying view of terms and programs of process algebras and programming languages, Cool supports the automatic construction of the CFA-components required by optimizing compilers, which are usually still hand-coded. In addition to providing an adequate tool support for this important initial step of optimizer construction, the combination of Cool with data-flow analysis and optimization generators like the DFA&OPT-MetaFrame tool kit [9] renders possible the generation of complete optimizers.

Keywords

Process algebras, transition systems, concurrent and distributed systems, programming languages, flow graphs, control-flow analysis, optimizing compilers, optimizer generators.

Contents

1 Motivation

Analysis and verification procedures are typically designed for *automata*-like representations of the systems under consideration. Two prominent examples are verification procedures for concurrent and distributed systems given in terms of *process algebra terms* and analysis procedures for programs of high-level *programming languages*, which are a prerequisite for the generation of highly efficient code by optimizing compilers. In both cases the application of the relevant analysis and verification procedures relies on transforming the terms and programs of process algebras and programming languages into appropriate graphical representations, called *transition systems* and *flow graphs* in their respective contexts.

In the field of optimizing compilers this is known as *control-flow analysis* (*CFA*), performed by a control-flow analyser, later on called a CFA-component. Typically, it takes the abstract syntax tree constructed by the parser of a compiler, and transforms it into the corresponding *flow graph*, whose edge structure can be considered a concise encoding of the (dynamic) order of the program instructions at run-time. The relevance of flow graphs stems here from the fact that they are the syntactic basis of almost all important performance improving optimizations. In fact, optimizing compilers lacking a CFA-component are almost inconceivable. However, in spite of their importance, CFA-components are usually still hand-coded. The short-comings are obvious: high expenditure, low portability, and costly extensibility. This situation is the more surprising as the construction of the components of almost every other phase of compilation from lexical (cf. [13]) and syntactic analysis (cf. [6]) over data-flow analysis (cf. [1]) and optimization (cf. [18]) to code generation (cf. [7]) is supported by powerful tools. In fact, generators allow compiler writers to construct them automatically from concise specifications.

On the other hand, there have been recently proposed a number of successful approaches and tools based thereon for the automatic transfer of (CCS-like) process algebra terms into transition systems (cf. [5, 3]). In essence, the transformations realized by the *Process Algebra Compiler* (*PAC*) and the *Process Algebra Rewriting System* (*PARIS*) of [5] and [3], respectively, rely on the "unrolling" of the process algebra term according to the *transition rules* of the *process algebra* under consideration as illustrated in Figure 1.

$$\begin{array}{l} \mathtt{X} \stackrel{\text{def}}{=} \mathtt{a.Y} \\ \mathtt{Y} \stackrel{\text{def}}{=} \mathtt{b.(c.d.Y\ +\ e.0)} \end{array} \quad \xrightarrow{\text{transition rules}}$$

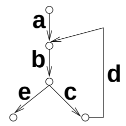

Figure 1: Transforming a process algebra term into a transition system.

From the perspective of a compiler writer, this means interpreting the effect of communication as control flow; in effect thus performing a control-flow analysis. In fact, identifying terms of a process algebra with *programs* of a *programming lan-*

guage, and transition rules of the process algebra with *rewriting rules* in the style of the *structural operational semantic* (*SOS*) rules of the programming language under consideration, the construction principle originally designed for and applied to the field of process algebras becomes directly applicable to the field of programming languages allowing us to automatically transform programs into flow graphs as illustrated in Figure 2.

```
n:=x;
z:=1;
WHILE n>1 DO
  z:=n*z;
  n:=n-1
END
```

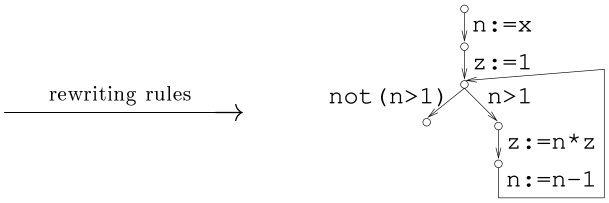

Figure 2: Transforming a program into a flow graph.

This analogy directly suggests the construction of a unifying *control-flow analysis* (*CFA*) *generator* for *system analysis*, which uniformly supports the automatic transfer of terms and programs of process algebras and programming languages into transition systems and flow graphs, respectively. As an immediate side-effect, this simultaneously yields a substantial contribution for overcoming the lack of appropriate tool support concerning the construction of the CFA-components of optimizing compilers. In fact, based on the Process Algebra Rewriting System PARIS, we developed the unifying control-flow analysis generator Cool exploiting this analogy systematically. In this article we focus on the application of the new generator to the field of programming languages. We demonstrate its generality and flexibility as a *CFA-generator* for imperative and object-oriented programming languages, which has successfully been tested within the DFA&OPT-MetaFrame project (cf. [9, 17]). Concerning CFA and the construction of optimizing compilers the major benefits of our approach are as follows:

1. *Generality:* The full range of imperative and object-oriented languages is captured.

2. *Simplicity:*

   (a) Language extensions can modularly be captured by incrementally extending the current generator specification.

   (b) The specification required for a new programming language can be constructed to a large extent in a "copy/paste"-style: adapting a specification at hand according to the "syntactic sugar" of the new language suffices.

3. *Flexibility:* The structure of the graphs generated can easily be tailored according to application-specific requirements by adapting the transition rules of the specification.

All this is achieved by means of *concise specifications* consisting of two components only:

1. a set of SO_{CFA}-rules containing essentially for every statement type (elementary and control statements) of the programming language under consideration a corresponding rewriting rule, which can usually straightforwardly be derived from the corresponding SOS-rule, and

2. a function adapting the format of abstract syntax trees provided by the parser according to the interface format of the computational cernel of the CFA-generator Cool, i.e., to transform the abstract syntax tree into prefix normal form. As this is straightforward, we do not consider this step in detail.

Structure of the Article

In the remainder of this article we demonstrate the benefits of our approach summarized above. After discussing related work in Section 2, we present our preliminaries in Section 3. Central is then Section 4, where we first illustrate the essential features of our approach considering the programming language WHILE for illustration, and subsequently demonstrate the generality and simplicity of the complete approach by showing how to incrementally extend this specification in order to capture "real world" programming languages like Oberon-2, which has been considered as a case-study in [11]. Afterwards, we focus in Section 5 on the flexibility of our approach by demonstrating as an example how to achieve specific graph formats being tailored according to application-specific requirements supporting user-customized solutions. In Section 6 we give a flavour of the system itself by presenting a couple of screenshots from a sample session, before drawing our conclusions in Section 7. The Appendix, finally, presents the SO_{CFA}-rule set required for the language WHILE, which as a side-effect illustrates the general format of these rules representing the essential part of a Cool-specification.

2 Related Work

Most closely related to our approach and the control-flow analysis generator Cool based thereon is the "Process Algebra Compiler" PAC presented in [5]. The application focus of PAC and the transition system generating component of Cool, however, and, as a consequence, their output generated is different. Whereas PAC focuses on process algebras and yields a set of functions allowing a user to transform a process algebra term into a corresponding transition system according to the interface requirements of a variety of targeted verification tools, the transition system generating component of Cool focuses on the immediate construction of graphs. Moreover, by means of the unifying view of process algebras and programming languages Cool provides additionally an important contribution to the construction of optimizing compilers. In the presentation here we concentrate on this field, i.e., the application to programming languages, the construction of optimizing compilers, and the generation of user-customized graphs satisfying application-specific requirements. In fact, this is out of the scope of PAC, and confirmatively answers a problem left for future research in [5] to which extent the SO-based construction principle can successfully be transferred and adapted to application scenarios different from that concentrated on in [5].

Other systems like MAUTO [2], ECRINS [12], or CENTAUR including its semantic component TYPOL [8], are more closely related to PAC and the transition system generating component of Cool than to Cool itself concerning the goals they are aiming at. In particular, they are not parameterized, but constructed for a fixed process algebra each. We thus do not consider them in more detail here.

3 Preliminaries

In this section we recall the similarity of transition systems and flow graphs which is the basis of the unifying view of our approach. Moreover, we sketch the functionality and the basic concepts of PARIS, the process algebra rewriting system underlying the implementation of Cool.

3.1 Transition Systems and Flow Graphs

A *transition system* $\mathcal{T}$ is a triple $(\mathcal{S}, \mathcal{A}, \rightarrow)$, where $\mathcal{S}$ is a finite set of *states* or *nodes*, $\mathcal{A}$ a set of *actions*, and $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ a set of *labeled transitions* defining the communication behavior of $\mathcal{T}$. As usual, we will write $p \xrightarrow{A} q$ instead of $(p, A, q) \in \rightarrow$.

A *flow graph* G is a triple $(N, E, \mathbf{s})$, where N is a finite set of nodes, $E \subseteq N \times N$ a set of edges, and $\mathbf{s}$ a unique *start node*, which is assumed of having no incoming edges. Following [10], we assume that the edges of G, which as usual represent the nondeterministic control flow, are labeled by the statements of the underlying program, while the nodes represent program points.

Thus, transition systems and flow graphs are both edge-labeled directed graphs.[2] Though their standard definitions do not formally coincide, one should note that a flow graph can be considered a transition system (and vice versa) by identifying the set of nodes N of a flow graph with the set of states $\mathcal{S}$ of a transition system, the set of statement patterns occurring in a flow graph with the set $\mathcal{A}$ of actions of a transition system, and the set of labeled edges E with the transition relation $\rightarrow$. This identification, which was first suggested in [15] and [16], is the key for the successful transfer of the generator principle underlying PAC and PARIS to the

[2]Flow graphs are often considered node-labeled basic-block graphs. However, edge-labeled single-instruction graphs are conceptual superior from both a theoretical and practical point of view (cf. [10]).

field of programming languages. In the following section we recall this principle considering PARIS providing the computational cernel of Cool for illustration.

3.2 PARIS: A Process Algebra Rewriting System

In essence, the process algebra rewriting system PARIS (cf. [3]) accepts as argument a pair of a process algebra specification, which fixes its syntax and semantics, and a process algebra term, which is then transformed into a labeled transition system. This transition system can either be constructed incrementally in a step-by-step mode levelwise or at once in a single big step. The specification of the semantics of a process algebra consists of a set of transition rules matching the general precondition/postcondition-pattern shown in Figure 3.

Name	$\dfrac{\text{precondition}}{\text{postcondition}}$	side-condition
Act	$\dfrac{}{\text{a.X} \xrightarrow{a} \text{X}}$	
Alt1	$\dfrac{\text{X} \xrightarrow{a} \text{X}'}{\text{X+Y} \xrightarrow{a} \text{X}'}$	
Alt2	$\dfrac{\text{Y} \xrightarrow{a} \text{Y}'}{\text{X+Y} \xrightarrow{a} \text{Y}'}$	

Figure 3: Syntax of transition rules.

Each rule is a triple consisting of a precondition, possibly extended by an additional side-condition (cf. Figure 5), a postcondition, and a name uniquely identifying it. Rules lacking a precondition like *Act* are axioms. Intuitively, the precondition of a rule is a constraint controling its applicability. The postcondition specifies the action (communication) in which a term of the process algebra satisfying the precondition pattern engages together with its behaviour thereafter. The axiom *Act*, for example, can informally be read as follows: every term matching the pattern $a.X$ can engage in an a-communication, and behaves subsequently like process X. In contrast to *Act*, the two *Alt*-rules have non-empty preconditions. Informally, rule *Alt1* can be read as follows: if a process X can engage in an a-step, and behaves like process X' thereafter, then the process $X + Y$ can engage in an a-step as well and behaves like process X' afterwards. For the remaining rule this holds analogously.

Unrolling a process algebra term or a program according to rules matching the pattern above is the key for the automatic construction of transition systems and flow graphs.

4 Generality and Simplicity

In this section we demonstrate the generality and simplicity of the generator principle underlying the implementation of Cool. To this end we consider the programming language WHILE for illustration, which is the common of imperative programming languages. It is already complex enough in order to illustrate the essence of our approach. The syntax of WHILE together with the rules defining its *structural operational semantics* (*SOS*) are recalled in Figures 4 and 5 (cf. [14]).

$$\begin{array}{lcl}
\text{Program} & = & \text{Statement } \{ \text{ ";" Statement } \} \text{ .} \\
\text{Statement} & = & \texttt{SKIP} \\
 & | & \text{Ident ":=" Expr} \\
 & | & \texttt{IF} \text{ Expr } \texttt{THEN} \text{ Statement } \{ \text{ ";" Statement } \} \\
 & & \quad \texttt{ELSE} \text{ Statement } \{ \text{ ";" Statement } \} \texttt{ END} \\
 & | & \texttt{WHILE} \text{ Expr } \texttt{DO} \text{ Statement } \{ \text{ ";" Statement } \} \texttt{ END}.
\end{array}$$

Figure 4: Syntax of WHILE.

$$\begin{array}{ll}
\text{Assign}_{\text{sos}} & \langle \texttt{x := a}, s\rangle \longrightarrow s[\texttt{x} \mapsto [\![\texttt{a}]\!](s)] \\
\text{Skip}_{\text{sos}} & \langle \texttt{SKIP}, s\rangle \longrightarrow s \\
\text{Seq1}_{\text{sos}} & \dfrac{\langle S_1, s\rangle \longrightarrow \langle S_1', s'\rangle}{\langle S_1 \texttt{ ; } S_2, s\rangle \longrightarrow \langle S_1' \texttt{ ; } S_2, s'\rangle} \\
\text{Seq2}_{\text{sos}} & \dfrac{\langle S_1, s\rangle \longrightarrow s'}{\langle S_1 \texttt{ ; } S_2, s\rangle \longrightarrow \langle S_2, s'\rangle} \\
\text{If}^{\text{tt}}_{\text{sos}} & \langle \texttt{IF } b \texttt{ THEN } S_1 \texttt{ ELSE } S_2 \texttt{ END}, s\rangle \longrightarrow \langle S_1, s\rangle \quad \text{if } [\![b]\!](s) = \mathbf{tt} \\
\text{If}^{\text{ff}}_{\text{sos}} & \langle \texttt{IF } b \texttt{ THEN } S_1 \texttt{ ELSE } S_2 \texttt{ END}, s\rangle \longrightarrow \langle S_2, s\rangle \quad \text{if } [\![b]\!](s) = \mathbf{ff} \\
\text{While}_{\text{sos}} & \begin{array}{c}\langle \texttt{WHILE } b \texttt{ DO } S \texttt{ END}, s\rangle \longrightarrow \\ \langle \texttt{IF } b \texttt{ THEN } S \texttt{ ; WHILE } b \texttt{ DO } S \texttt{ END ELSE SKIP END}, s\rangle\end{array}
\end{array}$$

Figure 5: Structural operational semantic rules of WHILE.

Basics

The SOS-rules of Figure 5 can systematically be transformed in order to arrive at the transition rules required for the CFA-generator Cool. In essence, this is achieved by dropping all parts dealing with program states and adding appropriate transition labels. Figure 6 shows the result of this transformation. It is straightforward for all rules except for the rule Seq1_{cfa}, which requires an additional side-condition in order to exclude that at some stage of the "unrolling" of a process algebra term both sequence-rules are applicable which would lead to anomalies. Note that τ can be considered like a "don't care statement" indicating that the underlying statement (assignment, skip) terminates immediately according to the SO-semantics.

$\text{Assign}_{\text{cfa}}$	$\texttt{x := a} \xrightarrow{\texttt{x := a}} \tau$	
Skip_{cfa}	$\texttt{SKIP} \xrightarrow{\texttt{skip}} \tau$	
Seq1_{cfa}	$\dfrac{S_1 \xrightarrow{a} S_1'}{S_1 \ ; \ S_2 \xrightarrow{a} S_1' \ ; \ S_2}$	if S_1' does not equal τ.
Seq2_{cfa}	$\dfrac{S_1 \xrightarrow{a} \tau}{S_1 \ ; \ S_2 \xrightarrow{a} S_2}$	
$\text{If}^{\text{tt}}_{\text{cfa}}$	$\texttt{IF } b \texttt{ THEN } S_1 \texttt{ ELSE } S_2 \texttt{ END} \xrightarrow{b} S_1$	
$\text{If}^{\text{ff}}_{\text{cfa}}$	$\texttt{IF } b \texttt{ THEN } S_1 \texttt{ ELSE } S_2 \texttt{ END} \xrightarrow{\texttt{not}(b)} S_2$	
$\text{While}_{\text{cfa}}$	$\texttt{WHILE } b \texttt{ DO } S \texttt{ END} \xrightarrow{\texttt{while}}$ $\texttt{IF } b \texttt{ THEN } S \texttt{ ; WHILE } b \texttt{ DO } S \texttt{ END ELSE SKIP END}$	

Figure 6: SO_{CFA} transition rules of WHILE.

An illustrating example: The example of Figure 7 illustrates the effect of our generator by means of the program *fac* computing the factorial of x.

Unrolling this program according to the rules of Figure 6 results in the flow graph of Figure 8(a). Note that the node labels are present only for explanatory reasons and as auxiliary information during the construction of the flow graph. Intuitively, the program fragment attached to a node denotes the fragment of the original program, which has still to be considered. Dropping these labels results in the desired flow graph in the sense of Section 3, which is shown in Figure 8(b).

```
fac: n:=x;
     z:=1;
     WHILE n>1 DO
       z:=n*z;
       n:=n-1
     END
```

Figure 7: The program *fac*.

Intuitively, the generation of the flow graph of Figure 8(a) starts with a graph consisting of a single node only, which is labeled by the program under consideration. Investigating this label the assignment `n:=x` is identified according to the rules of Figure 6 as the first statement to be executed. Unrolling the program accordingly

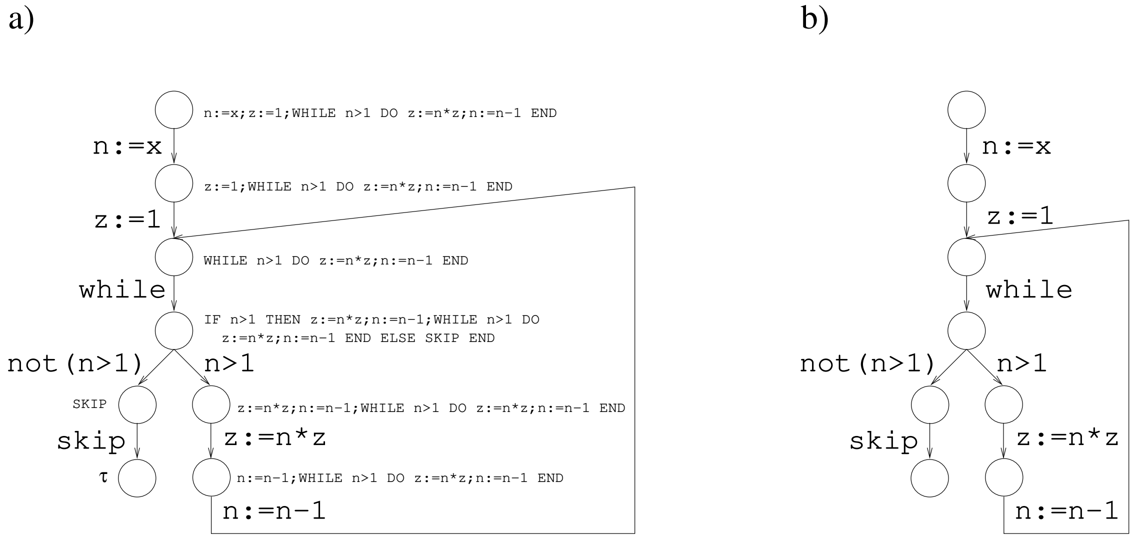

Figure 8: The flow graph of the factorial program *fac*.

results in the creation of a second node getting the program fragment remaining after executing the first assignment as its label. This derivation step results from sequentially applying rule Seq2_{cfa}, and the axiom $\text{Assign}_{\text{cfa}}$, as shown in Figure 9.

Important is then the treatment of the while-statement. Unrolling it according to the rules $\text{While}_{\text{cfa}}$ and Seq1_{cfa}, the while-statement is replaced by a "structurally equivalent" if-statement. Unrolling it as well results then in two control-flow path continuations reflecting the nondeterminism of the If-rules of Figure 6. The first path, following the edge labeled by `not(n>1)`, models the terminating execution of the while-loop. The second path reflects the execution of its body. After unrolling the two assignments of the body, a backward edge leading to the beginning of the loop is added. This is central for guaranteeing that the complete process always comes up with a finite representation of the flow graph. In essence, this is an immediate consequence of the fact that a new node is only created if there is not already a node carrying the same label.

$$\dfrac{\dfrac{}{\texttt{n:=x} \xrightarrow{\texttt{n:=x}} \tau}\ \text{Assign}_{\text{cfa}}}{\begin{array}{c}\texttt{n:=x;z:=1;WHILE n>1 DO z:=n*z;n:=n-1 END} \xrightarrow{\texttt{n:=x}} \\ \texttt{z:=1;WHILE n>1 DO z:=n*z;n:=n-1 END}\end{array}}\ \text{Seq2}_{\text{cfa}}$$

Figure 9: First derivation-step for the factorial program.

Generality and Simplicity

The language WHILE represents the common of imperative and object-oriented languages. The transition rules of Figure 6 applying to WHILE can easily be adapted to the corresponding "While"-cernel of "real world" programming languages like Pascal, Modula, C, or Oberon simply by adapting the "syntactic sugar" of the rules accordingly. Moreover, the resulting specifications can easily and incrementally be extended thereafter in order to capture the remaining control statements present in these languages. This is quite obvious for loop-statements like *repeat* and *for*, or for branch statements like *case*. It also holds for procedure calls, which can be treated like assignments on this level. In fact, only conditional and unconditional jump-statements like *goto* require some additional explanation. In essence, unrolling a goto-statement means to replace the program fragment currently considered by the program fragment following the label-statement addressed by the goto-statement. Based on this observation, it is straightforward to extend and adapt the transition rules accordingly. The simplicity and elegance of this approach carries also over to new programming languages. The transition rules required, which constitute the essential part of the input specification of the CFA-generator Cool, can be constructed to a large extent in a copy/paste-style starting from some different specification at hand. Like for WHILE and the "While"-cernel of other programming languages of interest, adapting the syntactic sugar accordingly suffices. This underlines the generality and simplicity of the overall approach.

5 Flexibility

In this section we give an example of the flexibility of our approach by demonstrating how to arrive at customized graphs being tailored for satisfying user or application specific requirements. Actually, adapting the set of transition rules is sufficient in our approach. In order to demonstrate this we reconsider the programming language WHILE, and impose the following constraint C on the shape of the flow graphs to be constructed:

$\mathcal{C}$: every edge denoted by an elementary statement of WHILE (i.e., an assignment or skip) must be followed by an unlabeled edge and vice versa.

The motivation of considering this constraint here stems from a case study performed within the DFA&OPT-MetaFrame project, where certain analyses relied on this specific graph format.

Figure 10 summarizes the SO_{CFA}-rules, which are sufficient to meet this constraint. Note that they are derived from the basic rules of Figure 6 essentially by splitting every rule introducing a new action (*seq*, *true*, *false*, ...) enforcing the unrolling of an additional edge as required by the constraint.

Figure 11(a) shows the flow graph resulting from the adapted set of transition rules for the program *fac* of Section 4. Note that after dropping the artifical "actions" like *seq*, *true*, and *false*, the resulting graph displayed in Figure 11(b) satisfies constraint C as desired.

Other constraints controling the shape of a graph to be constructed can be treated similarly. A typical example, which we considered as a part of our case

Assign$_{\text{ccfa}}$ $$\texttt{x := a} \xrightarrow{\texttt{x := a}} \tau$$

Skip$_{\text{ccfa}}$ $$\texttt{SKIP} \xrightarrow{\texttt{skip}} \tau$$

Seq1$_{\text{ccfa}}$ $$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1 \ \texttt{;}\ S_2 \xrightarrow{\alpha} S_1' \ \texttt{;}\ S_2}$$

Seq2$_{\text{ccfa}}$ $$\tau \ \texttt{;}\ S_2 \xrightarrow{seq} S_2$$ if S_2 is different from ENDIFT and ENDIFF.

IF1$_{\text{ccfa}}$ $$\texttt{IF}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{b} \texttt{IF2}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END}$$

IF2-1$_{\text{ccfa}}$ $$\texttt{IF2}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{true} S_1\ \texttt{;}\ \texttt{ENDIFT}$$

IF2-2$_{\text{ccfa}}$ $$\texttt{IF2}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{false} S_2\ \texttt{;}\ \texttt{ENDIFF}$$

ENDIFT$_{\text{ccfa}}$ $$\tau\ \texttt{;}\ \texttt{ENDIFT} \xrightarrow{endift} \texttt{SKIP}$$

ENDIFF$_{\text{ccfa}}$ $$\tau\ \texttt{;}\ \texttt{ENDIFF} \xrightarrow{endiff} \texttt{SKIP}$$

While$_{\text{ccfa}}$ $$\texttt{WHILE}\ b\ \texttt{DO}\ S\ \texttt{END} \xrightarrow{\texttt{while}}$$ $$\tau\ \texttt{;}\ \texttt{IF3}\ b\ \texttt{THEN}\ S\ \texttt{;}\ \texttt{WHILE}\ b\ \texttt{DO}\ S\ \texttt{END ELSE SKIP END}$$

IF3$_{\text{ccfa}}$ $$\texttt{IF3}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{b} \texttt{IF4}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END}$$

IF4-1$_{\text{ccfa}}$ $$\texttt{IF4}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{true} S_1$$

IF4-2$_{\text{ccfa}}$ $$\texttt{IF4}\ b\ \texttt{THEN}\ S_1\ \texttt{ELSE}\ S_2\ \texttt{END} \xrightarrow{false} S_2$$

Figure 10: SO$_{CCFA}$ transition rules of WHILE.

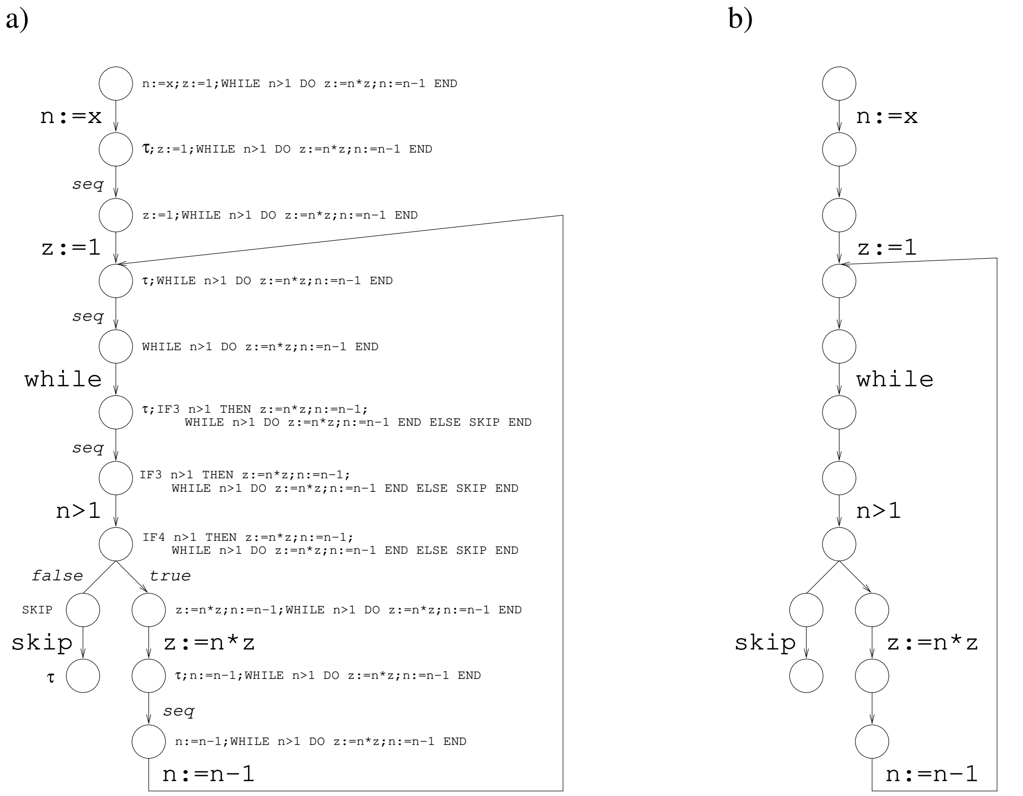

Figure 11: The flow graph of the factorial program *fac* under constraint C.

study on Oberon-2, was to replace all statements having more than two successors (i.e., *case*) into sequences of statements having (at most) two successors each. In fact, this constraint could also easily be met by adapting the SO_{CFA}-rules accordingly (cf. [11]).

6 Screen-shots from a Sample Session

In this section we give a flavour of the CFA-generator Cool by presenting a couple of screen-shots from a sample session, illustrating in particular its input required and its output generated.

In the screen-shot of Figure 12, the upper left window shows the command shell of the tool, while the lower left one displays a fragment of the SO_{CFA}-rule specification of Oberon-2. The windows on the right complement this presentation by displaying the flow graphs generated for the program shown in the central window, which is achieved by feeding the output of Cool into the automatical graph-layout component of the DFA&OPT-MetaFrame system (cf. [19]).

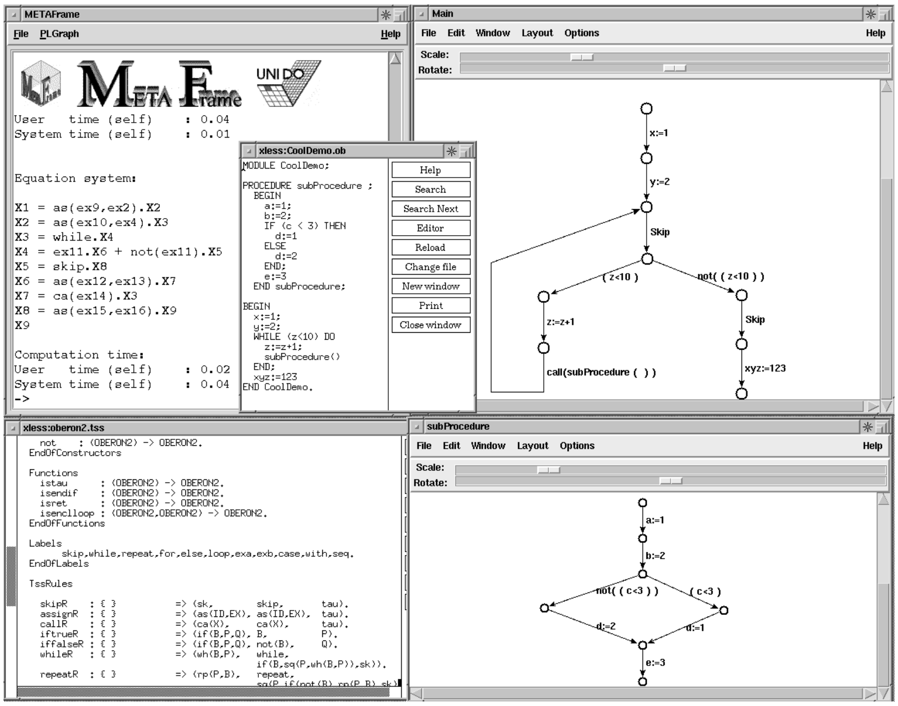

Figure 12: Cool: First screen-shot from a sample session.

The screen-shot of Figure 13 shows the effect of imposing constraint $\mathcal{C}$ introduced in Section 5 on the construction of flow graphs. Note that the flow graphs displayed in the two windows on the right are generated from the same program as in Figure 12, however, under constraint C, i.e., each control-flow edge labeled by a statement must be followed by an unlabeled statement and vice versa. As demonstrated in

Section 5 the use of a modified rule set, which is displayed in the lower left window of Figure 13, is sufficient in order to achieve this.

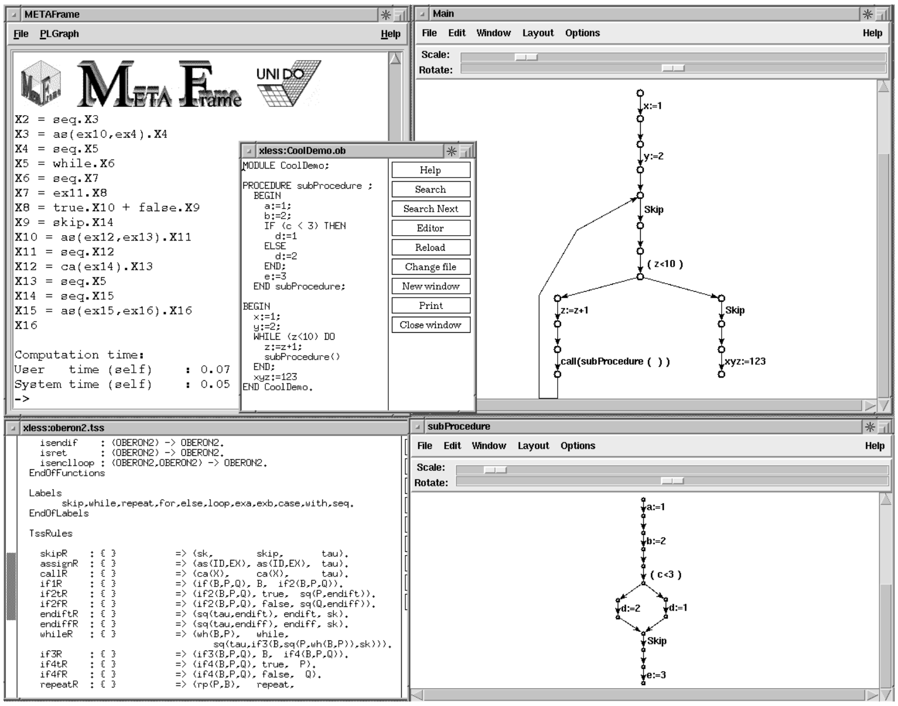

Figure 13: Cool: Second screen-shot from a sample session.

7 Conclusions

We presented the unifying *control-flow analysis generator* for *system analysis* Cool, which is implemented as part of the METAFrame system. Currently, Cool uniformly supports the automatic transfer of terms and programs of process algebras and programming languages into transition systems and flow graphs, respectively, but it is not limited to these application scenarios. Basically, the new tool results from properly enhancing the control-flow analysis generator CFAGen (cf. [11]) and its underlying process algebra rewriting system PARIS (cf. [3]). It has successfully been tested within the DFA&OPT-METAFrame project demonstrating that the generator principle it relies on captures the full range of imperative and object-oriented programming languages. Currently, we are integrating Cool and the DFA&OPT-METAFrame tool kit in order to arrive at CFA&DFA&OPT-METAFrame, a system covering the construction of the CFA- and DFA-components, and the optimizing transformations based thereon rendering thus possible the construction of *complete* optimizers.

References

[1] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Proceedings of the 2nd International Static Analysis Symposium* (*SAS'95*) (*Glasgow, UK*), Lecture Notes in Computer Science, vol. 983, pages 33 – 50. Springer-Verlag, Heidelberg, Germany, 1995.

[2] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. Technical Report RR1098, INRIA, France, 1989.

[3] V. Braun. A transition system generator for process algebras. Diplomarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, 1994. (In German).

[4] V. Braun, J. Knoop, and D. Koschützki. Cool: A control-flow generator for system analysis. To appear in *Proceedings of the 7th International Conference on Compiler Construction* (*CC'98*) (*Lisbon, Portugal*) (*April 1 – 3, 1998*), Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.

[5] R. Cleaveland, E. Madelaine, and S. Sims. A front-end generator for verification tools. In *Proceedings of the 1st International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'95*) (*Aarhus, Denmark*), Lecture Notes in Computer Science, vol. 1019, pages 153 – 173. Springer-Verlag, Heidelberg, Germany, 1995.

[6] Ch. Donnelly and R. M. Stallman. BISON – the YACC-compatible parser generator. Technical report, Free Software Foundation, 1991.

[7] C. W. Fraser and A. L. Wendt. Automatic generation of fast optimizing code generators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'88*) (*Atlanta, Georgia*), volume *23*,7 of *ACM SIGPLAN Notices*, pages 79 – 84, 1988.

[8] G. Kahn. Natural semantics. Technical Report RR601, INRIA, France, 1987.

[9] M. Klein, J. Knoop, D. Koschützki, and B. Steffen. DFA&OPT-MetaFrame: A tool kit for program analysis and optimization. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'96*) (*Passau, Germany*), Lecture Notes in Computer Science, vol. 1055, pages 422 – 426. Springer-Verlag, Heidelberg, Germany, 1996.

[10] J. Knoop, D. Koschützki, and B. Steffen. Basic-block graphs: Living dinosaurs? To appear in *Proceedings of the 7th International Conference on Compiler Construction* (*CC'98*) (*Lisbon, Portugal*) (*April 1 – 3, 1998*), Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, Germany, 1998.

[11] D. Koschützki. An so-semantics based control-flow analysis generator for imperative programming languages: Design and implementation of CFAGen. Diplomarbeit, Department of Computer Science, University of Passau, Germany, 1997. (In German).

[12] E. Madelaine and D. Verganini. ECRINS. User manual. Technical documentation, 1988.

[13] G. T. Nicol. Flex, the lexical scanner generator. Free Software Foundation, 1993.

[14] G. D. Plotkin. Structural operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981, reprinted 1991. Lecture Notes.

[15] B. Steffen. Data flow analysis as model checking. In *Proceedings of the 1st International Conference on Theoretical Aspects of Computer Software* (*TACS'91*) (*Sendai, Japan*), Lecture Notes in Computer Science, vol. 526, pages 346 – 364. Springer-Verlag, Heidelberg, Germany, 1991.

[16] B. Steffen. Generating data flow analysis algorithms from modal specifications. *Science of Computer Programming*, 21:115 – 139, 1993.

[17] B. Steffen, T. Margaria, A. Claßen, and V. Braun. The METAFrame'95 environment. In *Proceedings of the 8th International Conference on Computer–Aided Verification* (*CAV'96*) (*New Brunswick, New Jersey*), Lecture Notes in Computer Science, vol. 1102, pages 450–453. Springer-Verlag, Heidelberg, Germany, 1996.

[18] St. W. K. Tijan and J. L. Hennessy. Sharlit — A tool for building optimizers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'92*) (*San Francisco, California*), volume *27*,7 of *ACM SIGPLAN Notices*, pages 82 – 93, 1992.

[19] M. v. d. Beeck, V. Braun, A. Claßen, A. Dannecker, C. Friedrich, D. Koschützki, T. Margaria, F. Schreiber, and B. Steffen. Graphs in METAFrame: The unifying power of polymorphism. In *Proceedings of the 3rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS'97*) (*Enschede, The Netherlands*), Lecture Notes in Computer Science, vol. 1217, pages 112 – 129. Springer-Verlag, Heidelberg, Germany, 1997.

A The SO$_{CFA}$-Rules of WHILE

In this section we present the set of SO$_{CFA}$-rules for the programming language WHILE. As a side-effect this illustrates the general format of these rules constituting the essential part of a Cool-specification.

```
TSS {WHILE}
  Constructors
    tau   : ()                  -> WHILE.
    sk    : ()                  -> WHILE.
    expr  : ()                  -> WHILE.
    true  : ()                  -> WHILE.
    false : ()                  -> WHILE.
    as    : (WHILE,WHILE)       -> WHILE.
    if    : (WHILE,WHILE,WHILE) -> WHILE.
    wh    : (WHILE,WHILE)       -> WHILE.
    sq    : (WHILE,WHILE)       -> WHILE.
    not   : (WHILE)             -> WHILE.
  EndOfConstructors

  Functions
    istau : (WHILE)             ->  WHILE.
  EndOfFunctions

  Labels
    skip,while.
  EndOfLabels

  TssRules
    skipR    : {           } =>  ( sk,          skip,    tau).
    assignR  : {           } =>  ( as(X,A),     as(X,A), tau).
    iftrueR  : {           } =>  ( if(B,S1,S2), B,       S1).
    iffalseR : {           } =>  ( if(B,S1,S2), not(B),  S2).
    whileR   : {           } =>  ( wh(B,S), while,
                                   if(B,sq(S,wh(B,S)),sk)).
    seq1R    : { (S1,A,S3) } =>  ( sq(S1,S2), A,
                                   sq(S3,S2)); istau(S3)=false.
    seq2R    : { (S1,A,tau)} =>  ( sq(S1,S2),   A,       S2).
  EndOfTssRules

  RewRules
    istau(tau)       ->  true.
    istau(*)         ->  false.
  EndOfRewRules

  Guards
    wh.
  EndOfGuards
EndOfTss
```