Verification of protocol implementations by means of test generation from specifications

van Opstal, Marc F.

Award date:
1993

Technische Universiteit Eindhoven
Faculteit der Elektrotechniek
Vakgroep Digitale Systemen (EB)

Eindhoven University of Technology
Faculty of Electrical Engineering
Subfaculty Digital Systems (EB)

Verification of Protocol Implementations by means of Test Generation from Specifications

by
Marc F. van Opstal

id.nr. : 255505

Report of graduation project
Carried out from December 1992 until August 1993
In charge of prof. C.J. Koomen

TRT Le Plessis Robinson
Philips Research Laboratories Eindhoven
University of Technology Eindhoven

Summary

In order to graduate at the Eindhoven University of Technology, I have been working on automatically deriving test cases in the test language *TTCN* for protocols specified in the formal description language *LOTOS*. This work has been done during one month (December 1992) at Philips Research Laboratories in Eindhoven, the Netherlands, and eight months (January until August 1993) at TRT, a subsidiary of Philips' Product Division Communication Systems, in Le Plessis Robinson near Paris in France. My work was a continuation of the graduation project done by M. ten Hacken at TRT in 1992. The aim of my work was to improve the test environment of TRT.

In order to do this, some research is done about the various existing methods to automatically derive test sequences for formally specified protocols. The most useful methods are based on transforming the protocol to an *(Extended) Finite State Machine* (EFSM) representation and traversing in some way this machine while checking if the outputs generated by the *Implementation Under Test* (IUT) are as expected. In order to transform the LOTOS specification to an EFSM a tool called *SMILE* of the toolset *LITE* is used.

There are two major aspects of protocols to be checked, viz. the control flow of the protocol which is the base structure of a protocol and the data flow of a protocol which can be used to lead the protocol in some direction. The method used to derive test sequences for control flow is a variant of the so called *UIO*-method and is based on checking each input of the IUT and checking the state reached after checking this input. The method used to derive test sequences for data flow is a variant of the so called *Data Flow chain* method and is based on checking all sequences from a definition of a variable to the output use of a variable influenced by this definition.

A method is given how to derive test cases in TTCN from a specification in LOTOS. This methods consists of the following steps: specification of the protocol in LOTOS, construction of a LOTOS EFSM with SMILE, rewriting this EFSM in a suitable way to derive test sequences, transformation of the EFSM into an *Extended Flow Graph* (EFG), derivation of test sequences for control flow with a variant of the UIO-method, derivation of sequences for data flow sequences with the Data Flow chain method, linking and optimizing all sequences, checking executability and determining input test values and finally testing the IUT.

For the derivation of control and data flow test cases in TTCN a tool called *ProTeGe* is developed. This tool works as expected for the protocols on which it is tested but has to be tested more thoroughly. Some possible improvements of this tool are given. Also it is needed to examine how to adapt the tool to other formal description languages.

Some problems arise when applying the method(s) to the protocol test environment of TRT, but directives for possible solutions are given. These problems are that the test environment of TRT is not based on the conformance testing methods such as the UIO-method, the absence in general of formal specifications and the use of SDL instead of LOTOS in case of formal specifications, and TTCN is not used in the test environment at this moment.

Résumé

Dans le cadre de mes études à l'École Polytechnique d'Eindhoven il est nécessaire d'effectuer un stage de fin d'études concrétisant ma formation d'ingénieur. J'ai effectué le premier mois (décembre 1992) à Philips Research Laboratories en Eindhoven, et les huit autres mois (janvier 1993 jusqu'à août 1993) à TRT, une filiale de la division de produits "Communications Systems" de Philips, situé au Plessis Robinson dans la banlieue de Paris. Le travail réalisé consistait à dériver automatiquement des tests écrits dans le langage *TTCN* pour des protocoles spécifiés dans le langage formel *LOTOS*; le but étant d'améliorer l'environnement de test de TRT.

Afin de trouver des méthodes pour la dérivation automatique des tests, j'ai effectué une recherche bibliographique. Les méthodes les plus utilisables étaient basées sur la transformation d'un protocole vers une *Machine à Etat Fini Etendu* (Extended Finite State Machine, EFSM) et après traverser cette machine d'une manière donnant la possibilité de contrôler les sorties genérées par l'*Implementation Testée* (Implementation Under Test, IUT) du protocole. Afin de transformer la spécification en LOTOS vers une EFSM, un outil nommé *SMILE* est utilisé.

Les deux plus importants aspects pour contrôler les protocoles sont: le *control flow* (flux des informations), la structure de base d'un protocole, et le *data flow* (flux des données) qui permet de conduire le protocole en utilisant des variables. La méthode pour la dérivation des tests pour le control flow est une variante de la méthode nommée *Unique Input/Output (UIO) method.* Cette méthode contrôle toutes les entrées de l'IUT et contrôle l'état atteint après l'entrée. La méthode pour la dérivation automatique des tests pour le data flow est une variante de la méthode nommée *Data Flow Chain method.* Cette méthode contrôle toutes les séquences qui commencent avec une définition d'une variable et qui s'arrêtent avec l'utilisation d'une variable influencée par cette définition.

Une méthode pour la dérivation des tests en TTCN à partir d'une spécification en LOTOS est expliquée. Cette méthode se décompose en une dizaine de phases : spécification du protocole en LOTOS, construction d'une EFSM en LOTOS avec SMILE, adaptation de cette EFSM afin d'effectuer la dérivation plus facilement, transformation de la EFSM vers une *Extended Flow Graph* (EFG), dérivation des séquences de test pour le control flow avec une variante de la UIO method, dérivation des séquences de test pour le data flow avec le Data Flow Chain method, combinaison et optimisation des séquences, contrôle de l'exécutabilité, détermination des entrées pour tester les protocoles et enfin test de l'IUT.

Un outil nommé *ProTeGe* est développé pour la dérivation des tests en TTCN pour le control flow et le data flow. Cet outil marche comme prévu mais il faut le tester avec des protocoles plus réalistes. Quelques conseils pour améliorer cet outil sont donnés. La recherche de l'adaptation de cet outil pour d'autres langages formels est nécessaire.

Quand on applique les méthodes à l'environnement de test de TRT, on constate qu'il existe quelques problèmes, mais des conseils pour résoudre ces problèmes sont donnés. Ces problèmes sont : l'environnement de test à TRT n'est pas basé sur les méthodes de test comme l'UIO method, l'absence en géneral des spécifications formelles, l'utilisation de LDS au lieu de LOTOS quand des spécifications formelles sont utilisées, et la méconnaissance de TTCN dans l'environnement de test actuel.

Acknowledgments

First of all I would like to thank prof. C.J. Koomen for giving me this great opportunity of carrying out my graduation project at Philips Research Laboratories at Eindhoven and TRT near Paris. Especially being for eight months in a city like Paris, meeting all kind of people, getting to know all kinds of cultures, improving speaking all kinds of languages and visiting all kinds of interesting musea, beautiful parks and streets and really nice pubs, while working on a project that really interested me, has been one of the best experiences of my life.

Also I would like to thank Ron Koymans for following my project from Philips Research Laboratories, giving me useful advice concerning my project and correcting my report. I would also like to thank Yat Man Lau for this. At TRT, I would like to thank all people from Service Génie Logiciel, especially my daily supervisor Benoit Pinta, for their support to help getting me around in a new environment in a foreign country. Also all other trainees who made my stay at TRT a very pleasant one I would like to thank, especially François Rousselot for his efforts in showing me the French way of live. Further I would like to thank Chantal Samy for her efforts to make me feel home and making it possible for me to move to a student residency.

In the Netherlands I would like to thank all my friends and family for visiting me in Paris, for sending me all kinds of cards and mail to overcome the more difficult periods, and for making the weekends back efficiently used weekends. I especially would like to thank Sjoerd Roorda for his lots of mailed support and suggestions when having problems.

Finally I would like to thank all other nice people I met for making this time in Paris a wonderful time, and all not nice people for not meeting me.

Contents

CHAPTER 1: Introduction

The importance of telecommunication systems has grown enormously during the last decennia. Communication nowadays is done by means of television, radio, phone, computer, fax and other media. Integration of these techniques has been another issue to work on. To provide standardized communication with few errors during communication, complex communication protocols have been developed.

To deal with these very complex protocols formal description techniques (FDTs) have been developed. Examples are Estelle, SDL and LOTOS. These languages provide an unambiguous specification of the protocols.

Using some methods to traverse a state machine model of the protocol, it is possible to automatically derive test scenarios directly from the formal description. These scenarios can then be used for the validation of the implementation of a protocol. The advantage of doing this automatically is the reduction in the testing time and the reduction in errors.

From 1 December 1991 till 14 July 1992 M. ten Hacken has been working on automatically deriving test suites from formal specifications at TRT in Paris. The purpose of my project was the improvement of the algorithm developed by M. ten Hacken and the application of this improved algorithm at the test environment of TRT. This work was done with the ambition to graduate from the Eindhoven University of Technology within the faculty Electrical Engineering, subfaculty EB (digital systems).

To do this I have been working from 1 December 1992 till 1 January 1993 at the Philips Research Laboratories Eindhoven and I will work from 1 January 1993 till 1 September 1993 at TRT in Paris. The next chapter is an abstract of the work done at the Philips Research Laboratories.

CHAPTER 2: Project specification and tools

§ 2.1 Project specification

The aim of my project is:

> ***The improvement of the protocol test environment of TRT at Le Plessis Robinson by means of automatic generation of test suites from formal specifications of communication protocols.***

As told in chapter 1, this automatic generation of test cases causes a significant reduction in the duration of the testing part of the development cycle. Moreover, automatic generation can improve the quality of testing by making less faults and by being more complete.

This project is a continuation of the graduate project (title: "Using FDT LOTOS to derive tests") carried out by M. ten Hacken from 1 December 1991 till 14 July 1992. During his project [Ha92], M. ten Hacken has done research for the derivation of Extended Finite State Machines (EFSM) from the specification of protocols in LOTOS, a Formal Description Technique [ISO8807]. On the basis of this derived EFSM a method is found which derives one test suite consisting of one test trace for every input action of the Implementation Under Test. The developed tools to establish this conversion have the EFSM as input and produce the test case in the language TTCN [ISO9646-3] which is commonly used for test descriptions.

During the last years, tools have been developed which are helpful for verification, testing and simulation of protocols. One of these (sets of) tools is LITE, developed by the LOTOSPHERE consortium. Besides other things, LITE consists of a syntax and semantics checker for LOTOS and an simulator called SMILE. Among other things, SMILE can automatically derive EFSMs from the LOTOS input. These EFSMs can then be used for derivation of test cases by means of the algorithms developed by M. ten Hacken. A problem is that this algorithm puts some restrictions on the EFSM that is used as input. My work consists of improving the algorithm and extending the applicability on a more fundamental level.

At the Philips Research Laboratories I have obtained acquaintance and experience with the toolset LITE and in particular with SMILE. Also getting to know LOTOS was done at Eindhoven. This chapter consists besides the above project specification of a section about LOTOS, a section about LITE, a section about SMILE and a section about EFSMs.

§ 2.2 Specification language LOTOS

§ 2.2.1 Introduction

LOTOS (Language of Temporal Ordering Specification) is one of the two FDTs developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems, and in particular for those related to the Open System Interconnection (OSI) computer network architecture. It was developed during the years 1981-1986 and became an ISO standard in 1988 [ISO8807]. In the following sections we will give a brief overview of LOTOS. More complete introductions are given in [BoBr87], [DrChBl92] and [LoFaHa92].

In LOTOS a distributed, concurrent system is seen as a *process*, possibly consisting of a hierarchy of several sub-processes. A process is an entity able to perform *internal, unobservable actions*, and to interact with other processes, which form its *environment.* The language is split in two parts:

1) A part for the description of data structures and value expressions. This part is based on the formal theory of abstract data types. It is inspired by the abstract data type formalism "ACT ONE".

2) A part for the description of processes. This part is based on the formal theory of process algebras. It is inspired by CCS (Calculus of Communicating Systems) and CSP (Language of Communicating Sequential Processes).

First we will introduce *basic LOTOS* where processes interact with each other by pure synchronizations, without exchanging values. After that we will introduce *full LOTOS*, where processes may exchange values, or be parametrized by them. Also predicates can then be used.

§ 2.2.2 Basic LOTOS

The typical structure of a basic LOTOS process definition is as follows:

```
process proc_name[gate list] :=
      behaviour expression
where
      process definitions
endprocess
```

Reserved LOTOS words are bold and syntactic categories, that is, nonterminal symbols are italic. The names of the gates in the gate list identify the observable actions in basic LOTOS.

An essential component of a process definition is its *behaviour expression.* A *behaviour expression* is built by applying an operator to other behaviour expressions. A behaviour expression may also include *instantiations* of other processes, whose definitions are provided in the **where** clause following the expression. We will now give the complete list of basic LOTOS behaviour expressions. Symbols `B`, `B1` and `B2` stand for any behaviour expression.

• inaction `stop`

stop represents the completely inactive process.

• action prefix

- unobservable (internal) `i; B`
- observable `g; B`

This is an operator which produces an new behaviour expression out of an existing one, by prefixing the latter with an action (gate name) followed by a semicolon.

• choice `B1 [] B2`

This operator denotes a process that behaves either like `B1` or like `B2`.

• parallel composition

- general case `B1 |[g1, ... , gn]| B2`
- pure interleaving `B1 ||| B2`
- full synchronization `B1 || B2`

In the general case this operator says that a parallel composition expression is able to perform any action that either component is ready to perform at a gate not denoted

between [and], or any action that both components are ready to perform at a gate denoted between [and]. The pure interleaving operator ||| is shorthand for the situation where the set of synchronization gates is empty. The full synchronization operator || is used when the set of synchronization gates is the set of *all* gates. In this situation the two composed processes are forced to proceed in complete synchrony, except for possible internal actions.

• hiding `hide g1, ... , gn in B`

The hide operator allows abstracting from the internal functioning of a process, by hiding actions that are internal to it.

• process instantation `p[g1, ... , gn]`

A process instantation defines some other process, or itself. Other processes can be specified in the where clause.

• successful termination `exit`

`exit` is a process whose purpose is solely that of performing the successful termination action δ, after which it transforms into the dead process `stop`.

• enabling `B1 >> B2`

If `B1` terminates successfully, and not because of a premature deadlock, then the execution of `B2` is enabled.

• disabling `B1 [> B2`

This operator allows `B2` to interrupt, disable, `B1` at any point in time when `B1` is performing actions.

§ 2.2.3 Full LOTOS

In full LOTOS, it is possible to describe process synchronization involving the exchange of data values. Data structures and value expressions are defined using the abstract data type specification language ACT ONE. We will not describe ACT ONE or the definition of a specification in full LOTOS, but we will give some principles of value passing in LOTOS' behaviour part.

In full LOTOS an action consists of three components: a *gate*, a list of *events*, and an optional *predicate*. Processes synchronize their actions, provided that they name the same

gate, that the list of events are matched, and that the predicates, if any, are satisfied. An event can either *offer* (!) or *accept* (?) a value. For example, consider the following action:

```
g ?X:Nat !1 [X =< 2]
```

This is a LOTOS action that occurs at gate `g` and expects a value for `X` of sort `Nat` restricted to be less than or equal to 2, while at the same time offering the value 1.

Furthermore, a behaviour expression can be proceeded by a guard that must be true in order for the former to be enabled. For example:

```
[X =< 2] -> g !X !0 ; stop    []    [X > 2] -> g !X !1 ; stop
```

If variables are defined, they can be used as parameters for a process. Also the functionality of a process, i.e. is it likely to terminate successfully (exit) or not (noexit), is given in the process definition. A process header can then look like this:

```
process consumer [In_ele] (Msg1, Msg2:Nat): exit(Nat, Nat) :=
```

§ 2.3 Toolset LITE

LITE (LOTOS Integrated Tool Environment) is a set of tools which support the development of complex distributed and concurrent systems in the whole trajectory from specification to implementation. It was developed in the ESPRIT project LOTOSPHERE [ViPiLa92] which was started in 1989. Nowadays still new tools are developed which can be incorporated in future releases of LITE. The tools can handle full LOTOS. More information about LITE can be found in the user manual [Lot92], an introduction [Ma92] and an evaluation [Lau92].

For my project I have used only some of the tools that were available:

- A syntax checker. This tool checks whether the input file in LOTOS is syntactically all right.
- A semantics checker. This tool checks whether the input file is semantically all right.
- Symbolic simulator SMILE. This tool provides means to "unfold" the specification for close inspection of behaviour composition and is

able to generate EFSMs. This tool is described in the following section.

Besides the above tools, LITE contains tools for (structure) editing, report generation, graphical representation, verification, conversion to basic LOTOS, compilation and testing.

The main window is shown below.

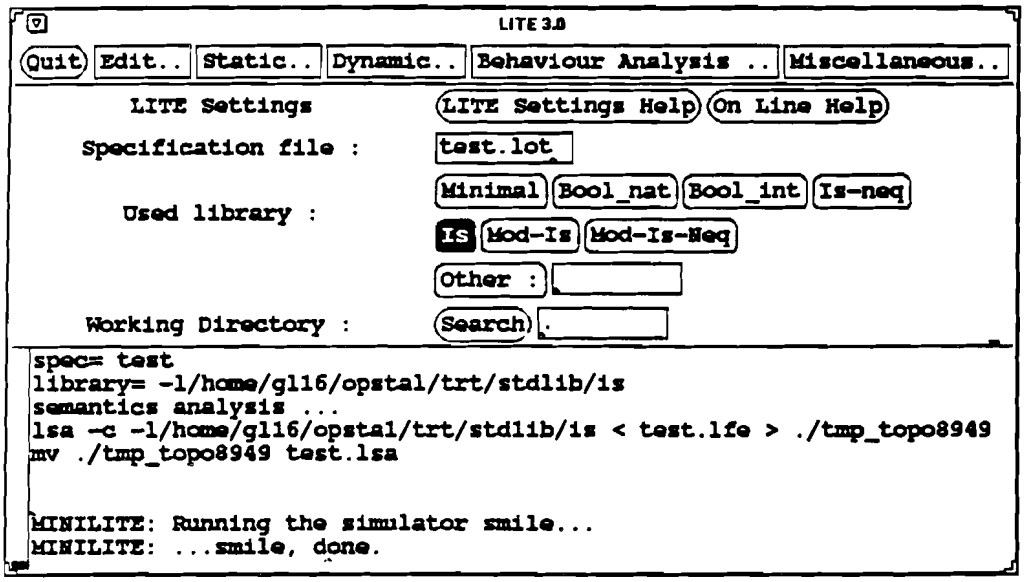

Figure 2.1 : Complete main window LITE

In the window all parameters are shown. When you start LITE, only the upper three parameters are visible. The parameter *Specification file* requires a character string representing the LOTOS input file. The extension *.lot* need not be written in this text area. For the parameter *Used library* several buttons can be chosen. Each button represents a standard defined library which is to be used during checking or simulation.

The top row of the main window contains some buttons, each indicating a certain group of actions one might want to perform on a specification. These submenus are:

- *Edit* Commands for the creation and modification of a specification.
- *Static* Syntax and semantics checker and generation of reports based on static analysis of the specification.

- *Dynamic* Tools for simulation (SMILE), compilation or transformation of LOTOS specifications.
- *Behaviour analysis* Transformation and verification tools.
- *Miscellaneous* Additional tools like a reports browser, a graphical browser and cleaning facilities.

In the bottom panel all kinds of messages concerning the execution of LITE are displayed.

§ 2.4 Symbolic simulator SMILE

In this section a small introduction to SMILE will be given. More information is found in the user manual [Lot92] and an introduction [Eer92].
When simulation is selected in the *Dynamic* menu, the symbolic simulator SMILE is started on the current specification. This specification must be correct with respect to the LOTOS static checkers. The created window, after some simulating, is shown in figure 2.2. This window consists of the following four parts:

1) an *error-line*. In this line error messages, warnings and diagnostics are displayed. This is the line directly below the title.

2) the *specification window*. In this window, the behaviour part of the input specification is shown. The simulation of the whole specification, or of a part from the specification by selecting with the mouse, can be started by clicking on the **simulate** button. The root node of the simulation tree representing the initial state of the selected expression is formed.

3) the *simulation tree window*. This window contains the (partly) unfolded simulation tree. Unfolding is done by clicking on the **unfold** button, after which the selected simulation tree will be unfolded up to the given depth (chosen in unfold option **depth**). An other important possibility is **complete EFSM**. This command makes the finite state machine corresponding to the currently simulated behaviour expression complete by expanding every state in the state machine. **Instantiate** as well as **Adt interface** give the possibility of analysis of the abstract data type part of the specification. By clicking on **write EFSM** an extended finite state machine representation of the state machine is printed to the specified file in the **output files** part. An example of such an EFSM is given in the next section. **Write tree** prints the simulation tree in the specified tree output file. The other buttons make analyzing the simulation tree easier.

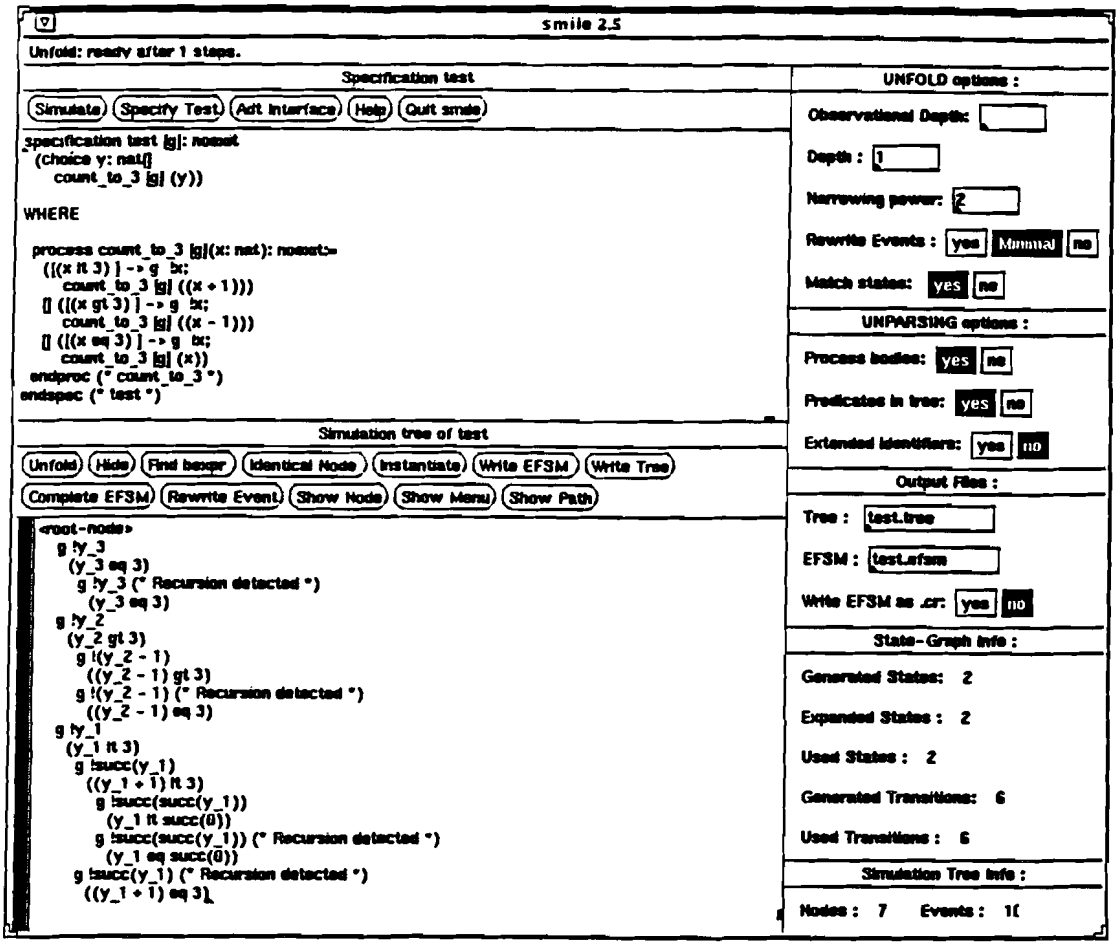

Figure 2.2 : SMILE main window

4) the *control panel.* In this panel some options for the commands can be set. The *unfold options* describe options used during the unfold-process; the *unparsing options* describe how the structures are displayed on the screen; the *output files* specify the names of the files that are created when writing the simulation results to a file; the *state-graph info* section gives some information about the number of states and transitions in the computed labelled transition system; the *simulation tree info* section gives the number of nodes and events of the simulation tree.

If during unfolding it is derived that the resulting node is equivalent to some other node, the event leading to the resulting node is annotated with `(* Analyzed elsewhere *)` or, if it is a loop, `(* Recursion detected *)`.

§ 2.5 Extended Finite State Machines (EFSMs)

It is possible to translate a finite state LOTOS specification, e.g. a LOTOS specification that can be represented by a state machine with a limited number of states, into an extended finite state machine (EFSM) by repeated unfolding of states and removing operators such as parallel composition and disable. Such an EFSM is a state machine in which the states are parametrized with some free variables. The EFSM contains the following information M, which can be seen as a 7-tuple:

$M = <J, N, V, j_0, J_F, R, f_0>$

with:
J is set of states
N is set of transitions
V is set of data declarations (variables)
$j_0 \in J$ is the initial state
J_F a subset of J is a set of final states
R is a set of rules
f_0 is the set of initial value assignments to the variables of V

R, the set of rules itself is a 5-tuple:

$r = <a, j, j', p, f>$

with:
a = an action
j = From state
j' = To state
p = precondition
f = sequence of variable assignments

SMILE is able to generate an EFSM representation out of a finite state LOTOS specification automatically. This transformation removes all parallelism from the initial specification, and transforms it into a LOTOS specification in which in the behaviour part only (generalized) choice expressions, action prefix expressions, guarded expressions and process instantiations occur. An example of such an SMILE output LOTOS EFSM is

given below. First the input specification is given. The EFSM is generated by SMILE using the button **complete EFSM** and written to a file using the button **write EFSM.**

```
specification test[g] : noexit
library basicnaturalnumber, naturalnumber, boolean
endlib

type my_nat is naturalnumber
opns
  _-_                          : nat, nat   -> nat
  1,2,3                        :            -> nat
eqns ofsort nat
  forall x, y : nat
  x - 0 = x;
  0 - x = 0;
  succ(x) - succ(y) = x - y;
  1 = succ(0);
  2 = succ(1);
  3 = succ(2);
endtype  (* my_nat *)

behaviour
(choice y:nat [] count_to_3[g](y))
where
  process count_to_3[g](x : nat) : noexit:=
        [x lt 3] -> g!x ; count_to_3 [g] (x + 1)
    []  [x gt 3] -> g!x ; count_to_3 [g] (x - 1)
    []  [x eq 3] -> g!x ; count_to_3 [g] (x)
  endproc (* count_to_3 *)
endspec  (* test *)
```

SMILE input LOTOS specification

Such an EFSM contains a set of LOTOS processes, each corresponding to a state. A transition to another state is composed of an event and a call to the next process. For example `g !x_69_0; State2 [g] (x_69_0)`. The data declarations are given in the header of a process, like `(x_69_0:nat)`. The initial state is `State1`, because that is where the behaviour starts. The set of rules is embedded in the processes. If a machine is in state j and the precondition p is true, it can be transferred to state j' under transmission or reception of action a and activating the sequence of variable assignments defined in f.

Some remarks regarding the generation of EFSMs by SMILE and the transformation from an EFSM into a TTCN test case have to be made. The generation of an EFSM from a LOTOS specification is not being optimized. Further investigation about how the SMILE output EFSM can be made smaller is desirable. A smaller EFSM will decrease the output TTCN test case. A second remark is that the algorithm developed by M. ten Hacken cannot manage variables. This restriction in his derivation method will in fact eliminate every variable in the state machine and will make every condition true.

```
SPECIFICATION test [g]: noexit(* EFSM generated by SMILE *)


BEHAVIOUR
State1 [g]
WHERE

    PROCESS State1 [g]: noexit  :=
    (choice y_70_0: nat []
         [(y_70_0 lt 3)] -> g !y_70_0;
             State2 [g] ((y_70_0 + 1)))
    [] (choice y_70_0: nat []
         [(y_70_0 gt 3)] -> g !y_70_0;
             State2 [g] ((y_70_0 - 1)))
    [] (choice y_70_0: nat []
         [(y_70_0 eq 3)] -> g !y_70_0;
             State2 [g] (y_70_0))
    ENDPROC

    PROCESS State2 [g] (x_69_0:nat): noexit  :=
    [(x_69_0 eq 3)] -> g !x_69_0;
         State2 [g] (x_69_0)
    [] [(x_69_0 gt 3)] -> g !x_69_0;
         State2 [g] ((x_69_0 - 1))
    [] [(x_69_0 lt 3)] -> g !x_69_0;
         State2 [g] ((x_69_0 + 1))
    ENDPROC

ENDSPEC
```

SMILE output LOTOS EFSM

CHAPTER 3: Conformance testing methods

§ 3.1 Introduction

In the world of communication protocols two kinds of tests are important:

• **Interoperability Testing**: Can the protocol interoperate with another protocol implementation that conforms to the same standard?

• **Conformance Testing**: Does the protocol conform to the standard?

In this chapter we will concentrate on conformance testing. Conformance testing is done to ensure a correct operation of an implementation. It consists of examining if the implementation behaves exactly as specified by the standard, i.e. if it conforms to the specification. For more information about OSI conformance standards we refer to the standard [ISO9646-1,2] which defines the guidelines for various aspects of conformance testing, such as abstract test suite specification, test description notation, test realization and test result evaluation.

There are two major methods to check whether an Implementation Under Test (IUT) has a conforming behaviour to a formal description using a FDT:

• **Generation of a testing process from the formal description**: This theory is developed for basic LOTOS [Br88] [BrAlLa90] [BrScSt87] and expanded for full LOTOS [Tre90] [Do91]. The main idea of this theory is generation of *canonical testers* that can test for conformance of an implementation with respect to a *formal notion of conformance*. An algorithm to generate such canonical testers is developed by Wezeman and is called the *CO-OP method* [We92] [WeBaLy91]. It is integrated in LITE by Alderen calling it *COOPER* [Al90]. The main problem of this theory is that the canonical tester represents complete, exhaustive testing. This means that the generation of the canonical tester does not necessarily terminate and, even if it terminates, that it may not be possible to decide upon the conformance in a finite or reasonable number of execution steps.

• **Generation of test cases as sequences of I/Os of (E)FSMs**: This theory is based on the modeling of the control portion of a protocol as a (E)FSM. A variant of this method is

used for the developed algorithm by M. ten Hacken [Ha92]. In the following section this theory will be explained including several variants.

§ 3.2 Methods based on generating testsequences for FSMs

There are several methods based on the FSM representation of a protocol. For a summary of these methods we refer to [SiLe89] [WaHu87] [BoUy91]. In all these methods a protocol specification is modeled as a deterministic FSM. A test sequence tests every state transition in the implementation of the FSM to check whether it is implemented as defined by the specification. Testing a state transition from state s_i to state s_j with input a en output o is done in three steps:

1) The FSM implementation is put into state s_i.
2) Input a is applied and the output is checked to verify that it is o, as expected.
3) The new state of the FSM implementation is checked to verify that it is s_j, as expected.

Normally these steps are followed or proceeded by a *reset step* to put the FSM in its initial state. The sequence of input symbols to test a transition or a state is called a *test subsequence*. These test subsequences are concatenated to form a *test sequence* which can be optimized by providing that no (test) subsequence of this test sequence is completely contained in any other subsequence.

As told before, there are several methods to generate protocol test sequences:

1) Transition tour method (T-method)

In this method, a test sequence (called a transition tour sequence) can be generated by simply applying random inputs to a fault-free machine until the machine has traversed every transition at least once. The major advantage of this method is its simplicity. The major disadvantage is that step 3 of the test procedure is not executed, i.e. a test sequence generated by this method only checks for the existence of transitions and does not test the tail states of the transitions. This approach has been applied to protocol testing by Sarikaya and Bochmann [SaBo82].

The problem of finding the minimum-cost *tour* (non-null sequence of consecutive transitions that starts and ends in the same state) of a *directed* and *strongly connected*

graph (for each state pair (s_i, s_j) there is a transition path going from s_i to s_j) can be seen as the *Chinese Postman Problem* (see also textbox on next page). For solving this problem several algorithms exist [AhDaLeUy88].

2) Distinguishing sequences (D-method)

A distinguishing sequence is defined as a set of inputs that generate a set of outputs different for each starting state s_i in a FSM. The distinguishing sequence is used in step 3 of the test procedure to check if the FSM is in the expected state. The major disadvantages are that many practical implementations do not have a distinguishing sequence, and secondly, the distinguishing sequences are typically very long. The method was developed by Gonenc [Go70].

3) Characterizing sequences (W-method)

In case an implementation does not have a distinguishing sequence, this method may overcome this problem. The characterizing sequences method defines "partial" distinguishing sequences. Instead of distinguishing a state s_i from every state of the FSM, a characterizing sequence distinguishes a state s_i from a subset of the remaining states. The complete set of such input sequences for a FSM is called the *characterizing set W* of the FSM. This method works better in real-life FSMs than the distinguishing sequence method. A disadvantage still is that it does not address the *controllability problem*, i.e. the ability to bring the implementation into a desired state by applying inputs. The method was developed by Chow [Ch78].

4) Unique input/output (UIO) sequence (U-method)

A *UIO sequence* for a state is an input/output behaviour that is not exhibited by any other state. An UIO sequence is calculated for every state of the FSM. Preferably the shortest UIO sequence for a state is used for checking if the FSM is in that state. So the difference with the D-method and W-method is that this sequence does not identify in which state the FSM is, but it checks whether the FSM is in state s_i. There are two important advantages for the UIO-method compared to the distinguishing or characterizing sequences. First, the length of the UIO sequences are typically smaller because the UIO sequences are a special subclass of the distinguishing sequences. The second advantage is

that almost all FSMs in practice have UIO sequences for each state while few have a distinguishing sequence. The method was developed by Sabnani and Dahbura [SaDa88].

In [ChVuIt89] this method is improved, calling it the *UIOv-method.* This method is based on the observation that although each UIO sequence is unique for the FSM constructed out of the specification, the uniqueness of UIO sequences may not hold in a faulty implementation.

Aho et al. [AhDaLeUy88] have shown that the UIO method in combination with the *Rural Chinese Postman Algorithm* (see textbox below) can be used to generate minimum-cost test sequences for graphs with the following two sufficient, but not necessary, conditions: a) the FSM specification has the reset feature (it can move into the start state from any other state with a single input operation), and b) each FSM state has the self-loop property (there exists at least one self-loop per state).
So using the UIO method in combination with the Rural Chinese Postman Algorithm addresses both the *controllability* and the *observability* (ability to observe the current state of a FSM) problem. This method is sometimes called the *SUIO-method.*

The Rural Chinese Postman Problem

The Rural Chinese Postman problem involves finding a minimum cost tour of a graph involving a selected set of edges. Normally this is done in the following way:
- Represent a protocol FSM by a directed graph G with a vertex for each state and an edge for each transition.
- Construct graph G' from G by adding an edge from vertex v_i to vertex v_k for each testsubsequence where v_k is the state reached on by applying an UIO sequence to state v_j. The edge represents the transition to be tested (v_i -> v_j) and the UIO sequence for final state v_j.
- Construct augmented graph G^* by including edges to make graph G' symmetric, i.e. there are as many incoming as outgoing edges from each vertex.
- Construct optimal length testsubsequence by finding an Euler tour of G^*, i.e. a tour that contains every edge exactly once.

In [Ur92] several variations and improvements of the above methods are discussed. The first improvement is the *MUIO-method,* based on [ShLoDa92]. The key idea of this method is based on the fact that there may be more than one minimum-length UIO sequence for a state of FSM M. Since these Multiple UIO sequences may be of different

length, and may end with different states of M, a judicious choice of UIO sequences for each state could reduce the overall length of a test sequence. Also an further improvement for the MUIO-method to derive a Rural Chinese Postman Tour in polynomial time and to provide also a minimal test sequence length instead of minimal UIO sequences, is presented.

More possible improvements are presented in [SuShLoSc91], where test sequence length is improved by using *discriminating* UIO-sequences, and in [ChAm92], where *partial* UIO-sequences (PUIO) are used when no UIO-sequences exist for a given state. These PUIO-sequences distinct a given state from a number of other states, not from all as with normal UIO-sequences. By using several PUIO-sequences for a state, this state can still be distinguished from other states although it has no UIO-sequence.

A different approach to minimalize test sequences is allowing partial overlapping of test subsequences during the construction of minimal length test sequences. In [MiPa91] it is shown that it is possible to generate optimal length test sequences which include MUIO sequences and overlapping under certain conditions. In the absence of the above mentioned conditions, a heuristic technique is used to obtain sub-optimal solutions which show significant improvement over optimal solutions without overlapping. In [ChChKe90] two new procedures are presented. The first procedure incorporates the overlapping phenomenon into a Rural Chinese Postman Problem formulation. The second one further takes into account the MUIO-sequences and heuristic techniques to generate test sequences.

§ 3.3 Including data in test sequences

Except for minimalizing lengths of test sequences other problems arise when trying to automatically derive test suites from formal specifications. For example, some research has been done in the area of fault coverage of test sequences [ChVuIt89] [SaDa88].
Another important area is including data aspects. The approach presented in the previous section only considers control flow aspects of protocols and not parameter depending and data flow aspects. Therefore, FSM-based methods are only capable of deriving tests for basic control flow aspects, and are incapable of deriving tests for the data aspects. To overcome these deficiencies of the FSM-model the EFSM-model is used (see § 2.5). Out of the EFSM modelled representation there are several ways to generate test suites:

1) Transform EFSM into FSM and apply conventional methods.

2) Transform EFSM into Control Flow Graph and Data Flow Graph and derive test sequences by conventional methods combined with parameter variation.
3) Method using axiomatic semantics approach.
4) Canonical testers.
5) Method which formalizes data flow aspects.
6) Non-fully automated methods.

Ad 1) This method is the most simple. The approach is to transform a given input specification into an equivalent FSM model during one of the initial steps, before the test generation takes place. For SDL this method is implemented by v.d. Burgt et al [BuKr-Kw90]. The problem with this approach is that the transformation from EFSM to FSM can easily lead to a very large or even infinite FSM. This state space explosion is caused by the fact that for every value of a parameter a new state is constructed.

Ad 2) This method is mostly used. The basic approach here is first transforming the EFSM into a *Control Flow Graph* (CFG) and a *Data Flow Graph* (DFG), possibly combined in one graph. The data flow is tested by variation of parameters chosen according to some method and the control flow is tested in the conventional way. Also a combined test sequence generation is possible. Because of the possible importance of these methods for my project, some articles containing variations of these method, are summarized in the next section.

Ad 3) This method is described in [WaLi92]. It is based on defining a set of *axioms*, which specify the semantics of the various types of actions of transitions. Such an axiom gives rules for transforming a precondition into a postcondition in case of an action. Testsequences are generated by checking transitions and final states of transitions and looking how conditions change by the axioms.

Ad 4) Some theoretical research is done in this area to provide canonical testers for full LOTOS, which means that also data aspects are tested [Do91] [Tre90] [We92]. Unfortunately this method is still not practically applicable.

Ad 5) The starting point of this method is that data flow aspects are given in normal English. Testing is done by formalizing these aspects and then checking them. An example is given in [Kw91].

Ad 6) These methods are based on the observation that other approaches are often suffering from the fact that the test purposes employed are unrelated to the protocol functions that should be tested. In addition, the number of tests generated is often too large to be useful in practice. The methods include parameters that can be adjusted by a test designer. An example can be found in [VeSchZo92]. The method used is based on exploiting the structure information available in formal descriptions.

§ 3.4 DFG-based methods

In this section some articles containing variations of the method to derive test sequences by transforming an EFSM into a CFG and a DFG and applying parameter variation, are summarized.

In [SaBoCe87], an Estelle EFSM is transformed into a simpler form consisting of *Normal Form Transitions* (NFTs). It can then be modeled by a CFG and a DFG. The CFG is decomposed into subtours by conventional transition tour methods. The DFG is decomposed into blocks representing functions of the protocol. Tests are designed by considering parameter variations of the input primitives of each data flow function and determining the expected outputs.

In [Ur87], this method is adjusted because of the observation that a considerable amount of effort is needed for the identification of functions and their relationships in the case of complex protocol specifications. In the new method, it is also assumed that the protocol specification is given in Estelle as a normal form specification. The specification is transformed into a graph modelling both control and data in the specification. The graph explicitly identifies the associations between definitions and usages of each variable employed in the specification. Based on this information, test sequences are constructed to cover all definition and usage pairs satisfying certain constraints.

Because [Ur87] does not explore associations between individual definition and use pairs in the given protocol specification and thus not provides full coverage of data flow aspects, the method is adjusted in [UrYa91]. Again a flowgraph modeling the flow of both control and data expressed in the given specification is constructed. In this flowgraph, definitions and uses of each context variable as well as each input and output interaction parameter employed in the specification are identified. Based on this information,

associations between each output and those inputs that influence the output are established. Then, test sequences are selected to cover each such association at least once.

In [MiPa92] a similar method is used, but now including the generation of test data and the fault coverage issues of such test sequences.

In [TriSa91], a similar approach is applied for full LOTOS. The specification is transformed into a simpler form. Then the specification is mapped to an EFSM called *chart*. The flow of data in the chart is modeled by a DFG. The DFG is decomposed into different functions that can be tested independently to give complete data flow coverage. Control flow coverage is assured by generating all the test cases from the chart. Protocol functions obtained from the DFG must be tested using the test cases obtained from the chart.

Finally, in [ChVuIt90], a hybrid technique is proposed to generate testsequences that cover both control and data flow aspects of protocols. First testsequences covering control flow are generated based on one of the variants of the UIO-method. Then data flow is tested by a method called *static data flow analysis* [RaWe85] which was first used by Ural [Ur87] for protocol testing. Ural however did not take control flow testing into consideration. Data flow paths are added to the CFG so that an *Extended Flow Graph* (EFG) is constructed. A minimal tour of the EFG is obtained, which ensures each data flow path is visited at least once. Also reducing the total testsequence by checking if data flow paths are also traversed during control flow testing and removing these flow paths, is possible.

§ 3.5 Used method of developed algorithm by M. ten Hacken

In this section a brief explanation of the method used by M. ten Hacken is given. The method is placed in the context of the rest of this chapter. For a more extensive explanation we refer to the report of his work [Ha92] or the articles on which he based his method [GuLo90] [NaSa92].

The basic idea of the method is generating one test trace for each input event of the input LOTOS EFSM as defined in § 2.5. The test traces will be given in TTCN [ISO9646-3], a commonly used language for testing. Some restrictions are made to the EFSM to ensure that a FSM-based method as explained in § 3.2 can be used to generate the test traces. These restrictions will in fact eliminate every variable in the EFSM and every condition will be guarded true. Now a test process similar to the 3 steps in § 3.2 can be applied. For

the verification of the new state in step 3 a variant of the UIO-method is used. The steps used in the method are called *initialization step, evaluation step, verification step* and *termination step.* We will now shortly explain each of these steps to test a state transition from state s_i to state s_j with input action a.

Initialization step:
This step contains the path which brings the IUT from the initial state of the machine into state s_i. To bring the state machine into state s_i, the path with the minimum amount of test events is selected.

Evaluation step:
In this step the input event to be tested is tested by sending the event to the IUT after which any response frame received, is accepted. If the received event is as expected, the procedure is continued by verifying the reached state s_j by applying the verification step. If there is no or a wrong frame received, or if an internal action is possible after sending the event, some measures are taken to continue or terminate the procedure.

Verification step:
In this step the present state is verified to see if it is s_j, as expected. In order to do this, a variant of the UIO-method is used. In this variant, to every event a variable is added which indicates if an event is *unique, half unique* or *not unique.* If an event is unique then no other equal event is used in the specification. Half unique means that the name is not unique but there is no other equal event with the same destination. In case of a not unique event, more equal events with the same destination occur in the IUT.

To find a unique path, we have to search for a unique event which is starting in the state to verify. This state can be an output event of the IUT or an input event of the IUT, which generates an output afterwards. If it is a unique event which ends up in a stable state, you didn't deliver a proof, because no reaction event from the IUT is given. If it is not possible to directly find a path like this, you can move to another state via a unique or half unique event and verify this state.

Termination step:
This step can be seen as the reset step of the test process. As with the initialization step, the path containing the minimum amount of events is selected.

CHAPTER 4: TRT and its test environment

§ 4.1 An introduction to TRT

TRT is a subsidiary of the Philips Product Division Communication Systems. This division is the most important Product Division of Philips for professional activities with 18% of the total Group sales volume (31.6 billions of francs in 1992). It employs about 14 500 people.

Philips Communication Systems is organised in six "Business Units" (BU). Its main centres are the following subsidiaries:

- **TRT** : Télécommunications Radioélectriques et Téléphoniques in France
- **PRCS** : Philips Radio Communication Systems Ltd in Great-Britain
- **PKI** : Philips Kommunikations Industrie AG in Germany
- **PTDSN** : Philips Telecommunicatie en Data Systemen BV in the Netherlands

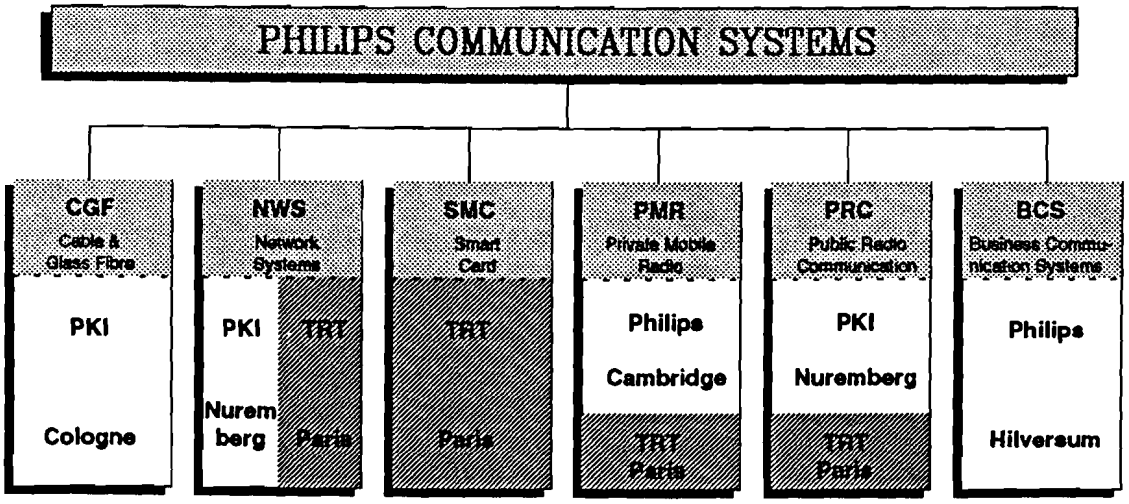

Figure 4.1 : Philips Product Division Communication Systems

Each BU has a worldwide responsibility, within its allocated activity scope, for strategy, marketing, development, manufacturing, sales, maintenance and training.

In France, TRT sells its own manufactured equipment units as well as products from other Business Units. On an international level, TRT sells its products either directly or through the Philips National Sales Organisations (about forty worldwide). The activities of TRT are implementing infrastructure for microwave radio systems, subscriber distribution, rural telephony, digital leased line-based communication networks, access networks, specific digital equipment, data transmission equipment, X25 and multiprotocol networks, Voice/Data multiplexers, secure network access and smart cards.

In 1992 its sales volume was 2415 millions of francs which is about 8% more than in 1991. About 40% of its sales volume is export. The profit of TRT was 38 millions of francs in 1992 against 15 million of francs in 1991. TRT employs about 2330 people in 6 centres in France, of which about 900 people work in Le Plessis Robinson.

§ 4.2 TRT Software Engineering Department

As of 1980, TRT started a Software Engineering Group, with the purpose to improve software design methods. In order to support software development they used some available tools like **GEODE** (a development tool based on the specification language SDL). Other tools, like **Platine 2**, were developed for this goal. Platine 2 provides a structured software development method, which follows a predefined lifecycle.

Nowadays, the Software Engineering Department (Service Génie Logiciel) consists of about 10 engineers and 5 trainees. The main project of this department is to develop a new tool called **Platine 3**, which defines an environment for software development. Platine 3 will help the developer to follow a structured path of developing software given by the design lifecycle of Platine 3. It makes integrating practically every tool in this platform possible. Platine 3 is based on the **EAST** framework which uses the **PCTE** (Portable Common Tool Environment) standard.

My graduation project took place in this Software Engineering Department of TRT, situated in Le Plessis Robinson, a suburb of Paris.

§ 4.3 Development and Test Procedure used within TRT

To explain the development procedure [Gu], the various stages used in developing software are shown in figure 4.2. *Requirements* is a description of the product which the customer wants to purchase. *System Specification* is the description of the product that is agreed upon to produce. In the next stage, *Software Specification*, the specifications of the software (DOC190) and the way how this system should be verified (DOC170) will be written. These documents are normally written in french or english and provided with lots of time diagrams for protocols. Formal specifications are rarely used.

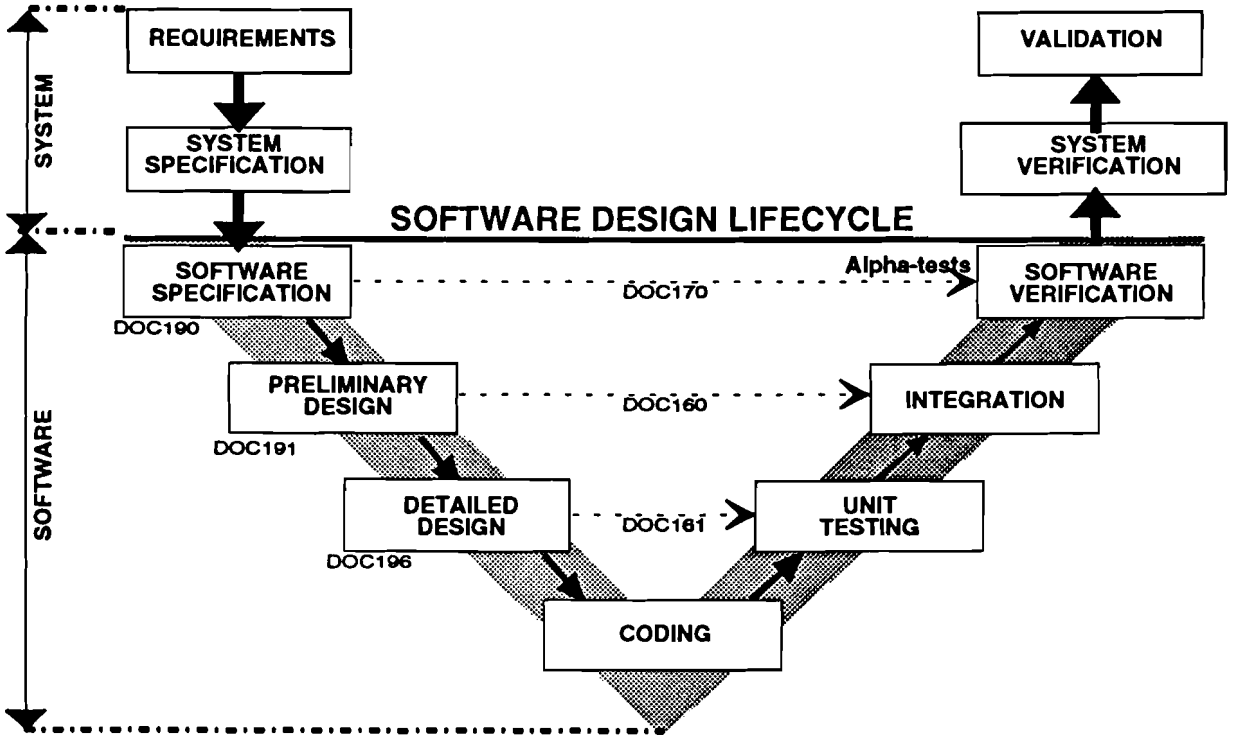

Figure 4.2 : Software Design Lifecycle

After this phase, the specification in DOC190 will be implemented. First of all a *Preliminary Design* will be made to define the architecture of the system. This architecture is usually written in SDL and/or in an executable real time system. During the stage *Detailed Design* the architecture will be refined and the algorithms to be used will be selected. During *Coding* the software will be written in a high level programming

language like **C** or **Pascal**. After coding the software units will be tested and integrated in the phases *Unit Testing* and *Integration*. In the phases *Preliminary Design* through *Integration* it is possible to use development programs like **GEODE**, based on SDL, to produce DOC160 and DOC161 automatically. This software can then be tested by tools like **TESTEUR** and **LOGISCOPE**.

The verification of the software takes place in the phase *Software Verification*; the verification of the system will be done in the phase *System Verification*. The alphatests, written in DOC170 by the verification team after design by the design team, will be transformed by the test team from a specification in time diagrams and normal french/english into executable tests in **C** or **Basic** for the protocol test environment. This environment consists of several protocol test machines such as the **Chameleon 32** (Tekelec), the **PT500** (HP), the **K11** (Siemens) and the **DA30** (WG). The reasons for doing this transformation automatically are reducing the time needed for the transformation (approximately 6 men working during one month) and the possible improvement of the criterions used for the tests. At this moment the decisions what to test are based on experience and indications by the design team. Typical protocols to be tested at the moment are X25, SDLC, BSC, VIP, RNIS, RTC, asynchronous (PAD, PAVI) and Ethernet.

The last phase is *Validation*, during which the system will be launched on the market to let the customer test the system.

§ 4.4 Applying automatic test methods to test the environment of TRT

When applying automatic test methods to the specific test environment of TRT, some problems arise. In this section these problems are explained and recommendations how these problems can be resolved are given.

- **Test environment not based on conformance testing**

This problem is the most severe. The ISO testing methodology [ISO9646-1] defines abstract test methods in terms of input/output behaviour of an Implementation Under Test (IUT). The observation and control of an IUT are carried out by an entity, called tester, which surrounds the IUT. At the abstract level, the tester can be divided into two parts - upper tester (UT) and lower tester (LT). The upper (lower) tester is responsible for the

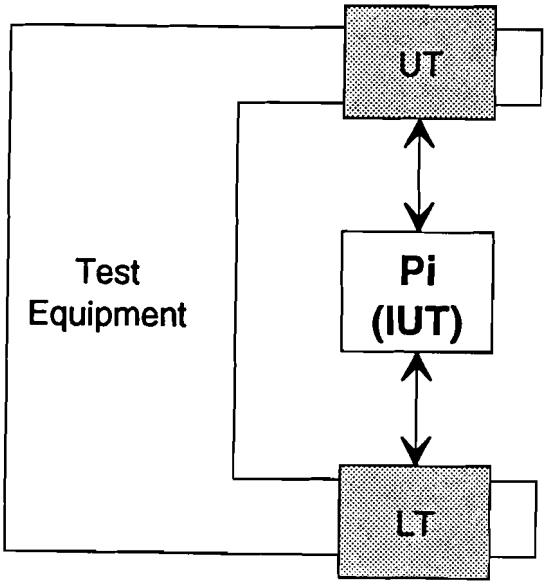

Figure 4.3 : Local Test Method

control and observation of events at the upper (lower) interface of the IUT. Two types of test methods are possible: *local*, suitable for in-house testing of products and *external*, suitable for remote testing by third parties. An example of the local test method is shown in figure 4.3.

However, the test method at the test environment of TRT is different from this test method. In the test environment of TRT (figure 4.4), the protocol *Pi* to be tested is not considered as a stand-alone protocol that can be tested separately. Instead, the IUT consists of the user protocol *Pi* and the network protocol *Pj* together. The observation points for the test equipment are the user side of the IUT *A* and the network side of the IUT *A*. At these observation points, different protocols can be working. The protocol *Pi* cannot be considered seperately because the product made by TRT consists of these two protocols together. A possible solution for this problem is to consider not only

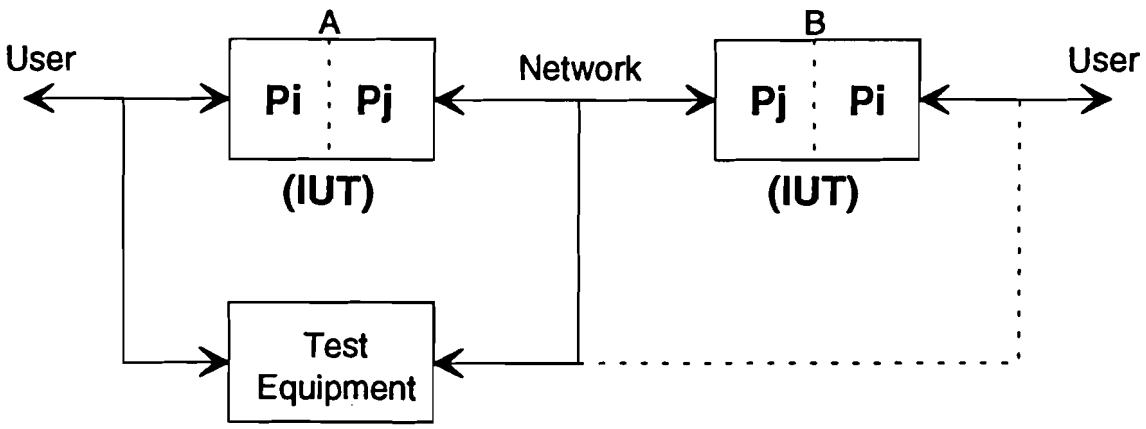

Figure 4.4 Test environment at TRT

IUT *A*, but IUT *A* combined with IUT *B*. In this situation the observation points are

different user sides of protocols exchanging information via the network. So only events for the user protocol *Pi* have to be considered.

Another possible solution is not to test the protocol in this environment, but in an earlier stage, when the protocol can still be considered seperately. In such a stage, the protocol is also tested at TRT with a test machine called **ITI**. Applying the automatic test generation method presented in this report to this stage may also be useful.

- **Few formal specifications and no LOTOS**

In about 90% of the cases, the protocol is not formally specified but in normal french/english with time diagrams to show the functionalities of the protocol. When formally specified, no LOTOS is used but SDL. In the future however, the use of formal specifications will probably grow because of increasing complexity of the protocols, an improved formal development environment and because designers will get used to specify in a formal way. To overcome the problem that SDL is used instead of LOTOS, the program to be developed will be made flexible with a seperate module for transforming the LOTOS EFSM into the EFG. In this way the program only needs another compiler to transform the SDL specification or SDL EFSM into the EFG and the program will have to be adapted for the possible differences between SDL and LOTOS like internal events or time dependent events. More research in this area is necessary.

- **No TTCN used**

In the test environment of TRT, no TTCN is used. Instead, the tests consist of indications given by the designteam and time diagrams and are transformed by the testteam into executable **C** and **Basic** tests that can be provided to the test machines like the Chameleon. However, there exists nowadays a tool that can transform the machine version of TTCN (TTCN-MP) into executable tests for the Chameleon. The output of the automatic test generation program is given in the graphical version of TTCN (TTCN-GR). This difference can be solved by using a tool for transforming TTCN-GR into TTCN-MP or to rewrite the program such that it (also) produces TTCN-MP output.

CHAPTER 5: The used method(s)

In this chapter the chosen approach for testing control and data flow is presented. Before the method is explained more thoroughly, the circumstances and considerations which led to this method, are given. In order to present a good approach, the previously given control and data flow testing methods are analyzed (§ 5.1 and § 5.2) and the program developed by M. ten Hacken is analyzed (§ 5.3).

§ 5.1 Considerations concerning control flow testing methods

First of all the main approaches to test control flow are considered.
Although the transition tour method is very simple, this method is not used because of its incapability to test the final state of a tested transaction. The other three methods have about the same fault coverage, but because the UIO-method produces in general the shortest testsubsequences and because this method is best documented and many improvements have been found, this method is preferred. Concerning which variant of the UIO-method is to be used, one has to choose between the following techniques:

- UIO/UIOv with resetting after each transition/final state testing.
- Omitting the reset for example by constructing optimal length testsequences with the Rural Chinese Postman Tour (RCPT).
- Methods using Multiple UIO sequences.
- Methods using overlap, possibly in combination with MUIO sequences and RCPT.
- Methods using Discriminating UIO sequences.
- Methods using Partial UIO sequences when no normal UIO sequences exist.

The methods using Discriminating UIO sequences are not chosen because they are to different from all the other methods and have no better performance than for example the methods using overlap. The method using Partial UIO sequences is not used because of the increased complexity to implement. This method can possibly be used in the future to improve the developed tool.

Concerning the other methods, no choice is made at this moment because it is necessary to examine first whether and how the various methods can be implemented and whether the increased complexity is worthwhile, with respect to the length of the testsequence.

§ 5.2 Considerations concerning data flow testing methods

Concerning the six main approaches for testing data aspects of protocols, mentioned in § 3.3, the following remarks can be made:

Transforming EFSMs into FSMs (approach 1) is not chosen because of the state space explosion caused by this method. Possibly this method is in some way combined with the method which formalizes data flow aspects (approach 5) at PTT Research in the Netherlands. Because of the unclearness how to use this method for the LOTOS EFSM and because of the initial stage in which the development of this method is, it is not chosen.
The method using the axiomatic semantics approach (approach 3) is not chosen because of the limited documentation (only one article), the unclearness how to apply the method to LOTOS, the deviating way of testing a state by mutants of the graph, and because also this method is in its initial stage of development. However in the future this method may well become a useful method. The method with canonical testers (approach 4) is not used because it is not practically applicable at this moment. The approach containing non-fully automated methods (approach 6), described in [VeSchZo92], is not chosen because of its limited documentation and because it is still in its initial stage. The general idea, however, to use interaction with the user instead of fully automated test derivation methods may well turn out to be useful.

These eliminations leave us with the methods which construct a Data Flow Graph or combined Control/Data Flow Graph from an EFSM and then examine data flow paths of this graph. The major drawback of this approach is that most methods use Estelle Normal Form Transitions (NFTs) as the base for constructing the graph. Therefore it is unclear whether the method can be used for the LOTOS EFSM, the output of SMILE. However, because the problem of applying a method to LOTOS or a LOTOS EFSM is a general problem for all methods, and because the LOTOS EFSM has many similarities with Estelle NFTs, it is believed that this problem will not be insolvable.

The first advantage of the approach is that there are many articles using different variations of the method. Therefore it will be easier to apply this method to the LOTOS EFSM. Also future improvements of the methods can be expected because of its rather wide use, so possibly one may improve the tool to be developed. Another advantage is that the method is sure to work also on Estelle specifications and may well be applicable

to SDL, if such a SDL specification can be transformed to an EFSM, which is probably the case.

Which variant of the approach is to be used depends mainly on how the method can be applied to the LOTOS EFSM. Probably a mixture of elements from the several articles will be used. For example, in [TriSa91] the chart representation may be helpful, whereas the extension of testing also input/output parameters [UrYa91] may be useful too. The approach in [SaBoCe87] will probably not be used because of its amount of effort needed to identify functions and their relationships in the case of complex protocol specifications.

The last remarks concern the hybrid technique proposed in [ChVuIt90]. This approach has several advantages compared to directly applying the forementioned DFG-based methods. Contrary to these methods, which hardly take control flow into consideration, the hybrid method takes both control and data flow in consideration, although principally separated. The advantages of this approach are the following:

- Separation between control and data flow testing is more clear, which makes it easier to apply several methods and to implement.
- Latest improvements of testing methods will be easier to be used for updating the program to be developed.
- Length of the final testsequence will probably not very long because besides possible use of (sub)optimal methods, the final testsequence may possibly be reduced by removing control flow paths that are already contained in the data flow paths.

In summary, one can say that this method is very flexible and open for improvements. Taking these advantages in mind, this approach is preferred.

§ 5.3 Method and developed tool of M. ten Hacken

In this section the method and developed tool by M. ten Hacken [Ha92] are analyzed in order to see if it is possible to use this tool/method or parts of this tool/method for the tool that can test control and data flow.

A main problem M. ten Hacken was confronted with was the absence of the tool SMILE during almost his total graduation period. Therefore it was unclear which expressions were possible in what way in the LOTOS EFSM. The result is the incapability of his tool to handle several constructions like:

- The *choice* operator: `(choice x:nat [] [(x lt 3)] -> g!x; State2[g] ((x)))`
- Expressions with a combination of several variables, constants and brackets: `g!x; State2[g] ((x - succ(0)))`
- Constructions with several conditions for one action: `[(x lt 3)] -> [(x gt 1)] -> g!x; State2[g]`
- Constructions with several actions at one transition: `g!x!x; State2[g]`
- Constructions with several types of internal actions: `(*middle !x*) i; State2[g]` **or** `(*exit*) i; State2[g]`
- Conditions for variables after input: `g?x:sort [(valid(x))] ;`

Also several data constructions are not considered because his tool was developed to test only control flow.

Probably because of lack of time, the tool is not well documented and not provided with lots of comment. If we should use this tool, this should be adjusted because of the necessity to make the tool well understood by its users.

Concerning the method used by his tool, it may be worthwhile to examine if improved UIO-methods can be used, as also proposed by M. ten Hacken. While examining the working of the improved method doubts have risen whether the tool correctly handles all internal actions. Little has been written about how to handle internal actions.

Of course there are also several advantages to the tool. First of all it has already been implemented, saving much development time. Secondly it has a simple way of transforming the testsequences to TTCN. Also a general approach how to interpret the LOTOS EFSM is given.

Taking the advantages and disadvantages in mind, probably the best approach is to rewrite the tool for handling all constructions of the LOTOS EFSM including data constructions, testing the control flow with a similar or improved (MUIO/RCPT/overlap) method and testing data flow with a static data flow analysis method. Principally the same method is used to traverse the graph and constructing a TTCN testsuite out of this traversal. Improvements and adaptations are applied whenever necessary and possible. In the next sections these methods will be explained more elaborately.

§ 5.4 Automatic test generation approach

In this section a survey will be given of the methodology used to generate tests automatically. As told before the best approach is to split the methodology into different parts. In this way the program can easily be adapted for other input or output languages and other methods to derive test sequences. In the next sections the various elements of the approach will be looked at.

The approach can be divided into the following steps:

1) Specifation of the protocol in LOTOS
No automatic test generation can take place without a formal specification. LOTOS is chosen because of the availability of a toolset to derive an EFSM (step 2) and to check executability and to determine test values (step 7).

2) Construction of a suitable LOTOS EFSM
Automatic test generation methods are generally based on a representation of a protocol as a EFSM (chapter 3). The construction of a LOTOS EFSM out of a specification with full LOTOS can be done with the tool SMILE as explained in § 2.4. To reduce the amount of test sequences and to provide a LOTOS EFSM more suitable to derive test sequences this SMILE EFSM has to be rewritten manually by the test person or designer. More about these problems is told in § 5.5.

3) Transformation of the LOTOS EFSM into an EFG
This is the first step that has to be implemented. This step is necessary to adapt the EFSM into a structure well suited for the derivation of the test sequences and to dispose of redundant information (for example complete predicates). This EFG can also be used as the basic structure (some adaptations may be necessary) when not using SMILE output as the input EFSM. This step will be elucidated in § 5.6.

4) Derivation of test sequences for Control Flow with variant of UIO-method
This also has to be implemented. Some variant of the UIO-method will be used to derive the test sequences. This method will be explained in § 5.7.

5) Derivation of test sequences for Data Flow with adapted IO-df-chain method
The third step to be implemented. The method will be explained in § 5.8.

6) Linking and optimizing all Control and Data Flow test sequences
Several methods can be used to link the test sequences. Because of probable lack of time only the reset method or the direct link method will be implemented. Also optimizing the final test sequence by combining the same test paths for different aspects of the protocol will be left for the future. The linking will be explained in § 5.9.

7) Checking executability and determining input test values
This step may also take place before linking the test sequences. In this step the executability of the test sequences will have to be tested and the test values that will be provided to the IUT will have to be determined. Non-executability can for example happen because condional transitions are not taken in consideration. SMILE can be used to help testing executability and determining test values. Some more explanation can be found in § 5.11.

8) Testing IUT with executable test sequences and test values
This is the final step. The tests will be provided to the IUT and the outputs are checked.

Because of the considerations in the previous section, the program will be rewritten but useful elements of the program and approach of M. ten Hacken will be used. An example of this is the transformation of the test sequences into TTCN. This will be explained in § 5.10.

§ 5.5 Constructing a suitable LOTOS EFSM

In order to explain the various phases more clearly, an example SMILE output LOTOS EFSM is written. This is not an EFSM of an existing protocol, but contains the most typical structures of SMILE output LOTOS EFSMs.

As can be seen in this example LOTOS EFSM (see text box), reception and transmission of specific frames can be represented by the combination of a transmission/reception of a general frame and selection predicates for this frame. For example, consider the following frame reception:

```
L ?fr_1: framesort [(fr_1 eq DM)]
```

The only frame value which can be generated from the selection predicate is a DM frame. Therefore this event can be rewritten to `L ?DM`. This reduces the amount of test sequences

```
SPECIFICATION example [L]: noexit

BEHAVIOUR
State1 [L]
WHERE

      PROCESS State1 [L]: noexit :=
            L ?fr_1: framesort [(fr_1 eq DM)]; State2 [L] ((0))
      ENDPROC

      PROCESS State2 [L] (X:nat): noexit :=
            (choice fr_2: framesort []
                  [(fr_2 eq SABM)] -> [(X <> 2)] -> L !fr_2; State3 [L] (X))
            [] (choice fr_3: framesort []
                  [(fr_3 eq DM)] -> [(X = 2)] -> L !fr_3; State1 [L])
      ENDPROC

      PROCESS State3 [L] (Y:nat): noexit :=
            L ?fr_4: framesort [(fr_4 eq DISC)]; State4 [L]
            [] i; State2 [L] ((Y+1))
      ENDPROC

      PROCESS State4 [L]: noexit :=
            (choice fr_5: framesort []
                  [(fr_5 eq UA)] -> L !fr_5; State1 [L]
      ENDPROC

ENDSPEC
```

Example SMILE output LOTOS EFSM

for the data flow because the variable `fr_1` is not considered any more. Another advantage is that the determination of UIO sequences will be better possible. If general frames are used, two general frames `fr_1` and `fr_2` are always considered distinctive although they might have equal values. For UIO sequences this means that the two reception sequences `L ?fr_1` and `L ?fr_2` are considered to be *unique* which well might not be the case. While testing the IUT this means that manually generated frames must be provided to the IUT. A difficulty at doing this is that complex selection predicates admit that more than one frame value appears. When this is the case, suitable frame values must be selected or the expression must be left intact and possible faults in the UIO sequences must be taken into consideration. Normally suitable frame values can be chosen.

The case is a little bit different when rewriting transmission of frames. For example, take the following frame transmission:

```
(choice fr_2: framesort []
      [(fr_2 eq SABM)] -> [(X <> 2)] -> L !fr_2
```

This can be rewritten into `[(X <> 2)] -> L !SABM`. This has the same advantages as the ones mentioned for the reception of frames, i.e. reduction of the number and complexity of test sequences for Data Flow and improved possibility to derive UIO sequences. The

difference however is that the frames must be generated by the IUT and not by an external person. The consequence is that in the case of more possible frame values these values cannot be selected but have to be tested all. In this case, the transmission of the general frame must be rewritten into parallel transmission of the possible values. For determining the possible values, also in the case of reception, the **instantiate** tool of SMILE can be used. This tool generates possible values for variables with selection predicates. In the case of complex frames, it is recommended to replace the general frame by the action (SABM, DM, etc.) and to place other predicates concerning the frame behind the action as a selection predicate. For example:

```
(choice fr_2: framesort []
      [(control(fr_2) eq SABM)] -> [valid(fr_2)] -> L !fr_2
```

can be replaced by `L !SABM [valid(SABM)]`. In this way the variables to be tested are restricted to the important variables and the UIO sequences can be determined properly.

The rewritten example LOTOS EFSM is given in the next text box.

```
SPECIFICATION example [L]: noexit

BEHAVIOUR
State1 [L]
WHERE

       PROCESS State1 [L]: noexit :=
              L ?DM; State2 [L] ((0))
       ENDPROC

       PROCESS State2 [L] (X:nat): noexit :=
              [(X <> 2)] -> L !SABM; State3 [L] (X)
              [] [(X = 2)] -> L !DM; State1 [L]
       ENDPROC

       PROCESS State3 [L] (Y:nat): noexit :=
              L ?DISC; State4 [L]
              [] i; State2 [L] ((Y+1))
       ENDPROC

       PROCESS State4 [L]: noexit :=
              L !UA; State1 [L]
       ENDPROC

ENDSPEC
```

Rewritten example LOTOS EFSM

§ 5.6 Transformation of the LOTOS EFSM into an EFG

In order to be able to derive test sequences for both the Data and Control Flow of a protocol, the rewritten LOTOS EFSM will be transformed to an *Extended Flow Graph* (EFG). This is a flowgraph $G = (V,E)$ where V is a set of nodes representing control flow states and possible actions and E is a set of edges representing the flow between nodes. It is called *extended* because not only the control flow but also data flow is considered.

The basic nodes of the EFG are the control flow nodes (*f-nodes*), one for each process of the LOTOS EFSM. Each event of the EFSM is also mapped on a node. In this way the following nodes are constructed:

- an input node (*i-node*) for each input event (events containing '?'),
- an output node (*o-node*) for each output event (events containing '!' and events consisting of only ports),
- an internal event node (*x-node*) for each internal event *i* of the specification.

Directed edges are constructed from a control flow state to i-nodes, o-nodes and x-nodes which represent actions in the process represented by the f-node. Also directed edges are constructed from these action nodes to f-nodes representing the next state after the action. In case of a *choice*-construction such as

```
(choice fr_2: framesort []
      [(fr_2 eq SABM)] -> L !fr_2
```

a choice state (*c-node*) is constructed, a directed edge is formed from the control flow node to this c-node and an edge is constructed from the c-node to the action node, here an o-node. The last node constructed is a start node (*s-node*) representing the *BEHAVIOUR*-part of the specification. This node is necessary to include the initial calling of the first process which might contain parameter values for the first process.

In order to be able to derive test sequences for the Data Flow, each variable occurrence in the specification is represented in the EFG and classified as being a definition, computational use, or predicate use which are referred to as *def*, *c-use* and *p-use*, respectively. We use the following convention to identify defs, c-uses and p-uses of each variable in G:

a) A process header `PROCESS State2 [L] (`X_1`:sort,...,`X_n`:sort)` in a f-node contains defs of $X_1, \ldots, X_n$.

b) An input action `port ?X: sort` in a i-node contains a def of X.

c) An output action `port !X` in a o-node where X is a defined variable contains a c-use of X.

d) A *choice*-construction `choice X: sort` in a c-node contains a def of X.

e) A call to a next state `; Statei [port] (`$f_1(X_1,\ldots,X_n)$`, ... ,` $f_2(X_1,\ldots,X_n)$`)` where $f(X_1,\ldots,X_n)$ represents a function of defined variables, contains for each parameter of the called state (the defs in the header, see a)) c-uses of the used variables. For example, `; State3 [port] (X, X*Y+2)` contains a c-use of variable X for the first parameter of state 3 and c-uses of X and Y for the second parameter of state 3.

f) A predicate `[`$p_1(X_1,\ldots,X_n)$`] -> ... -> [`$p_2(X_1,\ldots,X_n)$`] ->` in a edge where $p(X_1,\ldots,X_n)$ represents a predicate contains p-uses of $X_1,\ldots,X_n$.

Figure 5.1 shows the EFG of the example LOTOS EFSM including definitions and uses of the variables.

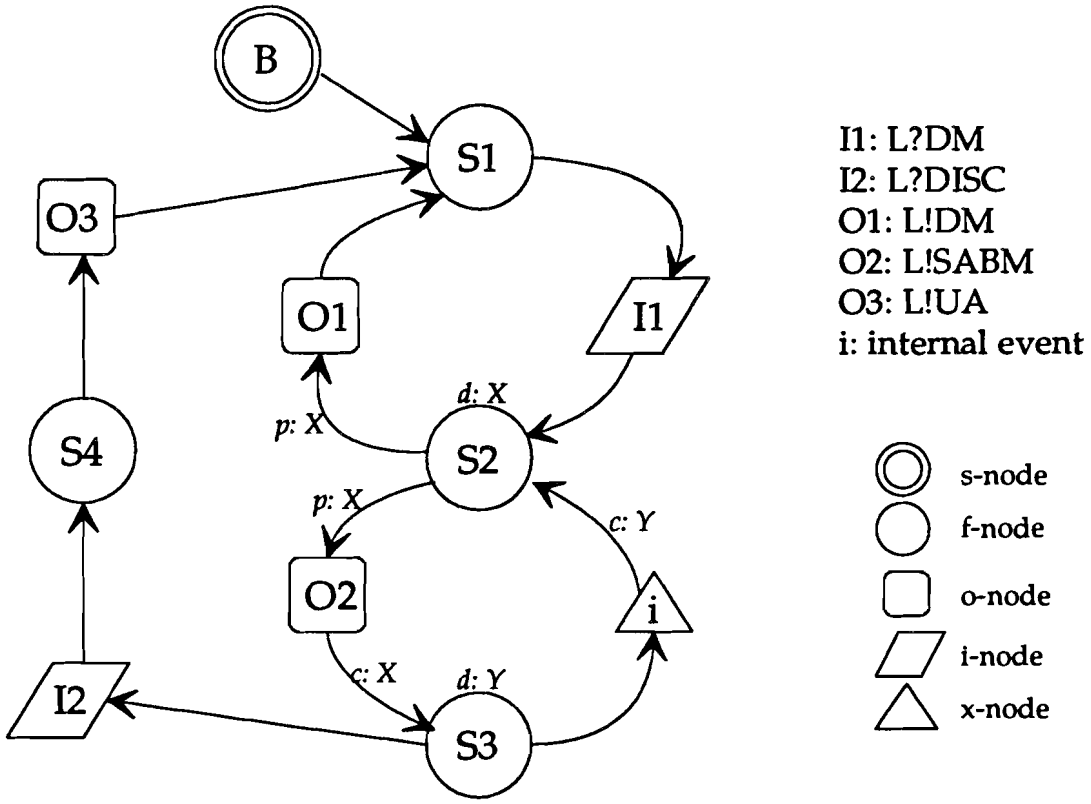

Figure 5.1 : EFG and definitions/uses example LOTOS EFSM

§ 5.7 Derivation of test sequences for control flow

For the derivation we make use of a variant of the UIO-method, the UIOv method. The basic principle of the method is the one explained in § 3.2. Test sequences are constructed for each input event a from state s_i to state s_j by the following steps: First put the state machine into state s_i; details of this step are given in § 5.9. Then input a is supplied and possible output events are checked; this procedure is explained more elaborately in § 5.10. Finally, the tail state after receiving the output events is checked to verify that it is as expected. In order to do this, we use (a variant of) the UIO-method.

The UIO-method was first introduced in [SaDa88]. UIO-sequences have to be computed for every state to which the evaluation step can lead. An UIO-sequence for a state is an input/output behaviour that is not exhibited by any other state. For example, if we consider the state machine in figure 5.2 (edges are expected to have an input and an output, a slightly different situation compared to the LOTOS EFSM), the following UIO-sequences can be found:

S1: *a/1*

S2: *a/0* *a/1*

S3: *a/0* *a/0*

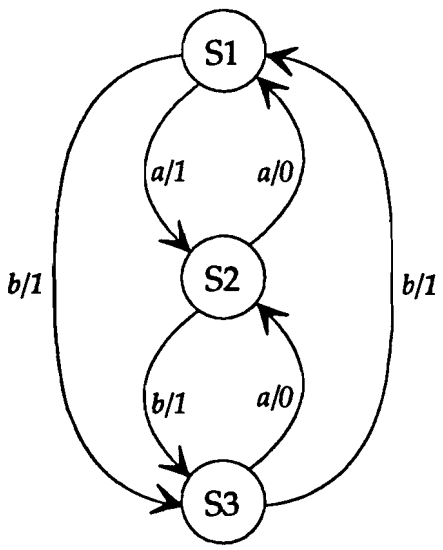

Figure 5.2: Example FSM

The algorithm to compute all UIO-sequences for a state machine is as follows (details in [SaDa88]): Check the uniqueness of the input/output sequences of increasing length from the possible tail states until a UIO-sequence is located. For each possible tail state, all input/output sequences of length 1 are computed and checked for uniqueness. If there is no unique sequence of length 1, then the same procedure is repeated for all sequences of length 2. This procedure is continued for longer sequences until a unique input/output sequence is found or the length of the sequences exceeds a certain upper bound.

A possible improvement of the UIO-method is the *UIOv*-method [ChVuIt89]. This method is based on the observation that although each UIO-sequence is unique in the FSM_s of the specification, the uniqueness of the UIO-sequences may not hold in a faulty implemented

FSM_i. That is, for some FSM_i, the input/output behaviour related to an UIO-sequence may be exhibited by more than one state. This method requires the following procedures to be performed after computing the UIO-sequences:

1) For each state s_i of specification S, apply the UIO-sequence for state s_i after FSM_i is brought into state s_i. This procedure verifies that each state in FSM_s exists in FSM_i.
2) For each state s_i of specification S, apply the UIO-sequence for state s_i after FSM_i is brought to each state s_j, where the input part of the UIO-sequence for state s_j is different from the one of the UIO-sequence for state s_i. This procedure verifies that the UIO-sequence for each state is unique in the given FSM_i.

As already mentioned, the LOTOS EFSM is different from the above assumed (E)FSM. First of all the LOTOS EFSM does not have edges containing an input and an output leading from one flow state to another flow state. The fact that for inputs and outputs separate states are defined instead of assigning these events to an edge is only a matter of definition. The fact that between such an input event and an output event another flow state is defined from which several transitions may take place is more difficult to handle. For flow states which are necessary only for connecting an input and an output event it is not necessary to search UIO-sequences. Only for flow states from which an input event can take place UIO-sequences have to be computed. We will call such states **basic flow states**. Another difference is the presence of internal events. There are two types of internal events in the LOTOS EFSM. The first one appears after an input or an output and is harmless because it only increases the time to pass an input/output action. The second one is more difficult to handle. It provides the possibility in a certain flow state not to choose the desired action from that state but to jump to another, not desired action. The path initiated by such an internal event must be marked as *inconclusive*. More details are given in § 5.10.

§ 5.8 Derivation of test sequences for data flow

Several criterions can be used to derive test sequences for the data flow part of a protocol. A survey of several criterions can be found in [RaWe85]. One of these criterions, the **all-uses** criterion is used in [Ur87]. In this section the criterion from [UrYa91] will be explained, the **IO-df-chain** criterion. An adapted form of the method in [UrYa91] will be used to derive test sequences for data flow.

Before the criterion is stated that should be satisfied by test sequences selected by the method, some terms are defined that will be used in the formulation of the criterion.

> **Definition 5.1:** *An input is the start of a def of a variable which*
> *1) occurs in an input interaction*
> *2) is defined by a call to a next state whose parameter list contains only constants*

For example, in figure 5.3 there are no definitions of variables in an input interaction and there is one call with only constants, the definition of *X* with 0. This input is called **in1** and is defined on the transition that leads to the def of the variable *X*.

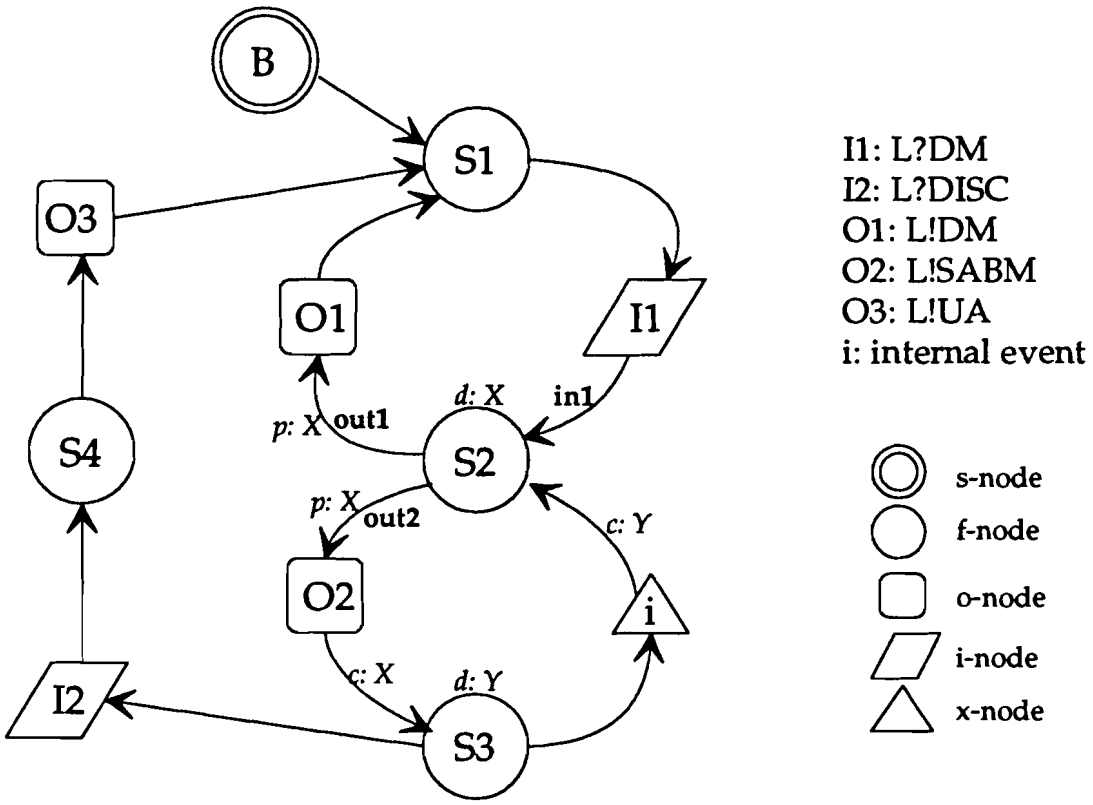

Figure 5.3 : EFG, definitions/uses and inputs/outputs

> **Definition 5.2:** *An output is either a* variable output *(i.e., a c-use of a variable in an output interaction) or a* constant output *(i.e., an output interaction whose parameter list contains only constants).*

For example, no *variable outputs*, however three *constant outputs* **out1**, **out2**, and **out3** exist in figure 5.3.

> **Definition 5.3:** *An use of a variable x, which may be a c-use or a p-use, is **affected** by a def of a variable y if either*
> *1) x and y are the same variable and the use of x is reached by the def of y through a def-clear path[1] with respect to y or*
> *2) a def of variable z is given in terms of a use of y at a node which is reached by the def of y through a def-clear path with respect to y, and the use of x is affected by the def of z.*

For example, all p-uses and c-uses of *X* in figure 5.3 are affected by the def of *X* and by the def of *Y* and the c-use of *Y* is affected by the defs of *X* and *Y*. An advantage of the LOTOS EFSM is that all variables are defined only once and are only valid within one process.

> **Definition 5.4:** *An input I influences an output O if*
> *1) O is a variable output which is affected by I, or*
> *2) O is a constant output and the last p-use which leads the control flow to O is affected by I. Searching back to such a p-use is stopped whenever a basic flow state is reached.*

For example, in figure 5.3 input **in1** influences output **out1** and **out2** but not **out3** because there is no p-use after basic flow state S3.

> **Definition 5.5:** *An **input-output dataflow chain (IO-df-chain)** C is a tuple (I,S,O) where input I influences output O through S which is an ordered sequence of alternating uses and defs containing at most two occurrences of any $(u_x,d_y)_n$.*

---

[1] A path $(n_1,n_2,...,n_{m-1},n_m)$ is a *def-clear path* with respect to a variable y from node n_1 to edge (n_{m-1},n_m) if there are no defs of y at node n_1 following the def of y and there are no defs of y at nodes n_2 to n_{m-1} inclusively.

A path $(n_1,n_2,...,n_{m-1},n_m)$ is a *def-clear path* with respect to a variable y from node n_1 to node n_m if there are no defs of y: a) at node n_1 following the def of y, b) at nodes n_2 to n_{m-1} inclusively, and c) at node n_m preceding the c-use of y.

For example, the tuple $(d_X, (u_X,d_Y)_{S3}, (u_Y,d_X)_{S2}, u_X)$ with the last u_X representing one of the p-uses of *X* is an IO-df-chain in figure 5.3. In order to be able to connect the test sequences we will start and end a path covering such a chain in basic flow states. The path covering the above mentioned IO-df-chain is [S1,I1,S2,O2,S3,i,S2,O1,S1] or [S1,I1,S2,O2,S3,i,S2,O2,S3]. All possible paths covering IO-df-chains in figure 5.3 are:

[S1,I1,S2,O1,S1]
[S1,I1,S2,O2,S3,i,S2,O1,S1]
[S1,I1,S2,O2,S3,i,S2,O2,S3,i,S2,O1,S1]
[S1,I1,S2,O2,S3]
[S1,I1,S2,O2,S3,i,S2,O2,S3]
[S1,I1,S2,O2,S3,i,S2,O2,S3,i,S2,O2,S3]

> **Definition 5.6:** *For a given EFG G, the **IO-df-chain criterion** requires that test sequences be selected to cover each IO-df-chain of G at least once.*

This criterion will be used to derive the test sequences for the data flow part of the protocol.

§ 5.9 Linking the test sequences

After having derived test subsequences for control and data flow with the methods explained in the above sections, these test subsequences have to be linked to one test sequence that can be used for testing the IUT. Several methods can be used to do this. The most simple one is the **reset method**, used for example by M. ten Hacken in [Ha92]. This method returns after a test subsequence to the initial state S1 and departs from this initial state to the first state of a next test subsequence to test this next subsequence. When applying this method to the LOTOS EFSM this method needs sequences to be generated from state S1 to all other basic flow states and to state S1 from all other basic flow states. Only basic flow states have to be considered because these states are possible start and end states of test subsequences. It is probably better to generate all possible linksequences instead of computing such a linksequence for each test subsequence to avoid computing several times the same resetsequence for test subsequences that start or end in the same state.

Clearly, this method is the most simple but not the most efficient method. A better method is the **direct link method** which provides computing linksequences between all basic flow states. After generating all these linksequences a suitable one has to be chosen. For example, if a subsequence ends in state S3 and a next starts in state S6 the linksequence from S3 to S6 has to be used to link these subsequences. For computing such a link path between two states a shortest-path algorithm can be used, for example the **Dijkstra's Algorithm** [AhHoUl76].

Other possible improvements are the **Rural Chinese Postman Tour** or linking using an **overlap method** such as the ones referred to in § 3.2. These are not used because of their complexity. Also research can be done in the area of combining test subsequences and thus reducing the total test sequence in the case of equivalence of several test subsequences.

When deriving TTCN link sequences out of the EFG some difficulties arise. For example, when computing a shortest path it is prefered not to have such a path containing many unstable flow states, i.e. flow states from which several outputactions or internal actions may depart. Such a state will make the TTCN sequence more complex and less effective because of the several *inconclusive paths*. The transformation into TTCN will be explained in the next section.

§ 5.10 Transformation of test sequences into TTCN

The fundamental idea on which this transformation can be based can be found in [GuLo90], the article on which M. ten Hacken based his derivation method. However, some adaptations were necessary because of the application to the SMILE output LOTOS EFSM and because the control flow, data flow and link subsequences are already constructed before the transformation to TTCN. The notation form of TTCN to be used will be **TTCN-GR**, the human-readable tabular form of TTCN. In the future, also the machine-processable form, **TTCN-MP**, can be used by adapting the transformation procedure or using an TTCN-GR to TTCN-MP compiler.

As explained in § 3.5, the method of [GuLo90] consists of 4 steps, the **initialization step**, the **evaluation step**, the **verification step** and the **termination step**. In our method, the initialization and termination step are replaced by the link subsequences, the verification step is replaced by the derivation of UIO-sequences and the test subsequences for the data

flow are added to the steps. So the transformation consists of two basic parts; for the **evaluation step** the same approach as in [GuLo90] will be used with some minor adaptations for applying to the SMILE output LOTOS EFSM; for the transformation of control flow sequences, data flow sequences and link sequences a procedure has to be developed. This procedure is the same for all these sequences because they all consist of a certain path through the EFG and they all start and end in a basic flow state.

Testing the control flow consists of applying the **evaluation step** for every inputaction of the EFG and adding the UIO-sequence to the tail basic flow state of this evaluation step. The next procedure will be used for this evaluation step: For a reception of a frame f_s in the EFG, there is a transmission of frame f_s in the evaluation step. Successively, for a transmission of a frame f_r in the EFG, there is a reception of the frame f_r in the evaluation step. The transmission of f_s and the reception of f_r, if any, represent the body of the test case.

If a frame f_r is received after sending f_s, the verification path of the basic flow state following the reception of f_r will be added to the body of the test case after reception of f_r. This *adding* means that after the evaluation step the UIO-sequence for the tail state of the evaluation state has to be executed. This reception of f_r can be preceded or followed by a number of internal events. These internal events are of the harmless sort, so they may only increase the time necesarry to receive f_r. In the case of several possible receptions two approaches can be followed. Either one of them is declared the right one and the others are declared *inconclusive* paths, or each possible reception is treated as a separate sidepath. The first approach is the most simple but will cause many problems when applying the test sequences to an IUT; the second approach is more complete but results in possibly complex test subsequences, certainly when linking these subsequences. However, this approach is prefered. If no reception follows the sending of f_s and the first basic flow state is a *stable* flow state, i.e. no outputs or internal events lead from this state, the verification path for the stable basic flow state is added to the test case after the timer starts and stops (to make sure that nothing is received). If the basic flow state is unstable, the verification path is added immediately to the body of the test case. An *otherwise* is added and a verdict *fail* is issued at any point where something different from what is expected to be received can occur. All this is summarized in the following textbox.

As already mentioned, the same procedure can be followed for UIO-sequences, data flow sequences and link sequences because they all start and end in a basic flow state and

port!*f*,	
possible **I's**	
-if reception before basic state	
port?*f*,	
possible **I's**	
+verification path basic flow state	***pass***
port?other_frames	
possible **I's**	
+verification paths basic flow states	***pass***
?otherwise	***fail***
?expiration of timer	***fail***
-if no reception and basic flow state is unstable	
+verification path basic flow state(s)	***pass***
-if no reception and basic flow state is stable	
start timer	
?time_out	
+verification path basic flow state	***pass***
?otherwise	***fail***

Transformation procedure evaluation step

because they consist of an already defined path. The procedure to be followed here is as follows: Transform all actions and internal events to subsequent steps in TTCN (reception in EFG becomes transmission in TTCN and transmission in EFG becomes reception in TTCN). If at a certain moment in the path a unstable flow state is reached, all paths containing outputactions, also via internal events must be marked as *inconclusive*. If the internal events lead to a stable flow state this path must be marked as *inconclusive* also. An *otherwise* is added and a verdict *fail* is issued at any point where something different from what is expected to be received can occur. All this is summarized in the second textbox of this section.

If no UIO-sequence is found for a certain basic flow state, a warning must be provided in order to let the testperson take the necessary actions.

actions

-at any unstable flow state:
-for all paths leading to a reception (output in EFG)
possible **I's**

port?f_r	***inconclusive***
?otherwise	***fail***
?expiration of timer	***fail***

-for all paths leading to a stable flow state via I's
possible **I's**, at least one

?expiration of timer	***inconclusive***
actions	***pass***

-for each intended reception

port?f_r	
?otherwise	***fail***
?expiration of timer	***fail***

Transformation procedure UIO, data flow and link sequences

§ 5.11 Checking executability and determining test values

The last thing that has to be done before the real testing can take place is checking the test sequences for executability and the determination of values for the variables that have to be provided to the IUT. Testing the executability is necessary because the conditions for a transition are not taken in consideration. For example, if we take the computed data flow test sequences in § 5.8, [S1,I1,S2,O1,S1] and [S1,I1,S2,O2,S3,i,S2,O1,S1] are not executable because the condition `(X = 2)` for making the transition from S2 to O1 is not met. Also [S1,I1,S2,O2,S3,i,S2,O2,S3,i,S2,O2,S3] is not executable because *X* has the value 2 the last time the condition `(X <> 2)` for making the transition from S2 to 02 has to be met. Therefore these 3 test subsequences can be removed from the set of test subsequences.

Determining input test values that have to be provided to the IUT is closely connected with testing executability. In some cases the test values are already computed. For example, step 2 of the automatic test generation approach (explained in § 5.5) can be seen as determining the appropriate values for the frame variables. For other variables, values may have to be computed. For example, imagine the value of *X* is not initialized 0 on the transition from I1 to S2, but that it is asked as a input in I1. In that case, values for *X*

higher than 1 will cause the transition from S2 to 02 never to be executed and thus will effect in inreachability for the states O2, S3, i, I2, S4 and O3.

For checking executability and determining appropriate test values SMILE can be used (see figure 2.2). Checking executability can be done by **unfolding** each action of the test subsequence successively. During this traversing of a sequence, test values can be determined using the **Instantiate** or **Adt Interface** buttons. For example, after having traversed a certain path, the option **Solve Goal** after calling **Instantiate** will provide all possible values for the variables of the traversed path.

CHAPTER 6: Implementation

§ 6.1 Introduction

In order to derive test cases automatically a tool called **ProTeGe** (Protocol Test Generator) is developed by me. This tool can construct an EFG out of an (adapted) LOTOS EFSM as described in section 5.6, it can derive TTCN test cases for control flow with the algorithm explained in section 5.10 (evaluation step), it can derive UIO sequences and transform them to TTCN as described in section 5.7 and section 5.10 (transformation procedure UIO) and it can derive data flow test sequences as explained in section 5.8 and have them transformed to TTCN with a similar algorithm as for UIO sequences.

The tool is written in the language C and can be compiled to work under the Sun4 workstations, which are also used for the tool Lite and Smile, necessary to derive input files for ProTeGe. The *makefile* used to compile the various modules and to construct ProTeGe is given in appendix A. In this makefile can be seen that ProTeGe is partitioned in two headerfiles and six source modules. In short these modules perform the following task:

typedef.h	This headerfile contains all type definitions used in all modules and all macro definitions.
efghead.h	This headerfile is used for all extern function declarations.
main.c	This module gives the main menu, calls the other modules and contains the error function.
makeefg.c	This module makes the EFG out of an input SMILE EFSM.
makeuio.c	This module makes UIO sequences for all basic flow states of the EFG.
gendflow.c	This module makes the data flow subsequences for the EFG.
linksubs.c	This module is almost empty, but should make link subsequences in the future.
ttcn.c	This module constructs TTCN control flow test cases (evaluation step) and it transforms UIO sequences and data flow subsequences to TTCN test cases.

The source code of these files is given in appendix L. In section 6.3 the headerfiles and main.c are considered, in section 6.4 makeefg.c, in section 6.5 makeuio.c, in section 6.6

gendflow.c and in section 6.7 ttcn.c. In the following section (6.2) the output files structure is explained, in section 6.8 a test session is being done and in section 6.9 conclusions and recommendations are given.

§ 6.2 Output files

ProTeGe uses various files to store the EFG, the calculated subsequences and to write the output test cases to. These files are defined to be constructed in a subdirectory of the directory containing ProTeGe called **efgfiles**, so this directory has to exist. Files are used instead of memory to store intermediate results such as the EFG because problems with insufficient memory in case of huge SMILE EFSMs can be avoided in this way. Also the results can be interpreted more easily if necessary. Disadvantages of using files are the increased complexity of the tool and the increased computing time.

The following files are constructed by ProTeGe (between parentheses the macro definitions for these files, used in the modules) :

state1.tmp	(STATE_FILE_1)	: file constructed to make file state2.efg
trans1.efg	(TRANS_FILE_1)	: contains all transitions of the EFG
var1.tmp	(VAR_FILE_1)	: contains defined vars for each flowstate
state2.efg	(STATE_FILE_2)	: contains all states of the EFG
var2.tmp	(VAR_FILE_2)	: contains all defined variables in the EFSM
intrans.tmp	(IN_FILE)	: contains incoming transitions for flowstates
outtrans.tmp	(OUT_FILE)	: contains outgoing transitions for flowstates
basseq.uio	(BAS_SEQ_FILE)	: contains basic sequences to construct UIO sequences for basic flowstates
basact.uio	(BAS_ACT_FILE)	: contains actions of these basic sequences
basuniq.uio	(BAS_UNIQ_FILE)	: giving existence of variables in basic actions
totseq.uio	(TOT_SEQ_FILE)	: contains all calculated sequences to construct UIO sequences
totact.uio	(TOT_ACT_FILE)	: contains all actions for these sequences
totuniq.uio	(TOT_UNIQ_FILE)	: giving uniqueness (or not) for actions
uio.seq	(UIO_FILE)	: contains UIO sequences for all basic states
dflow.seq	(DFLOW_FILE)	: contains all data flow subsequences
link.seq	(LINK_FILE)	: contains link subsequences
uio.ttcn	(UIO_TTCN_FILE)	: contains TTCN UIO sequences for basic states

cflow.ttcn	(CFLOW_TTCN_FILE)	: contains all TTCN control flow subsequences
dflow.ttcn	(DFLOW_TTCN_FILE)	: contains all TTCN data flow subsequences
link.ttcn	(LINK_TTCN_FILE)	: contains TTCN link subsequences

The most important files are explained in one of the following sections describing the module in which it is created. In appendix D until J examples of STATE_FILE_2, TRANS_FILE_1, UIO_FILE, DFLOW_FILE, CFLOW_TTCN_FILE, UIO_TTCN_FILE and DFLOW_TTCN_FILE are given.

§ 6.3 MAIN.C and headerfiles

The headerfile **typedef.h** contains as already told all type definitions and macro definitions for all files. The major type definitions are:
a structure called *files_type* used to transfer all pointers to all used files between functions.
a structure called *efg_type* used to transfer information about the EFG between functions.
a structure called *ttcn_el_type* used to construct a TTCN test case. It contains elements like the action at a certain time in a test case, a pointer to elements containing next possibilities at this time, a pointer to an element in a next time period, the indentation level and some comment about the action.

The header file **efghead.h** contains as already told external function declarations for the modules.

The module **main.c** contains the function *main* which is the overall structure for calling the other submodules. The user is given a menu giving the various possibilities to do, e.g. deriving an EFG, deriving UIO sequences, data flow sequences, link subsequences (not implemented) and transformation of the subsequences to TTCN test cases. Also this module contains the error function which is called in all modules when something is going wrong.

§ 6.4 MAKEEFG.C

This module creates an Extended Flow Graph (EFG) out of a description of a protocol as an EFSM as created by Smile, the symbolic simulator of the LITE-toolset. If you choose this module from the menu in main, the name of the Smile EFSM will be asked, after

which this EFSM will be scanned and transformed to some files of the suitable form for usage by the other modules. These files are:

- VAR_FILE_1:
This file contains for all flowstates (processes) in the EFSM the defined variables in the header. The structure of this file is as follows: For every flowstate a line '`STATE <num>`', where `<num>` the number of the state is as indicated in the process. After such a line a new line is created for each defined variable and consisting only of the name of this variable.

- VAR_FILE_2:
This file contains all defined variables in the EFSM, each given on a separate line. This file is not used in this module.

- STATE_FILE_1:
This file is actually a temporary file to construct STATE_FILE_2. The structure is the same as the structure of STATE_FILE_2, except for the incoming and outgoing transitions of flowstates which are given in the files IN_FILE en OUT_FILE.

- IN_FILE:
This file contains the incoming transitions of the flowstates. During scanning each time such a call to a flowstate is found, a line '`<statenum> <trans>`' is written to this file; `<statenum>` is the number of the flowstate, `<trans>` is the internal number of the transition (see TRANS_FILE_1).

- OUT_FILE:
This file contains the outgoing transitions of the flowstates. The structure is the same as the structure of IN_FILE.

- The files STATE_FILE_2 en TRANS_FILE_1 are the files that describe the EFG. In STATE_FILE_2 all flowstates, input actions, output actions, choice actions and internal events are described as states as explained in section 5.6. These states are connected by transitions which are given in TRANS_FILE_1. The structures of these files are as follows:

- STATE_FILE_2:

For each state the following lines are given:

'@ST <int_num>' : <int_num> is the statenumber used to discern the various flow-states and actions. This number is also used in TRANS_FILE_1.

'<type> <num>' : <type> gives the type of the state. This can be s (startstate), F (flowstate), I (inputaction), o (outputaction), x (internal event) and c (choice-action). <num> gives the number for a certain type of state. For example, for flowstates this is the number as indicated in the process.

'@ACT <action>' : <action> contains in case of an inputaction or outputaction this action. In other cases the line consists only of '@ACT'.

'@VAR <var_type>' : In case of a flowstate is looked in VAR_FILE_1 if there are variables defined in the header of this flowstate. If so, <var_type> is 'd' and this line is followed by lines each containing such a variable. In case of an inputaction <var_type> is 'd' and the line is followed by a line containing the input variable. In case of an outputaction is looked if there are any defined variables used in this outputaction. If so, <var_type> becomes 'c' and the line is followed by lines containing these output variables. In case of a choicestate <var_type> is 'd' and the line is followed by a line containing the choice variable. In other cases the line consists only of '@VAR'.

'@IN' : This line is followed by a certain number of lines, each containing a transition number as defined in TRANS_FILE_1, which leads to this state (incoming transitions).

'@OUT' : This line is followed by a certain number of lines, each containing a transition number as defined in TRANS_FILE_1, which leads from this state (outgoing transitions).

- TRANS_FILE_1:

For each transition the following lines are given:

'@TR <tr_num>' : <tr_num> is the transition number used to discern all transitions used in this file. This number is also used as incoming and outgoing transitions in STATE_FILE_2.

'@FROM <st_num>' : This line indicates in which state the transition starts. <st_num> is a statenumber, as given in the '@ST <int_num>' line in STATE_FILE_2.

'@TO <st_num>' : This line indicates in which state the transition ends. <st_num> is a statenumber, as given in the '@ST <int_num>' line in STATE_FILE_2.

'@VAR <var_type>' : In case of a transition containing predicates ('[..] ->' in the Smile EFSM), <var_type> is 'p' and the line is followed by a number of lines, each giving a variable used in the predicates. In case of a call to a flowstate which defines variables in his header (parameters), <var_type> is 'c'. This line is then followed by a line for each parameter. In case that one or more variables are used to form a certain parameter, these variables are given in this line separated by tabs. In case a constant value is given to the parameter, the line consists of a tab followed by '@CONST'. If the transition contains no predicates and there is no call to a flowstate containing parameters, the line consists only of '@VAR'.

§ 6.5 MAKEUIO.C

This module calculates for each basic flow state if possible one or more UIO sequences and writes these sequences to the UIO_FILE. In order to derive this file other intermediate files are created and used. The calculation of the UIO sequences consists of the following steps:

First calculate all basic sequences from all basic flow states (state from which input action takes place) to next basic flow states. These sequences are checked for not including variables in the IO sequence because this will complicate making statements about uniqueness of a sequence containing such variables. The basic sequences not containing variables will be checked for *uniqueness* (no other sequences having the same input/output sequence), *half-uniqueness* (other equal IO sequences exist but they lead to a different endstate) and *non-uniqueness* (other equal IO sequences to same endstate). If for a basic flowstate an UIO sequence is found this state is marked and in a next step no new sequences will be calculated. If for a certain flowstate no UIO sequence is found, all half unique sequences of the last step starting with this flowstate will be expanded with all possible basic sequences starting with the same basic flowstate as with which the half unique sequence ends. This algorithm continues until for each basic flowstate an UIO sequence is found or a certain limit is reached.

The files used in this module are:

- VAR_FILE_2:
This file, constructed in the module makeefg, is used for checking if variables exist in a basic sequence.

- BAS_SEQ_FILE:
This file contains all calculated basic sequences, each on a separate line. Such a sequence consists of all states traversed, for example 'F1 I3 F4 O2 X6 F5\n'.

- BAS_ACT_FILE:
This file consists of all input/output sequences matching to the state sequences of BAS_SEQ_FILE, each on the same line as the matching state sequence. Example: 'L?SABM L!DM \n'.

- BAS_UNIQ_FILE:
This file contains for each matching input/output sequence of BAS_ACT_FILE a '0' if this sequence contains no variables, and a '3' if it contains variables.

- TOT_SEQ_FILE:
This file contains all calculated state sequences in the different steps in a same structure as BAS_SEQ_FILE.

- TOT_ACT_FILE: This file contains all input/output sequences matching to the state sequences of TOT_SEQ_FILE.

- TOT_UNIQ_FILE:
This file contains for each sequence in the TOT-files a '0' if the sequence is non-unique, a '1' if the sequence is unique and a '2' if the sequence is half-unique.

- UIO_FILE: This file is the main file of this module. It contains all calculated UIO sequences for all basic flow states. For each basic flow state, it has the following structure:

'_____'

'F<x>' <x> is number of flowstate

For each UIO sequence of this flow state:

a blank line

state sequence as in TOT_SEQ_FILE 'NO UIO' if no UIO sequence found

IO sequence as in TOT_ACT_FILE 'NO UIO' if no UIO sequence found

§ 6.6 GENDFLOW.C

This module calculates all data flow subsequences for the EFG and writes them to DFLOW_FILE. The method used to do this is explained in section 5.8. A data flow chain is a state sequence from an 'input' to an 'output'. An 'input' (see definition 5.1) can be a real input of a variable but can also be a call to a flowstate with only constants. An 'output' (see definition 5.2) can be a variable in a variable output or a variable in a predicate that leads to a constant output before a basic flowstate is encountered. In the calculated data flow chains the 'output' must be influenced by the 'input', e.g. the value of the 'output' depends in some way on the 'input'. The written data flow sequences are always extended at the beginning and end to the nearest basic flow state. Stop criteria for calculating data flow chains are, besides the situations when there are no new definitions of variables in a new flowstate influenced by the input variable, the existence of three subsequent flowstate loops in the state sequence and reaching a maximal length in states of the data flow chain without extension to basic flow states. This maximal length is asked before calculating the data flow chains. Also is asked if data flow chains have to be calculated for variable inputs, constant calls or for both.

The constructed file in this module, DFLOW_FILE, has the following structure:

- A line '---------- Real Inputs ----------\n'
- For each data flow chain where the 'input' is a real variable input:
    - a line 'var_out <input_var> <output_var> <sequence>\n' where <input_var> is the variable in the input, <output_var> is the variable in the variable output en <sequence> is the total data flow chain with extensions to the nearest basic flow state.
    - a line 'const_out <input_var> <output_var> <sequence>\n' where <input_var> is the variable in the input, <output_var> is the variable used in a predicate leading to a constant output en <sequence> is the total data flow chain with extensions to the nearest basic flow state.

- A line '`---------- Constant Calls ----------\n`'
- For each data flow chain where the 'input' is a constant call:
    - a line '`var_out <input_var> <output_var> <sequence>\n`' where `<input_var>` is the variable in the input, `<output_var>` is the variable in the variable output en `<sequence>` is the total data flow chain with extensions to the nearest basic flow state.
    - a line '`const_out <input_var> <output_var> <sequence>\n`' where `<input_var>` is the variable in the input, `<output_var>` is the variable used in a predicate leading to a constant output en `<sequence>` is the total data flow chain with extensions to the nearest basic flow state.

§ 6.7 TTCN.C

This module constructs TTCN test cases for the control flow and it transforms the UIO sequences and the data flow sequences formed in makeuio.c and gendflow.c to TTCN. The transformation of link subsequences is not yet implemented. The TTCN test cases for control flow are written, with a header giving the purpose of the test case, its identifier and some comment lines giving information about the test case, to CFLOW_TTCN_FILE. For example, if we take the test case for control flow of the following text box the header

```
------------------------------------------------------------------------------
                        Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?SABM' from flowstate F1
Identifier: CFL_1
------------------------------------------------------------------------------
 Line |             Behaviour Description          |  Verdict
      |                                            |
    1 |    L !SABM                                 |
    2 |        L ?UA                               |
    3 |            +UIO sequence state F3          |  PASS
    4 |        L ?DM                               |
    5 |            START TIMER                     |
    6 |                ?TIME OUT                   |
    7 |                    +UIO sequence state F1  |  PASS
    8 |                ?OTHERWISE                  |  FAIL
    9 |        ?OTHERWISE                          |  FAIL
   10 |        ?EXPIRATION OF TIMER                |  FAIL
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F1, check input 'L ?SABM'
Line   5 : Checking if state F1 is stable
```

Example control flow TTCN test case

of a test case contains information about the **purpose** of the test case and a **identifier** for the test case. The identifier starts with *CFL* in case of a control flow test case, *UIO* in

case of an UIO sequence and *DFL* in case of a test case for data flow. In the body of the test case the test sequence is given in the graphical form of TTCN and preceded by a line number. This line number is used in the comment part to explain some actions or to give some additional information that can be useful to the user. If in the body a line starts with a '+' this means that here some other TTCN test case or test case part must be inserted.

The algorithm to construct TTCN control flow test cases is the one explained in section 5.10 (evaluation step). The UIO sequences and data flow subsequences are read from DFLOW_FILE and UIO_FILE (only one UIO sequence for each basic state) and transformed to TTCN and written, also preceded by a header and followed by comment lines, to DFLOW_TTCN_FILE and UIO_TTCN_FILE. The used algorithm is the second one in section 5.10 (for UIO, data flow and link sequences).

§ 6.8 Example session

In this section an example session will be elucidated in order to show how ProTeGe works, what choices can be made and which messages are given by the tool. As an example input EFSM we take a simple version of a Connection Disconnection LAPB protocol. This LOTOS specification is given in appendix B. After we transformed this specification in an EFSM using SMILE, we have a LOTOS EFSM called **cd+.efsm.** Because all the input frames only can have one value, we can transform this EFSM into a simplified version using the transformation procedure of section 5.5. After we have done this, we have the adapted EFSM called **cd++.efsm** that is given in appendix C. We will use this EFSM in our example session. The state machine representing this EFSM is given in figure 6.1 on the next page.

We start our session with typing **protege** in the directory where we can find this tool. The output of the session on the screen and the inputs given (bold) are given in appendix K. We see that the tool starts with giving a main menu, asking whether you want to create an EFG out of an EFSM (1), make UIO sequences for the (basic) flow states of this EFG (2), generate data flow subsequences for this EFG (3), generate link subsequences (4) which is not yet implemented, transform subsequences to TTCN test cases (5) or if you want to quit ProTeGe (0).

We always have to start with creating an EFG out of the EFSM, so option **1.** After choosing this option, the name of the input LOTOS EFSM (the SMILE output) is asked.

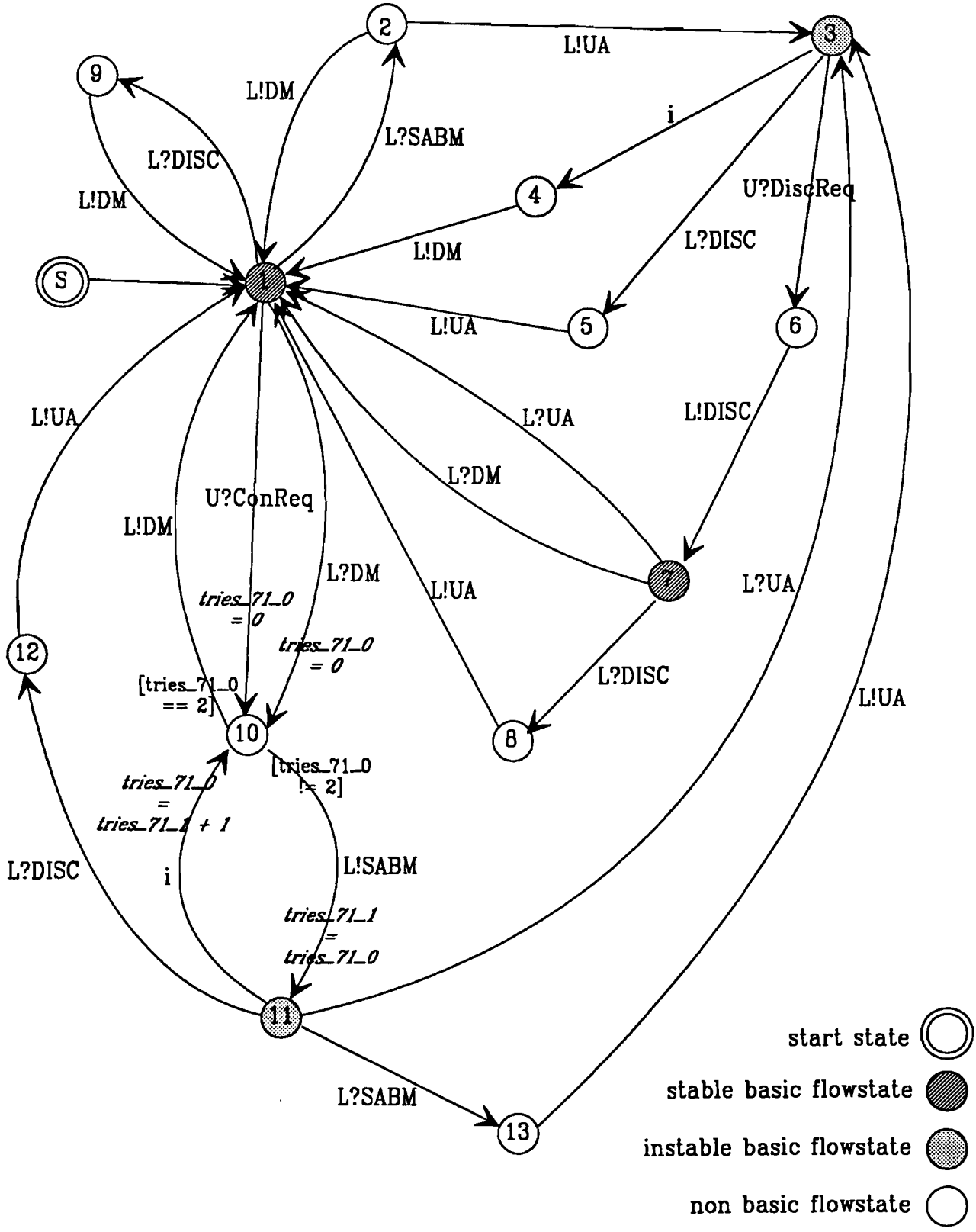

Figure 6.1: State machine representation cd++.efsm

We type here the name of our example EFSM with a path relative to the directory in which the tool is called. So, if this is the same directory as where ProTeGe can be found, we type **cd++.efsm**. Next ProTeGe starts to scan this LOTOS EFSM, starting with scanning all defined variables in all flowstates (processes in the LOTOS EFSM) and writing these variables to *var1.tmp*. Then it scans the name and the ports of the EFSM, the *SPECIFICATION* part of the EFSM, the *BEHAVIOUR* part and all processes (flowstates). While doing this, the user is being informed on the progress by message lines. The stars in the message lines for a process indicate the number of lines scanned in this process. The next thing being done is finishing the EFG by adding incoming and outgoing transitions of flowstates. At the end the user is informed about some statistics of the EFG, e.g. how many transitions and states are constructed to form the EFG. After this the main menu is shown again.

The next option we can choose is making UIO sequences, generating data flow sequences, generating link sequences or even a suboption in transformation to TTCN, making control flow TTCN test cases. If we choose making UIO sequences (**2**), ProTeGe first calculates basic sequences for constructing UIO sequences (BAS_SEQ_FILE), e.g. all sequences from a basic flow state to a basic flowstate. Next it constructs the actions of these basic sequences (BAS_ACT_FILE), and it checks which basic actions contain variables (BAS_UNIQ_FILE) which will not be used for constructing UIO sequences. Then the maximal number of steps to construct UIO sequences will be asked, e.g. how many steps from a flowstate to another flowstate. The default is the square of the number of basic flow states. Calculating will continue until this maximal number of steps is reached or until an UIO sequence is found for each basic flowstate. After this, the main menu is shown again.

If we choose to generate data flow subsequences (**3**), the maximal length of such a sequence in states (also input, output, internal events and choice states) is asked. This maximal length doesn't include the extensions from the start and end of such a sequence to the nearest basic flowstate. Then it is asked if you want to have sequences derived for 'inputs' being formed by a constant call or by a real input of a variable (or both). During calculating these sequences the user is being informed about the progress, e.g. the present length of the sequences to check if they are all right.

If we choose generating link sequences (**4**), a message is given that this option is not yet implemented.

If we have constructed all these sequences we can transform them to TTCN test cases and also derive TTCN test cases for the control flow. It is asked whether you want to make TTCN control flow test cases, TTCN data flow test cases or TTCN link test cases. If you choose control flow test cases, also the UIO sequences are transformed to TTCN test case parts. At the end we choose **0** to quit the ProTeGe.

§ 6.9 Conclusions and recommendations

If we look at the results we can say a tool has been developed that can help by testing implementations of protocols specified in LOTOS. However, some action is needed from the user to rewrite the input LOTOS EFSM and to test the executability of the calculated test cases. For example, if we take the example protocol and we want to apply the UIO sequence for flowstate 11, we have to provide that *tries_71_1* when starting in flowstate 11 equals 0, because otherwise the condition for going from flowstate 10 to IUT output L!SABM can never be true. In relation also with these conditional jumps a number of data flow test sequences cannot be executed: *DFL_1*, *DFL_3*, *DFL_6*, *DFL_7*, *DFL_9* and *DFL_12*. But the tool provides the basic tools for constructing a number of tests for a protocol in both the field of control flow and data flow.

Unfortunately the tool is not tested exhaustively. Besides for the inputfile *cd*, used in the example session and given in the appendices ProTeGe has been used to derive test cases for a more complex version of this protocol called *cdo*. This protocol has also been adapted for this purpose with the methods of section 5.5. This protocol consists of 36 flowstates. No errors appeared when using ProTeGe and the time necessary to get all TTCN test cases was in the order of a few minutes. Another protocol called *lapb* containing about the same number of flowstates as *cdo* but much more transitions and more complex conditional transitions had no problems when deriving the EFG, but because this protocol was not rewritten with the methods of section 5.5 there were too many real input variables to derive reasonable test cases in a reasonable time.

There are several possibilities to improve the tool. First of all, ProTeGe doesn't have a graphical user interface which would be more user friendly in showing the various possibilities and the results. Also, for showing the intermediate and final outputfiles and offering possibilities to change these files in relation to non-executable sequences (these sequences can be removed) or a restricted number of test cases some tools could be developed. These tools should be able to provide reading and changing the files for

example between deriving test sequences and transforming them to TTCN. To be able to check executability some link to SMILE or some simple tools to traverse the EFG could be provided. Another improvement that can be realized quite easily is the possibility to quit ProTeGe between the different steps and returning to it without losing information. This would require the information in the structure *efg* to be stored in some file instead of in a structure. A very obvious amelioration is providing a tool that can derive link sequences between basic flowstates. This can be easily done by a breadth-first search in the EFG from a basic flowstate to another (or to one reset basic state and back). For transforming these sequences to TTCN the same procedure as for transforming UIO sequences and data flow sequences can be used with only some minor adaptations for the header and for the comment. A next improvement would be the amelioration of the procedure to derive UIO sequences. At this moment the first UIO sequence that is found is chosen. This can result in bad UIO sequences like the one for flowstate 7 only containing one inputaction of the UIO. An improvement would be to show all found UIO sequences for a state and asking the user which one has to be chosen or whether searching new ones should continue. Also improving the method by choosing UIO sequences with the *UIOv*-method is recommendable. This would require a tool performing the two procedures in section 5.7. A tool combining several test cases can decrease the number of test cases to be executed. For example in case of loops in the case of data flow chains there is no need to consider each loop as another test case. Finally, a tool choosing the best order of executing the various test cases in order to have minimal link sequences and giving the best possibilities to go to the beginning of the next test case when entering an *inconclusive*-path or *fail*-path can improve ProTeGe.

The most complex improvement is to adapt the tool for deriving test cases for protocols specified in other languages like SDL or Estelle. There has been tried to give a general model to the EFG in order to have only compilers to be written which construct a similar EFG out the specification or specification EFSM, but research is necessary to find the differences between the languages and the impact of those differences on the adaptation of ProTeGe.

CHAPTER 7: Conclusions

The conclusions can be divided into conclusions about the methods to be used for deriving automatically test cases for protocols, conclusions about the tool ProTeGe developed to do this derivation and conclusions about the applicability of this tool and the methods at the protocol test environment of TRT.

For both control flow and data flow aspects of protocols several test methods exist. The methods for testing control flow are more examined and applied than the rather recently developed methods to test data flow aspects of protocols. The best control flow method to use for developing a tool to derive test is the UIO-method because of its generality and FSM-based approach. The data flow test method based on data flow chains was also chosen because of its generality and possibility to improve the method in the future.

The developed tool called ProTeGe is able to derive TTCN test cases for both control flow and data flow aspects of a protocol specified in LOTOS and converted to an EFSM with a tool called SMILE. However, it must be tested more thoroughly with bigger protocols to detect possible errors. The tool itself can be improved in many ways of which the most important are generating link sequences, improving the UIO sequence derivation procedure and making it more userfriendly by providing a graphical user interface and providing reading and changing output files. Also research has to be done about the differences between LOTOS and other specification languages and the impact of those differences on deriving TTCN test cases and using ProTeGe. Also, the method(s) on which ProTeGe is based needs some effort of the user like adapting the input EFSM in order to restrict the number of test cases and making it easier for ProTeGe to derive UIO sequences, and like checking the output test sequences for executability and finding test values. So the developed tool is not fully automatic but tries to find acceptable test cases with an acceptable level of user activity.

When applying the automatic test derivation methods and in especially ProTeGe to the test environment of TRT in order to improve this environment (the aim of my project) some difficulties arise. First of all, the test environment at TRT is not based on the conformance testing methods used to derive automatically test sequences. Possible solutions for this problem are changing the observation points of the test equipment or testing in an earlier stage of development of the protocol. A second problem is the absence of formal specifications in general and, if formally specified, the use of SDL instead of LOTOS. As

already mentioned, research has to be done to how to apply the methods and ProTeGe to SDL. A third problem is that TTCN is not used as the test language. However, a tool exists to transform TTCN to the language used at the test environment of TRT.

In general it can be concluded that the methods and developed tool cannot be used directly to improve the test environment of TRT, but a survey of the most important test methods is given and a tool is developed to help deriving test sequences more easily. However, more research can and must be done.

Glossary

Basic flowstate
Flowstate from which at least one inputaction takes place. These basic flowstates are used in ProTeGe as start and endpoints of test subsequences.
=> ProTeGe, test subsequence, stable flowstate

CFG
Control Flow Graph. Graph representation of a protocol only showing the control flow part of the protocol, so the basic structure of the protocol.
=> Control Flow, DFG, EFG

Conformance Testing
Testing method for communication protocols checking if an implementation of a protocol acts as expected with regard to its specification.
=> Interoperability Testing

Control Flow
Part of a protocol representing the basic structure of the protocol; part of protocol that can be represented by an FSM without conditional transitions, variable uses etc.
=> Data Flow, FSM, CFG

Data Flow Chain
Test sequence constructed by applying IO df-chain method, testing an aspect of the data flow. Chain starts at a definition of a variable and ends at some variable that is used in a output or in a conditional transition to an output. This end variable must be influenced by the defined variable.
=> IO df-chain method, Data Flow

Data Flow
Part of the protocol enabling to lead the protocol in some direction using conditional transitions and use of variables.
=> Control Flow, DFG

DFG

Data Flow Graph. Graph representation of a protocol showing the data flow part of a protocol.

=> Data Flow, CFG, EFG

EFG

Extended Flow Graph. Graph representation of a protocol showing both control and data flow part of the protocol.

=> Control Flow, CFG, Data Flow, DFG

EFSM

Extended Finite State Machine. Finite state machine in which the states are parametrized with some variables, allowing conditional jumps from one state to another.

=> FSM

Estelle

Formal Description Technique based on an EFSM representation.

=> FDT, EFSM, LOTOS, SDL

FDT

Formal Description Technique. Language used to specify protocols in a formal way. Most important FDTs: LOTOS, SDL, Estelle.

=> LOTOS, SDL, Estelle

FSM

Finite State Machine. State machine representation of for example a protocol.

=> EFSM

Half Unique IO sequence

Sequence of inputs and outputs in a protocol having at least one similar input/output (IO) sequence in the state machine that traverses different states. No one of these similar IO sequences leads however to the same endstate.

=> Unique IO sequence, Non Unique IO sequence

Interoperability Testing
Testing method for communication protocols checking if a protocol implementation can interoperate with another protocol implementation.
=> Conformance Testing

IO df-chain method
Input/Output Data Flow chain method. Method testing data flow part of a protocol by making test sequences for all sequences starting with a defintion of a variable and ending with a output use of a variable that is influenced by this definition.
=> Data Flow chain

IUT
Implementation Under Test. Implementation of a protocol that is to be tested.

LITE
LOTOS Integrated Tool Environment. LOTOS toolset supporting the development of complex real distributed and concurrent systems in the whole trajectory from specification to implementation.
=> SMILE

LOTOS
Language Of Temporal Ordering Specification. FDT for the formal specification of open distributed systems, which are seen as processes, possibly consisting of a hierarchy of several sub-processes.
=> FDT, SDL, Estelle

MUIO method
Multiple Unique Input/Output method. Variant of UIO method using multiple UIO sequences.
=> UIO method

NFT
Normal Form Transition. Simplified form of Estelle transitions.
=> Estelle

Non Unique IO sequence
Sequence of inputs and outputs in a protocol having at least one similar input/output (IO) sequence in the state machine that traverses different states, leading to a same endstate.
=> Unique IO sequence, Half Unique IO sequence

Overlap method
Method deriving test sequences for states using partial overlap of these sequences in order to reduce the total sequence.

ProTeGe
Protocol Test Generator. Developed tool to derive TTCN test cases for control and data flow of a protocol represented as a LOTOS EFSM.
=> TTCN, Control Flow, Data Flow, LOTOS, EFSM

PUIO method
Partial Unique Input/Output method. Variant of the UIO method using partial test sequences.
=> UIO

RCPT
Rural Chinese Postman Tour. A minimum cost roundtrip in a graph involving a selected set of edges.

SDL
Specification and Description Language. FDT based on a EFSM model, designed as a means of support to the specification and description process of telecommunication systems.
=> FDT, LOTOS, Estelle

SMILE
Tool of LITE, acting as a symbolic simulator for protocols specified in LOTOS. Can also derive LOTOS EFSMs which are used as inputfile for ProTeGe.
=> LITE, LOTOS, EFSM, ProTeGe

Stable flowstate
Flowstate from which only inputactions take place.
=> Basic flowstate

Test sequence
Concatenated test subsequences. However, in this report often test sequences is used when test subsequences was intended when wrong understanding was not probable.
=> test subsequence

Test subsequence
Sequence of input symbols for a protocol causing certain outputs by the implementation. Such a sequence is used for testing some aspect of a protocol.
=> test sequence

Transition Tour
Method to check a protocol implementation by traversing all nodes of the graph representation of the protocol.

TTCN
Tree and Tabular Combined Notation. Language used to describe test cases of protocols.

UIO method
Unique Input/Output method. Method for checking a state in a state machine by searching for unique input/output sequences starting at this state to be tested.
=> Unique IO sequence

UIOv method
Variant of the UIO method based on the observation that the uniqueness of UIO sequences may not hold in a faulty implementation.
=> UIO method

Unique IO sequence
Unique Input/Output sequence. Sequence of inputs and outputs in a protocol having no similar input/output (IO) sequence in the state machine that traverses different states.
=> Non Unique IO sequence, Half Unique IO sequence

Bibliography

[AhDaLeUy88] **Aho A.V., Dahbura A.T., Lee D., Uyar M.Ü.**
An optimization technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours
Protocol Specification, Testing and Verification, VIII, Page 75-86
Elseviers Science Publishers B.V. (North-Holland), 1988
also in:
IEEE Trans. on Communications, Vol 39, No 11, p 1604-1615, November 1991

[AhHoUl76] **Aho A.V., Hopcroft J.E., Ullman J.D.**
The Design and Analysis of Computer Algorithms
Addison-Wesley, Reading, MA, p 207-209, 1976

[Al90] **Alderen R.**
COOPER the compositional construction of a canonical tester
Formal Description Techniques II
S.T. Vuong (Editor) Elsevier Science Publishers (North-Holland), 1990

[BoBr87] **Bolognesi T., Brinksma E.**
Introduction to the ISO Specification Language LOTOS
Computer Networks and ISDN Systems Vol 14, No 1, p 25-59, 1987
Elsevier Science Publishers B.V. (North-Holland), 1987

[BoUy91] **Bosik B.S., Uyar M.Ü.**
Finite state machine based formal methods in protocol performance testing: from theory to implementation
Computer Netw. and ISDN systems, Vol 22, No 1, p 7-33, 1991

[Br88] **Brinksma E.**
A theory for the derivation of tests
Protocol Specification, Testing and Verification VIII
Elseviers Science Publishers B.V. (North-Holland), IFIP 1988

[BrAlLa90] **Brinksma E., Alderen R., Langerak R., Lagemaat J. v.d, Termans J.**
A Formal Approach to Conformance Testing
Formal Description Techniques II
S.T. Vuong (Editor), Elsevier Science Publishers (North-Holland), 1990

[BrScSt87] **Brinksma E., Scollo G., Steenbergen C.**
LOTOS specifications, their implementations and their tests
Protocol Specification, Testing and Verification, VI. Page 349-360
Elseviers Science Publishers B.V. (North-Holland), IFIP 1987

[BuKrKw90] **Burgt S.P. v.d., Kroon J., Kwast E., Wilts H.J.**
The RNL Conformance Kit
Proc. 2nd Int. Workshop on Protocol Test Systems
W. Effelsberg, L. Mackert, J. de Meer: editors
North-Holland 1990

[Ch78] **Chow T.S.**
Testing software design modeled by FSM's
IEEE Trans. on Software Eng., VOL SE-4, No 3, p 178-187, March 1978

[ChAm92] **Chun W., Amer P.D.**
Improvements on UIO sequence generation and partial UIO sequences
12th International Symposium on Protocol Specification, Testing and Verification
Lake Buena Vista, Florida, USA, June 22-25 1992

[ChChKe90] **Chen M.-S., Choi Y., Kershenbaum A.**
Approaches utilizing segment overlap to minimize test sequences
L. Logrippo, R. L. Probert, H. Ural (editors)
Protocol Specification, Testing and Verification X, p 85-98
Elseviers Science Publishers B.V. (North-Holland) 1990

[ChVuIt89] **Chan W.Y.L., Vuong S.T., Ito M.R.**
An improved protocol test generation procedure based on UIOS
Computer Communications Review, Vol 19, No 4, September 1989

[ChVuIt90] **Chan W.Y.L., Vuong S.T., Ito M.R.**
On test sequence generation for protocols
Protocol Specification, Testing and Verification, IX, Page 75-86
Elseviers Science Publishers B.V. (North-Holland), IFIP 1990

[Do91] **Doornbosch P.**
Test derivation for full LOTOS
ISSN 0923-1714, Memoranda Informatica 91-51 TIOS 91/020
April 1991

[DrChBl92] **Drayton L., Chetwynd A., Blair G.**
Introduction to LOTOS through a worked example
Computer Communications, Vol 15 No 2 March 1992

[Eer92] **Eertink H.**
Executing LOTOS specifications: the SMILE tool
Workshop proceedings Vol.I, Third Lotosphere Workshop & Seminar 1992

[Go70] **Gonenc G.**
A method for the design of fault detection experiments
IEEE Trans. Comput., Vol C-19, p 551-558, June 1970

[GuLo90] **Gueraichi D., Logrippo L.**
Derivation of Test Cases for LAPB from a LOTOS Specification
Formal Description Techniques II
S.T. Vuong (editor), Elseviers Science Publishers B.V. (North-Holland), 1990

[Gu] **Guillet Ph.**
Manuel de methodologie dossier du logiciel, Procedure generale
FBW 105 225, TRT internal report

[Ha92] **Hacken M.A.M. ten**
Using FDT LOTOS to derive tests
graduation report
Eindhoven University of Technology, 1992

[ISO8807] **ISO IS/8807**
LOTOS - a Formal Description Technique based on the temporal ordering of observational behaviour
1988

[ISO9646-1,2] **ISO IS/9646-1,2**
OSI Conformance Testing Methodology and Framework, Part 1 & 2: General Principles and Absract Test Suite Specification

[ISO9646-3] **ISO DIS/9646-3**
OSI Conformance Testing Methodology and Framework, Part 3: The Tree and Tabular Combined Notation (TTCN), March 1990
TTCN Extensions (interim working document), ISO/IEC JTC 1/SC 21 N5077

[Kw91] **Kwast E.**
Towards automatic test generation for protocol data aspects
B. Jonsson, J. Parrow, B. Pehrson (editors)
Protocol Specification, Testing and Verification XI, p 333-348, (North-Holland) 1991

[Lau92] **Lau Y.M.**
Evaluation of Lite toolset
Philips Natlab internal report, 1992, Eindhoven

[LoFaHa92] **Logrippo L., Faci M., Haj-Hussein M.**
An introduction to LOTOS: learning by examples
Computer networks and ISDN systems 23 (1992), page 325-342

[Lot92] **The Lotosphere Consortium**
Parts of the Lotosphere Lite user manual
1992

[Ma92] **Mañas J.A.**
Getting to use Lite
Workshop proceedings Vol.I, Third Lotosphere Workshop & Seminar, August 21, 1992

[MiPa91] **Miller R.E., Paul S.**
Generating minimal length test sequence for conformance testing of communication protocols
IEEE INFOCOM '91

[MiPa92] **Miller R.E., Paul S.**
Generating conformance test sequences for combined control and data flow of communication protocols
12th International Symposium on Protocol Specification, Testing and Verification
Lake Buena Vista, Florida, USA, June 22-25 1992

[NaSa92] **Naik K., Sarikaya B.**
Testing Communicating Protocols
IEEE Software, January 1992, p 27-37

[RaWe85] **Rapps S., Weyuker E.J.**
Selecting software test data using data flow information
IEEE Trans. on Software Engineering, Vol 11, No 4, p 367-375, April 1985

[SaBo82] **Sarikaya B., Bochmann G.v.**
Some experience with test sequence generation
(C. Sunshine editor) Protocol Specifications, Testing and Verification II
North-Holland 1982, p 555-567

[SaBoCe87] **Sarikaya B., Bochmann G. v., Cerny E.**
A test design methodology for protocol testing
IEEE Trans. on Software Engineering, Vol 13, No 5, p 518-539, May 1987

[SaDa88] **Sabnani K.K., Dahbura A.T.**
A protocol test generation procedure
Computer Networks and ISDN Syst. 15, p 285-297, 1988

[ShLoDa92] **Shen Y.-N., Lombardi F., Dahbura A.T.**
Protocol Conformance Testing Using Multiple UIO Sequences
IEEE Tr. on Communications, Vol 40, No 8, p 1282-1287, August 1992

[SiLe89] **Sidhu D.P., Leung T.-K.**
Formal methods for protocol testing: a detailed study
IEEE Trans. on Software Engineering, Vol 15, No 4, p 413-426, April 1989

[SuShLoSc91] **Sun X., Shen Y.-N., Lombardi F., Sciuto D.**
Protocol conformance testing by discriminating UIO sequences
B. Jonsson, J. Parrow, B. Pehrson (editors)
Protocol Specification, Testing and Verification XI, p 349-364, (North-Holland) 1991

[Tre90] **Tretmans J.**
Test Case Derivation from LOTOS Specifications
Formal Description Techniques II
S.T. Vuong (editor), Elseviers Science Publishers B.V. (North-Holland), 1990

[TriSa91] **Tripathy P., Sarikaya B.**
Test Generation from LOTOS Specifications
IEEE Transactions on Computers, Vol 40, No 4 April 1991

[Ur87] **Ural H.**
Test sequence selection based on static data flow analysis
Computer Communications, Vol 10, No 5, p 234-242, October 1987

[Ur92] **Ural H.**
Formal methods for test sequence generation
Computer Communications, Vol 15, No 5, p 311-325, June 1992

[UrYa91] **Ural H., Yang B.**
A test sequence selection method for protocol testing
IEEE Trans. on Communications, Vol 39, No 4, April 1991

[VeSchZo92] **Velthuys R.J., Schneider J.M., Zorntlein G.**
A test derivation method based on exploiting structure information
12th International Symposium on Protocol Specification, Testing and Verification
Lake Buena Vista, Florida, USA, June 22-25 1992

[ViPiLa92] **Vissers C.A., Pires L.F., Lagemaat J. v.d.**
Lotosphere, an attempt towards a design culture
Workshop proceedings Vol.I, Third Lotosphere Workshop & Seminar, July 27, 1992

[WaHu87] **Wang B., Hutchinson D.**
Protocol testing techniques
Computer Communications, Vol 10, No 2, p 79-87, April 1987

[WaLi92] **Wang C.J., Liu M.T.**
A test suite generation method for Extended Finite State Machines using axiomatic semantics approach
12th International Symposium on Protocol Specification, Testing and Verification
Lake Buena Vista, Florida, USA, June 22-25 1992

[We92] **Wezeman C.D.**
Deriving tests from LOTOS specifications
Workshop proceedings Vol.I, Third Lotosphere Workshop & Seminar 1992

[WeBaLy91] **Wezeman C.D., Batley S., Lynch J.A.**
Formal Methods to Assist Conformance Testing
Formal Description Techniques III
J. Quemada, J. Mañas, E. Vázquez (editors)
Elseviers Science Publishers B.V. (North-Holland), 1991

```
CFLAGS=       -g

OBJ=          main.o \
              makeefg.o \
              makeuio.o \
              gendflow.o \
              linksubs.o \
              ttcn.o

INC=          typedef.h efghead.h

protege :     $(OBJ)
       cc -o protege $(OBJ)

main.o :      $(INC) main.c

makeefg.o :   $(INC) makeefg.c

makeuio.o :   $(INC) makeuio.c

gendflow.o :  $(INC) gendflow.c

linksubs.o :  $(INC) linksubs.c

ttcn.o :      $(INC) ttcn.c

all:
       touch *.c
       make
```

```
specification connection_disconnection[L,U]:noexit
library NaturalNumber, DecNatRepr, DecDigit, Boolean endlib

type L_Frame_type is
sorts  L_Frame_sort
opns   SABM,UA,DISC,DM : -> L_Frame_sort
endtype

type U_Frame_type is
sorts  U_Frame_sort
opns   ConReq, DiscReq: -> U_Frame_sort
endtype

type control_l_type is Boolean, L_Frame_type
opns   _eq_,_ne_          : L_Frame_sort,L_Frame_sort      -> Bool
eqns
       forall x,y:L_Frame_sort
       ofsort Bool
              x = y =>
              x eq y = true;
              x ne y = not(x eq y);
              SABM eq UA = false;
              SABM eq DISC = false;
              SABM eq DM = false;
              UA eq SABM = false;
              UA eq DISC = false;
              UA eq DM = false;
              DISC eq SABM = false;
              DISC eq UA = false;
              DISC eq DM = false;
              DM eq SABM = false;
              DM eq UA = false;
              DM eq DISC = false
endtype (* control_l_type *)

type control_u_type is Boolean, U_Frame_type
opns   _eq_,_ne_          : U_Frame_sort,U_Frame_sort      -> Bool
eqns
       forall x,y:U_Frame_sort
       ofsort Bool
              x = y =>
              x eq y = true;
              x ne y = not(x eq y);
              ConReq eq DiscReq = false;
              DiscReq eq ConReq = false
endtype (* control_u_type *)

behaviour connection[L,U]

where

  process connection[L,U]:noexit:=
      L?lf:L_Frame_sort[lf eq DM]; connect_res[L,U](Natnum(0))
         []
      U?uf:U_Frame_sort[uf eq ConReq]; connect_res[L,U](Natnum(0))
         []
```

```
      L?lf:L_Frame_sort[lf eq DISC]; L!DM; connection[L,U]
         []
      L?lf:L_Frame_sort[lf eq SABM]; (L!DM; connection[L,U]
                     []
               L!UA; disconnection[L,U])
endproc
  process connect_res[L,U](tries:nat):noexit:=
     [tries ne NatNum(2)]  ->  (L!SABM;
        (L?lf:L_Frame_sort[lf eq SABM]; L!UA; disconnection[L,U]
              []
         L?lf:L_Frame_sort[lf eq UA]; disconnection[L,U]
              []
         L?lf:L_Frame_sort[lf eq DISC]; L!UA; connection[L,U]
              []
         i; connect_res[L,U] (succ(tries))))
         []
     [tries eq natnum(2)] -> L!DM; connection[L,U]
  endproc
  process disconnection[L,U]:noexit:=
    U?uf:U_Frame_sort[uf eq DiscReq]; L!DISC;
      (L?lf:L_Frame_sort[lf eq DM]; connection[L,U]
                  []
       L?lf:L_Frame_sort[lf eq UA]; connection[L,U]
                  []
       L?lf:L_Frame_sort[lf eq DISC];L!UA; connection[L,U])
          []
    L?lf:L_Frame_sort[lf eq DISC]; L!UA; connection[L,U]
         []
    i; L!DM; connection[L,U]
  endproc
endspec
```

```
SPECIFICATION connection_disconnection [L, U]: noexit(* EFSM generated by SMILE *)


BEHAVIOUR
State1 [L, U]
WHERE

    PROCESS State1 [L, U]: noexit  :=
    L ?SABM;
         State2 [L, U]
    [] L ?DISC;
         State9 [L, U]
    [] U ?ConReq;
         State10 [L, U] (Natnum(0))
    [] L ?DM;
         State10 [L, U] (Natnum(0))
    ENDPROC

    PROCESS State2 [L, U]: noexit  :=
    L !UA;
         State3 [L, U]
    [] L !DM;
         State1 [L, U]
    ENDPROC

    PROCESS State3 [L, U]: noexit  :=
     (*i*) i ;
         State4 [L, U]
    [] L ?DISC;
         State5 [L, U]
    [] U ?DiscReq;
         State6 [L, U]
    ENDPROC

    PROCESS State4 [L, U]: noexit  :=
    L !DM;
         State1 [L, U]
    ENDPROC

    PROCESS State5 [L, U]: noexit  :=
    L !UA;
         State1 [L, U]
    ENDPROC

    PROCESS State6 [L, U]: noexit  :=
    L !DISC;
         State7 [L, U]
    ENDPROC

    PROCESS State7 [L, U]: noexit  :=
    L ?DISC;
         State8 [L, U]
    [] L ?UA;
         State1 [L, U]
    [] L ?DM;
         State1 [L, U]
```

```
    ENDPROC

    PROCESS State8 [L, U]: noexit  :=
    L !UA;
         State1 [L, U]
    ENDPROC

    PROCESS State9 [L, U]: noexit  :=
    L !DM;
         State1 [L, U]
    ENDPROC

    PROCESS State10 [L, U] (tries_71_0:nat): noexit  :=
    [(tries_71_0 eq Natnum(2))] -> L !DM;
         State1 [L, U]
    [] [(tries_71_0 ne Natnum(2))] -> L !SABM;
         State11 [L, U] (tries_71_0)
    ENDPROC

    PROCESS State11 [L, U] (tries_71_1:nat): noexit  :=
     (*i*) i ;
         State10 [L, U] (succ(tries_71_1))
    [] L ?DISC;
         State12 [L, U]
    [] L ?UA;
         State3 [L, U]
    [] L ?SABM;
         State13 [L, U]
    ENDPROC

    PROCESS State12 [L, U]: noexit  :=
    L !UA;
         State1 [L, U]
    ENDPROC

    PROCESS State13 [L, U]: noexit  :=
    L !UA;
         State3 [L, U]
    ENDPROC

ENDSPEC
```

```
@ST 1
S 1
@ACT
@VAR
@IN
@OUT
1

@ST 2
F 1
@ACT
@VAR
@IN
1
13
21
23
29
31
33
35
37
49
@OUT
2
4
6
8

@ST 3
I 1
@ACT L ?SABM
@VAR
@IN
2
@OUT
3

@ST 4
F 2
@ACT
@VAR
@IN
3
@OUT
10
12

@ST 5
I 2
@ACT L ?DISC
@VAR
@IN
4
@OUT
5

@ST 6
F 9
@ACT
@VAR
@IN
5
@OUT
34

@ST 7
I 3
@ACT U ?ConReq
@VAR
@IN
6
@OUT
7

@ST 8
F 10
@ACT
@VAR d
tries_71_0
@IN
7
9
41
@OUT
36
38

@ST 9
I 4
@ACT L ?DM
@VAR
@IN
8
@OUT
9

@ST 10
O 1
@ACT L !UA
@VAR
@IN
10
@OUT
11

@ST 11
F 3
@ACT
@VAR
@IN
11
45
51
@OUT
14
16
18

@ST 12
O 2
@ACT L !DM
@VAR
@IN
12
@OUT
13

@ST 13
X 1
@ACT i
@VAR
@IN
14
@OUT
15

@ST 14
F 4
@ACT
@VAR
@IN
15
@OUT
20

@ST 15
I 5
@ACT L ?DISC
@VAR
@IN
16
@OUT
17

@ST 16
F 5
@ACT
@VAR
@IN
17
@OUT
22

@ST 17
I 6
@ACT U ?DiscReq
@VAR
@IN
```

```
18
@OUT
19

@ST 18
F 6
@ACT
@VAR
@IN
19
@OUT
24

@ST 19
O 3
@ACT L !DM
@VAR
@IN
20
@OUT
21

@ST 20
O 4
@ACT L !UA
@VAR
@IN
22
@OUT
23

@ST 21
O 5
@ACT L !DISC
@VAR
@IN
24
@OUT
25

@ST 22
F 7
@ACT
@VAR
@IN
25
@OUT
26
28
30

@ST 23
I 7
@ACT L ?DISC
@VAR
@IN
26
@OUT
27

@ST 24
F 8
@ACT
@VAR
@IN
27
@OUT
32

@ST 25
I 8
@ACT L ?UA
@VAR
@IN
28
@OUT
29

@ST 26
I 9
@ACT L ?DM
@VAR
@IN
30
@OUT
31

@ST 27
O 6
@ACT L !UA
@VAR
@IN
32
@OUT
33

@ST 28
O 7
@ACT L !DM
@VAR
@IN
34
@OUT
35

@ST 29
O 8
@ACT L !DM
@VAR
@IN
36
@OUT
37

@ST 30
O 9
@ACT L !SABM
@VAR
@IN
38
@OUT
39

@ST 31
F 11
@ACT
@VAR d
tries_71_1
@IN
39
@OUT
40
42
44
46

@ST 32
X 2
@ACT i
@VAR
@IN
40
@OUT
41

@ST 33
I 10
@ACT L ?DISC
@VAR
@IN
42
@OUT
43

@ST 34
F 12
@ACT
@VAR
@IN
43
@OUT
48

@ST 35
I 11
@ACT L ?UA
@VAR
@IN
```

```
44
@OUT
45

@ST 36
I 12
@ACT L ?SABM
@VAR
@IN
46
@OUT
47

@ST 37
F 13
@ACT
@VAR
@IN
47
@OUT
50

@ST 38
O 10
@ACT L !UA
@VAR
@IN
48
@OUT
49

@ST 39
O 11
@ACT L !UA
@VAR
@IN
50
@OUT
51
```

```
@TR 1
@FROM 1
@TO 2
@VAR

@TR 2
@FROM 2
@TO 3
@VAR

@TR 3
@FROM 3
@TO 4
@VAR

@TR 4
@FROM 2
@TO 5
@VAR

@TR 5
@FROM 5
@TO 6
@VAR

@TR 6
@FROM 2
@TO 7
@VAR

@TR 7
@FROM 7
@TO 8
@VAR c
       @CONST

@TR 8
@FROM 2
@TO 9
@VAR

@TR 9
@FROM 9
@TO 8
@VAR c
       @CONST

@TR 10
@FROM 4
@TO 10
@VAR

@TR 11
@FROM 10
@TO 11
@VAR

@TR 12
@FROM 4
@TO 12
@VAR

@TR 13
@FROM 12
@TO 2
@VAR

@TR 14
@FROM 11
@TO 13
@VAR

@TR 15
@FROM 13
@TO 14
@VAR

@TR 16
@FROM 11
@TO 15
@VAR

@TR 17
@FROM 15
@TO 16
@VAR

@TR 18
@FROM 11
@TO 17
@VAR

@TR 19
@FROM 17
@TO 18
@VAR

@TR 20
@FROM 14
@TO 19
@VAR

@TR 21
@FROM 19
@TO 2
@VAR

@TR 22
@FROM 16
@TO 20
@VAR

@TR 23
@FROM 20
@TO 2
@VAR

@TR 24
@FROM 18
@TO 21
@VAR

@TR 25
@FROM 21
@TO 22
@VAR

@TR 26
@FROM 22
@TO 23
@VAR

@TR 27
@FROM 23
@TO 24
@VAR

@TR 28
@FROM 22
@TO 25
@VAR

@TR 29
@FROM 25
@TO 2
@VAR

@TR 30
@FROM 22
@TO 26
@VAR

@TR 31
@FROM 26
@TO 2
@VAR

@TR 32
@FROM 24
@TO 27
@VAR

@TR 33
@FROM 27
@TO 2
@VAR

@TR 34
@FROM 6
```

```
@TO 28
@VAR

@TR 35
@FROM 28
@TO 2
@VAR

@TR 36
@FROM 8
@TO 29
@VAR p
tries_71_0

@TR 37
@FROM 29
@TO 2
@VAR

@TR 38
@FROM 8
@TO 30
@VAR p
tries_71_0

@TR 39
@FROM 30
@TO 31
@VAR c
       tries_71_0

@TR 40
@FROM 31
@TO 32
@VAR

@TR 41
@FROM 32
@TO 8
@VAR c
       tries_71_1

@TR 42
@FROM 31
@TO 33
@VAR

@TR 43
@FROM 33
@TO 34
@VAR

@TR 44
@FROM 31
@TO 35
@VAR

@TR 45
@FROM 35
@TO 11
@VAR

@TR 46
@FROM 31
@TO 36
@VAR

@TR 47
@FROM 36
@TO 37
@VAR

@TR 48
@FROM 34
@TO 38
@VAR

@TR 49
@FROM 38
@TO 2
@VAR

@TR 50
@FROM 37
@TO 39
@VAR

@TR 51
@FROM 39
@TO 11
@VAR
```

F1

F1 I1 F2 O2 F1
L?SABM L!DM

F1 I2 F9 O7 F1
L?DISC L!DM

F1 I3 F10 O8 F1
U?ConReq L!DM

F1 I3 F10 O9 F11
U?ConReq L!SABM

F1 I4 F10 O8 F1
L?DM L!DM

F1 I4 F10 O9 F11
L?DM L!SABM

F3

F3 I6 F6 O5 F7
U?DiscReq L!DISC

F7

F7 I9 F1
L?DM

F11

F11 X2 F10 O9 F11
L!SABM

```
------------------ Real Inputs ------------------
------------------ Constant Calls ------------------
const_out tries_71_0 tries_71_0 F1 I3 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I3 F10 O9 F11
const_out tries_71_0 tries_71_0 F1 I3 F10 O9 F11 X2 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I3 F10 O9 F11 X2 F10 O9 F11
const_out tries_71_0 tries_71_0 F1 I3 F10 O9 F11 X2 F10 O9 F11 X2 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I3 F10 O9 F11 X2 F10 O9 F11 X2 F10 O9 F11
const_out tries_71_0 tries_71_0 F1 I4 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I4 F10 O9 F11
const_out tries_71_0 tries_71_0 F1 I4 F10 O9 F11 X2 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I4 F10 O9 F11 X2 F10 O9 F11
const_out tries_71_0 tries_71_0 F1 I4 F10 O9 F11 X2 F10 O9 F11 X2 F10 O8 F1
const_out tries_71_0 tries_71_0 F1 I4 F10 O9 F11 X2 F10 O9 F11 X2 F10 O9 F11
```

```
------------------------------------------------------------------------------
                        Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?SABM' from flowstate F1
Identifier: CFL_1
------------------------------------------------------------------------------
 Line |             Behaviour Description          |  Verdict
      |                                            |
    1 |    L !SABM                                 |
    2 |        L ?UA                               |
    3 |            +UIO sequence state F3          |  PASS
    4 |        L ?DM                               |
    5 |            START TIMER                     |
    6 |                ?TIME OUT                   |
    7 |                    +UIO sequence state F1  |  PASS
    8 |                ?OTHERWISE                  |  FAIL
    9 |        ?OTHERWISE                          |  FAIL
   10 |        ?EXPIRATION OF TIMER                |  FAIL
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F1, check input 'L ?SABM'
Line   5 : Checking if state F1 is stable
```

```
------------------------------------------------------------------------------
                        Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DISC' from flowstate F1
Identifier: CFL_2
------------------------------------------------------------------------------
 Line |             Behaviour Description          |  Verdict
      |                                            |
    1 |    L !DISC                                 |
    2 |        L ?DM                               |
    3 |            START TIMER                     |
    4 |                ?TIME OUT                   |
    5 |                    +UIO sequence state F1  |  PASS
    6 |                ?OTHERWISE                  |  FAIL
    7 |        ?OTHERWISE                          |  FAIL
    8 |        ?EXPIRATION OF TIMER                |  FAIL
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F1, check input 'L ?DISC'
Line   3 : Checking if state F1 is stable
```

```
------------------------------------------------------------------------------
                        Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'U ?ConReq' from flowstate F1
Identifier: CFL_3
```

```
-------------------------------------------------------------------------------
 Line |             Behaviour Description                | Verdict
      |                                                  |
    1 |    U !ConReq                                     |
    2 |        L ?DM                                     |
    3 |            START TIMER                           |
    4 |                ?TIME OUT                         |
    5 |                    +UIO sequence state F1        | PASS
    6 |                ?OTHERWISE                        | FAIL
    7 |        L ?SABM                                   |
    8 |            +UIO sequence state F11               | PASS
    9 |        ?OTHERWISE                                | FAIL
   10 |        ?EXPIRATION OF TIMER                      | FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F1, check input 'U ?ConReq'
Line   3 : Checking if state F1 is stable
```

```
-------------------------------------------------------------------------------
                          Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DM' from flowstate F1
Identifier: CFL_4
-------------------------------------------------------------------------------
 Line |             Behaviour Description                | Verdict
      |                                                  |
    1 |    L !DM                                         |
    2 |        L ?DM                                     |
    3 |            START TIMER                           |
    4 |                ?TIME OUT                         |
    5 |                    +UIO sequence state F1        | PASS
    6 |                ?OTHERWISE                        | FAIL
    7 |        L ?SABM                                   |
    8 |            +UIO sequence state F11               | PASS
    9 |        ?OTHERWISE                                | FAIL
   10 |        ?EXPIRATION OF TIMER                      | FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F1, check input 'L ?DM'
Line   3 : Checking if state F1 is stable
```

```
-------------------------------------------------------------------------------
                          Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DISC' from flowstate F3
Identifier: CFL_5
-------------------------------------------------------------------------------
 Line |             Behaviour Description                | Verdict
      |                                                  |
    1 |    L !DISC                                       |
```

```
    2 |        L ?UA                                |
    3 |            START TIMER                      |
    4 |                ?TIME OUT                    |
    5 |                    +UIO sequence state F1   |  PASS
    6 |                ?OTHERWISE                   |  FAIL
    7 |        ?OTHERWISE                           |  FAIL
    8 |        ?EXPIRATION OF TIMER                 |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F3, check input 'L ?DISC'
Line   3 : Checking if state F1 is stable
```

```
                        Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'U ?DiscReq' from flowstate F3
Identifier: CFL_6
-------------------------------------------------------------------------------
 Line |             Behaviour Description           |  Verdict
      |                                             |
    1 |    U !DiscReq                              |
    2 |        L ?DISC                             |
    3 |            START TIMER                     |
    4 |                ?TIME OUT                   |
    5 |                    +UIO sequence state F7  |  PASS
    6 |                ?OTHERWISE                  |  FAIL
    7 |        ?OTHERWISE                          |  FAIL
    8 |        ?EXPIRATION OF TIMER                |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F3, check input 'U ?DiscReq'
Line   3 : Checking if state F7 is stable
```

```
                        Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DISC' from flowstate F7
Identifier: CFL_7
-------------------------------------------------------------------------------
 Line |             Behaviour Description           |  Verdict
      |                                             |
    1 |    L !DISC                                 |
    2 |        L ?UA                               |
    3 |            START TIMER                     |
    4 |                ?TIME OUT                   |
    5 |                    +UIO sequence state F1  |  PASS
    6 |                ?OTHERWISE                  |  FAIL
    7 |        ?OTHERWISE                          |  FAIL
    8 |        ?EXPIRATION OF TIMER                |  FAIL
-------------------------------------------------------------------------------
```

```
COMMENTS:
Line   1 : Start in F7, check input 'L ?DISC'
Line   3 : Checking if state F1 is stable
```

```
------------------------------------------------------------------------------
                          Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?UA' from flowstate F7
Identifier: CFL_8
------------------------------------------------------------------------------
 Line |       Behaviour Description     |  Verdict
      |                                 |
    1 |    L !UA                        |
    2 |        +UIO sequence state F1   |  PASS
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F7, check input 'L ?UA'
```

```
------------------------------------------------------------------------------
                          Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DM' from flowstate F7
Identifier: CFL_9
------------------------------------------------------------------------------
 Line |       Behaviour Description     |  Verdict
      |                                 |
    1 |    L !DM                        |
    2 |        +UIO sequence state F1   |  PASS
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F7, check input 'L ?DM'
```

```
------------------------------------------------------------------------------
                          Test Case Control Flow
------------------------------------------------------------------------------
Purpose:    Testing input 'L ?DISC' from flowstate F11
Identifier: CFL_10
------------------------------------------------------------------------------
 Line |             Behaviour Description          |  Verdict
      |                                            |
    1 |    L !DISC                                 |
    2 |        L ?UA                               |
    3 |            START TIMER                     |
    4 |                ?TIME OUT                   |
    5 |                    +UIO sequence state F1  |  PASS
    6 |                ?OTHERWISE                  |  FAIL
    7 |        ?OTHERWISE                          |  FAIL
```

```
    8 |        ?EXPIRATION OF TIMER                | FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F11, check input 'L ?DISC'
Line   3 : Checking if state F1 is stable
```

```
-------------------------------------------------------------------------------
                          Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'L ?UA' from flowstate F11
Identifier: CFL_11
-------------------------------------------------------------------------------
 Line |       Behaviour Description    | Verdict
      |                                |
    1 |    L !UA                       |
    2 |        +UIO sequence state F3  | PASS
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F11, check input 'L ?UA'
```

```
-------------------------------------------------------------------------------
                          Test Case Control Flow
-------------------------------------------------------------------------------
Purpose:    Testing input 'L ?SABM' from flowstate F11
Identifier: CFL_12
-------------------------------------------------------------------------------
 Line |         Behaviour Description      | Verdict
      |                                    |
    1 |    L !SABM                        |
    2 |        L ?UA                      |
    3 |            +UIO sequence state F3 | PASS
    4 |        ?OTHERWISE                 | FAIL
    5 |        ?EXPIRATION OF TIMER       | FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Start in F11, check input 'L ?SABM'
```

```
--------------------------------------------------------------------------------
                         Test Case Part UIO Sequence
--------------------------------------------------------------------------------
Purpose:    Testing uniqueness flowstate F1
Identifier: UIO_1
--------------------------------------------------------------------------------
 Line |      Behaviour Description      |  Verdict
      |                                 |
    1 |    L !SABM                      |
    2 |        L ?DM                    |  PASS
    3 |        L ?UA                    |  INCONCLUSIVE
    4 |        ?OTHERWISE               |  FAIL
    5 |        ?EXPIRATION OF TIMER     |  FAIL
--------------------------------------------------------------------------------
COMMENTS:
Line   3 : Possible sidepath to output
```

```
--------------------------------------------------------------------------------
                         Test Case Part UIO Sequence
--------------------------------------------------------------------------------
Purpose:    Testing uniqueness flowstate F3
Identifier: UIO_2
--------------------------------------------------------------------------------
 Line |      Behaviour Description      |  Verdict
      |                                 |
    1 |    U !DiscReq                   |
    2 |        L ?DISC                  |  PASS
    3 |        ?OTHERWISE               |  FAIL
    4 |        ?EXPIRATION OF TIMER     |  FAIL
    5 |    I                            |
    6 |        L ?DM                    |  INCONCLUSIVE
    7 |        ?OTHERWISE               |  FAIL
    8 |        ?EXPIRATION OF TIMER     |  FAIL
--------------------------------------------------------------------------------
COMMENTS:
Line   5 : Internal Event : i
Line   6 : Possible sidepath to output
```

```
--------------------------------------------------------------------------------
                         Test Case Part UIO Sequence
--------------------------------------------------------------------------------
Purpose:    Testing uniqueness flowstate F7
Identifier: UIO_3
--------------------------------------------------------------------------------
 Line |    Behaviour Description    |  Verdict
      |                             |
    1 |    L !DM                    |  PASS
--------------------------------------------------------------------------------
COMMENTS:
```

```
-------------------------------------------------------------------------------
                        Test Case Part UIO Sequence
-------------------------------------------------------------------------------
Purpose:    Testing uniqueness flowstate F11
Identifier: UIO_4
-------------------------------------------------------------------------------
 Line |      Behaviour Description    |  Verdict
      |                               |
    1 |    I                          |
    2 |        L ?SABM                |  PASS
    3 |        L ?DM                  |  INCONCLUSIVE
    4 |        ?OTHERWISE             |  FAIL
    5 |        ?EXPIRATION OF TIMER   |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Internal Event : i
Line   3 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_1
-------------------------------------------------------------------------------
 Line |      Behaviour Description    |  Verdict
      |                               |
    1 |    U !ConReq                  |
    2 |        L ?DM                  |  PASS
    3 |        L ?SABM                |  INCONCLUSIVE
    4 |        ?OTHERWISE             |  FAIL
    5 |        ?EXPIRATION OF TIMER   |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   3 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_2
-------------------------------------------------------------------------------
 Line |      Behaviour Description    |  Verdict
      |                               |
    1 |    U !ConReq                  |
    2 |        L ?SABM                |  PASS
    3 |        L ?DM                  |  INCONCLUSIVE
    4 |        ?OTHERWISE             |  FAIL
    5 |        ?EXPIRATION OF TIMER   |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
```

```
           Start in flowstate F1
Identifier: DFL_3
-------------------------------------------------------------------------------
 Line |          Behaviour Description       |  Verdict
      |                                      |
    1 |    U !ConReq                         |
    2 |        L ?SABM                       |
    3 |            I                         |
    4 |                L ?DM                 |  PASS
    5 |                L ?SABM               |  INCONCLUSIVE
    6 |                ?OTHERWISE            |  FAIL
    7 |                ?EXPIRATION OF TIMER  |  FAIL
    8 |        L ?DM                         |  INCONCLUSIVE
    9 |        ?OTHERWISE                    |  FAIL
   10 |        ?EXPIRATION OF TIMER          |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   5 : Possible sidepath to output
Line   8 : Possible sidepath to output
```

```
                        Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_4
-------------------------------------------------------------------------------
 Line |          Behaviour Description       |  Verdict
      |                                      |
    1 |    U !ConReq                         |
    2 |        L ?SABM                       |
    3 |            I                         |
    4 |                L ?SABM               |  PASS
    5 |                L ?DM                 |  INCONCLUSIVE
    6 |                ?OTHERWISE            |  FAIL
    7 |                ?EXPIRATION OF TIMER  |  FAIL
    8 |        L ?DM                         |  INCONCLUSIVE
    9 |        ?OTHERWISE                    |  FAIL
   10 |        ?EXPIRATION OF TIMER          |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
```

```
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Possible sidepath to output
Line   8 : Possible sidepath to output
```

Test Case Data Flow

```
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_5
```

Line	Behaviour Description	Verdict
1	U !ConReq	
2	L ?SABM	
3	I	
4	L ?SABM	
5	I	
6	L ?DM	PASS
7	L ?SABM	INCONCLUSIVE
8	?OTHERWISE	FAIL
9	?EXPIRATION OF TIMER	FAIL
10	L ?DM	INCONCLUSIVE
11	?OTHERWISE	FAIL
12	?EXPIRATION OF TIMER	FAIL
13	L ?DM	INCONCLUSIVE
14	?OTHERWISE	FAIL
15	?EXPIRATION OF TIMER	FAIL

```
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   7 : Possible sidepath to output
Line  10 : Possible sidepath to output
Line  13 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                        Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_6
-------------------------------------------------------------------------------
 Line |              Behaviour Description           |  Verdict
      |                                              |
    1 |    U !ConReq                                 |
    2 |        L ?SABM                               |
    3 |            I                                 |
    4 |                L ?SABM                       |
    5 |                    I                         |
    6 |                        L ?SABM               |  PASS
    7 |                        L ?DM                 |  INCONCLUSIVE
    8 |                        ?OTHERWISE            |  FAIL
    9 |                        ?EXPIRATION OF TIMER  |  FAIL
   10 |                L ?DM                         |  INCONCLUSIVE
   11 |                ?OTHERWISE                    |  FAIL
   12 |                ?EXPIRATION OF TIMER          |  FAIL
   13 |        L ?DM                                 |  INCONCLUSIVE
   14 |        ?OTHERWISE                            |  FAIL
   15 |        ?EXPIRATION OF TIMER                  |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   6 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   7 : Possible sidepath to output
Line  10 : Possible sidepath to output
Line  13 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                        Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_7
```

```
-------------------------------------------------------------------------------
 Line |      Behaviour Description    |  Verdict
      |                               |
    1 |    L !DM                      |
    2 |        L ?DM                  |  PASS
    3 |        L ?SABM                |  INCONCLUSIVE
    4 |        ?OTHERWISE             |  FAIL
    5 |        ?EXPIRATION OF TIMER   |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   3 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_8
-------------------------------------------------------------------------------
 Line |      Behaviour Description    |  Verdict
      |                               |
    1 |    L !DM                      |
    2 |        L ?SABM                |  PASS
    3 |        L ?DM                  |  INCONCLUSIVE
    4 |        ?OTHERWISE             |  FAIL
    5 |        ?EXPIRATION OF TIMER   |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_9
-------------------------------------------------------------------------------
 Line |          Behaviour Description       |  Verdict
      |                                      |
    1 |    L !DM                             |
    2 |        L ?SABM                       |
    3 |            I                         |
```

```
    4 |                L ?DM                  |  PASS
    5 |                L ?SABM                |  INCONCLUSIVE
    6 |                ?OTHERWISE             |  FAIL
    7 |                ?EXPIRATION OF TIMER   |  FAIL
    8 |         L ?DM                         |  INCONCLUSIVE
    9 |         ?OTHERWISE                    |  FAIL
   10 |         ?EXPIRATION OF TIMER          |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   5 : Possible sidepath to output
Line   8 : Possible sidepath to output
```

```
-------------------------------------------------------------------------------
                          Test Case Data Flow
-------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_10
-------------------------------------------------------------------------------
 Line |          Behaviour Description        |  Verdict
      |                                       |
    1 |    L !DM                              |
    2 |        L ?SABM                        |
    3 |            I                          |
    4 |                L ?SABM                |  PASS
    5 |                L ?DM                  |  INCONCLUSIVE
    6 |                ?OTHERWISE             |  FAIL
    7 |                ?EXPIRATION OF TIMER   |  FAIL
    8 |         L ?DM                         |  INCONCLUSIVE
    9 |         ?OTHERWISE                    |  FAIL
   10 |         ?EXPIRATION OF TIMER          |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Possible sidepath to output
Line   8 : Possible sidepath to output
```

```
------------------------------------------------------------------------------
                        Test Case Data Flow
------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_11
------------------------------------------------------------------------------
 Line |              Behaviour Description           | Verdict
      |                                              |
    1 |    L !DM                                     |
    2 |        L ?SABM                               |
    3 |            I                                 |
    4 |                L ?SABM                       |
    5 |                    I                         |
    6 |                        L ?DM                 | PASS
    7 |                        L ?SABM               | INCONCLUSIVE
    8 |                        ?OTHERWISE            | FAIL
    9 |                        ?EXPIRATION OF TIMER  | FAIL
   10 |                L ?DM                         | INCONCLUSIVE
   11 |                ?OTHERWISE                    | FAIL
   12 |                ?EXPIRATION OF TIMER          | FAIL
   13 |        L ?DM                                 | INCONCLUSIVE
   14 |        ?OTHERWISE                            | FAIL
   15 |        ?EXPIRATION OF TIMER                  | FAIL
------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   7 : Possible sidepath to output
Line  10 : Possible sidepath to output
Line  13 : Possible sidepath to output
```

```
------------------------------------------------------------------------------
                        Test Case Data Flow
------------------------------------------------------------------------------
Purpose:    Testing Data Flow Chain from constant call variable 'tries_71_0'
            to predicate variable 'tries_71_0' (constant output)
            Start in flowstate F1
Identifier: DFL_12
------------------------------------------------------------------------------
```

```
 Line |              Behaviour Description              |  Verdict
      |                                                 |
    1 |    L !DM                                        |
    2 |        L ?SABM                                  |
    3 |            I                                    |
    4 |                L ?SABM                          |
    5 |                    I                            |
    6 |                        L ?SABM                  |  PASS
    7 |                        L ?DM                    |  INCONCLUSIVE
    8 |                        ?OTHERWISE               |  FAIL
    9 |                        ?EXPIRATION OF TIMER     |  FAIL
   10 |                L ?DM                            |  INCONCLUSIVE
   11 |                ?OTHERWISE                       |  FAIL
   12 |                ?EXPIRATION OF TIMER             |  FAIL
   13 |        L ?DM                                    |  INCONCLUSIVE
   14 |        ?OTHERWISE                               |  FAIL
   15 |        ?EXPIRATION OF TIMER                     |  FAIL
-------------------------------------------------------------------------------
COMMENTS:
Line   1 : Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' formed by constant call
Line   2 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   3 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   4 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   5 : Internal Event : i
           Call to flowstate 10 in which the following variables are defined:
           'tries_71_0' influenced by variable(s) 'tries_71_1'
Line   6 : Call to flowstate 11 in which the following variables are defined:
           'tries_71_1' influenced by variable(s) 'tries_71_0'
Line   7 : Possible sidepath to output
Line  10 : Possible sidepath to output
Line  13 : Possible sidepath to output
```

```
AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
5               Transform subsequences to TTCN

0               Quit

Your choice? 1

Give name of Smile output (LOTOS EFSM): cd++.efsm

State variables written to efgfiles/var1.tmp

spec: connection_disconnection
port: L
port: U

SPECIFICATION-header scanned

BEHAVIOUR-part scanned

Scanning process 1: **** Ready
Scanning process 2: ** Ready
Scanning process 3: *** Ready
Scanning process 4: * Ready
Scanning process 5: * Ready
Scanning process 6: * Ready
Scanning process 7: *** Ready
Scanning process 8: * Ready
Scanning process 9: * Ready
Scanning process 10: ** Ready
Scanning process 11: **** Ready
Scanning process 12: * Ready
Scanning process 13: * Ready

Adding incoming and outgoing transitions:  ready

13 processes scanned, End Of File (Smile EFSM) reached
Extended Flow Graph consists of 39 states en 51 transitions
These states can be subdivided into:
1 Start State
13 Control Flow States
12 Input States
11 Output States
0 Choice States
2 Internal States
```

```
AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
```

```
5               Transform subsequences to TTCN

0               Quit

Your choice? 2

Calculating basic sequences for flowstate 11

Calculating actions of basic sequences

Checking variable existence in basic sequences

Give maximal number of steps for UIO-length (default 16): 16

Calculating step 1 of determination of UIO sequences

All uio's found

-------------------------------------------------------------

AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
5               Transform subsequences to TTCN

0               Quit

Your choice? 3

Give maximal length of sequences to look for (max 500 states) : 200

For which 'inputs' do you want to derive Data Flow testsubsequences?

1               Inputs of Variables and Constant Calls
2               Only Inputs of Variables
3               Only Constant Calls

0               No derivation Data Flow testsubsequences

Your choice? 1

Calculating Data Flow subsequences for real inputs   , length of sequence : none
Calculating Data Flow subsequences for constant calls, length of sequence : 4

-------------------------------------------------------------

AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
5               Transform subsequences to TTCN

0               Quit

Your choice? 4

NOT YET IMPLEMENTED
```

```
AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
5               Transform subsequences to TTCN

0               Quit

Your choice? 5

Which sequences do you want to have transformed to TTCN?

1               Control Flow, Data Flow and Link subsequences
2               Only Control Flow subsequences
3               Only Data Flow subsequences
4               Only Link subsequences

0               No transformation to TTCN

Your choice? 1

Transformation of Control Flow subsequences to TTCN
Transformation of UIO sequences to TTCN
Transformation of Data Flow subsequences to TTCN
Transformation of link subsequences to TTCN not yet implemented
```

```
AUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS


1               Create Extended Flow Graph out of Smile EFSM
2               Make UIO sequences for EFG flow states
3               Generate Data Flow testsubsequences
4               Generate linksequences
5               Transform subsequences to TTCN

0               Quit

Your choice? 0
```

```
/*****************************************************************************
 *
 * File            : typedef.h
 *
 * Description  :
 *          This header file contains all type definitions used in the other modules and all macro definitions
 *          for all used files in these modules.
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* TYPE DEFINITIONS */

typedef struct port_tp
{   char              name[80];             /* name of port */
    struct port_tp        *next;                /* next port */
} port_type;


typedef struct efg_tp
{   char        name[80];             /* specification name */
    port_type   *ports;               /* chain of ports */
    int         num_flow,             /* number of flowstates */
                num_input,            /* number of inputstates */
                num_output,           /* number of outputstates */
                num_intern,           /* number of internal states */
                num_choice,           /* number of choicestates */
                num_states,           /* number of total states */
                num_trans,            /* number of transitions */
                num_vars,             /* number of variables */
                num_basflow;          /* number of basic flowstates */
} efg_type;

typedef struct files_tp
{   FILE        *state1fp,            /* filepointer to file constructed to make file with pointer state2fp */
                *trans1fp,            /* filepointer to file containing all transitions of the EFG */
                *var1fp,              /* filepointer to file containing defined vars for each flowstate */
                *state2fp,            /* filepointer to file containing all states of the EFG */
                *var2fp,              /* filepointer to file containing all defined variables in the EFSM */
                *infp,                /* filepointer to file containing incoming transitions for flowstates */
                *outfp,               /* filepointer to file containing outgoing transitions for flowstates*/
                *basseqfp,            /* filepointer to file containing basic sequences for basic flowstates*/
                *basactfp,            /* filepointer to file containing actions of basic sequences */
                *basuniqfp,           /* filepointer to file giving existence of variables in basic actions */
                *totseqfp,            /* filepointer to file containing all calculated sequences */
                *totactfp,            /* filepointer to file containing all actions for the sequences*/
                *totuniqfp,           /* filepointer to file giving uniqueness (or not) for actions */
                *uiofp,               /* filepointer to file containing uio sequences for all basic states */
                *dflowfp,             /* filepointer to file containing all data flow subsequences */
                *linkfp,              /* filepointer to file containing link subsequences */
                *uiottcnfp,           /* filepointer to file containing TTCN uio sequences for basic states */
                *cflowttcnfp,         /* filepointer to file containing all TTCN control flow subsequences */
                *dflowttcnfp,         /* filepointer to file containing all TTCN data flow subsequences */
                *linkttcnfp,          /* filepointer to file containing TTCN link subsequences */
                *smilefp;             /* filepointer to input SMILE EFSM file */
} files_type;

typedef enum {start, flow, input, output, intern, choice}       state_sort_type;     /* type of state */
typedef enum {no, yes, half}                                    stable_type;        /* state stable? */
typedef enum {def, c_use, p_use}                                var_use_type;       /* def/use variable */
typedef enum {nothing, in, out}                                 var_io_type;        /* var 'input'/'output'? */
typedef enum {mn, m_efg, m_uio, gen_d, link_ss, ttcn}           module_type;        /* which module */

typedef struct
{   int         fstate;                         /* number of flowstate */
    int         found;                          /* showing if uio sequence is found for flowstate */
} uio_found_type;

typedef     struct
{   long    seq;                            /* position in total sequence file showing last end of file */
    long    act;                            /* position in total action file showing last end of file */
    long    uniq;                           /* position in total uniqueness file showing last end of file */
} ends_type;

typedef struct var_tp
{   char            name[80];               /* name of variable */
    var_use_type    use;                    /* def/p_use/c_use variable */
    var_io_type     io;                     /* variable is 'input'/'output' (io-df-chain)? */
    struct var_tp   *next;                  /* next variable of same transition/state */
} var_type;

typedef enum {pass,fail,inconclusive,none}      verdict_type;           /* pass, fail, inconclusive or none */

typedef struct ttcn_el_tp
{   char                action[200],            /* action in this element */
                        comment[500];
    verdict_type        verdict;                /* pass, fail, inconclusive or none */
    struct ttcn_el_tp   *nl,                    /* next level, the events to be executed after this event */
                        *np;                    /* next possibility, next alternative event */
    int                     indent,                 /* level of indention */
                        state;                  /* internal number of corresponding state */
```

```
} ttcn_el_type;

typedef struct case_tp
{  int          num_cflow,              /* number of control flow testcases */
                num_uio,                /* number of UIO sequence testcases */
                num_dflow,              /* number of data flow testcases */
                num_link;               /* number of link sequence testcases */
} case_type;

/* MACRO DEFINITIONS */

#define STATE_FILE_1     "efgfiles/state1.tmp"
#define TRANS_FILE_1     "efgfiles/trans1.efg"
#define VAR_FILE_1       "efgfiles/var1.tmp"
#define STATE_FILE_2     "efgfiles/state2.efg"
#define VAR_FILE_2       "efgfiles/var2.tmp"
#define IN_FILE          "efgfiles/intrans.tmp"
#define OUT_FILE         "efgfiles/outtrans.tmp"
#define BAS_SEQ_FILE     "efgfiles/basseq.uio"
#define BAS_ACT_FILE     "efgfiles/basact.uio"
#define BAS_UNIQ_FILE    "efgfiles/basuniq.uio"
#define TOT_SEQ_FILE     "efgfiles/totseq.uio"
#define TOT_ACT_FILE     "efgfiles/totact.uio"
#define TOT_UNIQ_FILE    "efgfiles/totuniq.uio"
#define UIO_FILE         "efgfiles/uio.seq"
#define DFLOW_FILE       "efgfiles/dflow.seq"
#define LINK_FILE        "efgfiles/link.seq"
#define UIO_TTCN_FILE    "efgfiles/uio.ttcn"
#define CFLOW_TTCN_FILE  "efgfiles/cflow.ttcn"
#define DFLOW_TTCN_FILE  "efgfiles/dflow.ttcn"
#define LINK_TTCN_FILE   "efgfiles/link.ttcn"
```

```
/*****************************************************************************
 *
 * File             : efghead.h
 *
 * Description  : This header file contains extern function declarations
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* FUNCTION DECLARATIONS */

/* MAIN.C */

#ifndef __STDC__
    extern  void    err();
#else
    extern  void    err(module_type, int, int);
#endif

/* MAKEFG.C */

#ifndef __STDC__
    extern  void    make_efg();
#else
    extern  void    make_efg(efg_type, files_type *);
#endif

/* MAKEUIO.C */

#ifndef __STDC__
    extern  void    find_line();
    extern  int         find_st_line_next_state();
    extern  char *  add_next_state();
    extern  char *  add_next_state_begin();
    extern  int         is_basic();
    extern  int         is_stable();
    extern  int         statenumber();
    extern  int         flownumber();
    extern  void    make_uio();
#else
    extern  void    find_line(char *, FILE *);
    extern  int         find_st_line_next_state(files_type *, char *);
    extern  char *  add_next_state(files_type *, char *);
    extern  char *  add_next_state_begin(files_type *, char *);
    extern  int         is_basic(int, files_type *);
    extern  int         is_stable(int, files_type *);
    extern  int         statenumber(int, files_type *);
    extern  int         flownumber(int, files_type *);
    extern  void    make_uio(efg_type *, files_type *);
#endif

/* GENDFLOW.C */

#ifndef __STDC__
    extern  int         is_inputstate();
    extern  void    gen_dflow();
#else
    extern  int         is_inputstate(int, files_type *);
    extern  void    gen_dflow(efg_type *, files_type *);
#endif

/* LINKSUBS.C */

#ifndef __STDC__
    extern  void    link_subseq();
#else
    extern  void    link_subseq(efg_type *, files_type *);
#endif

/* TTCN.C */

#ifndef __STDC__
    extern  void    to_ttcn();
#else
    extern  void    to_ttcn(efg_type *, files_type *);
#endif
```

```
/*****************************************************************************
 *
 * File             : main.c
 *
 * Description  :
 *          This module contains the function main which is the overall
 *          structure for calling the other submodules. Also this module
 *          contains the error function which is called in all modules when
 *          something is going wrong.
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* PREPROCESSOR COMMANDS */

#ifndef __STDC__
    static  void    print_menu();
    static  void    error_message();
#else
    static  void    print_menu(void);
    static  void    error_message(int);
#endif

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : main
 *
 * Description  :
 *      Main file for generating automatically testsequences for protocols.
 *      Gives menu for different aspects of the generation and calls the other
 *      files if chosen.
 *
 * Arguments    :
 *      int     argc        : number of arguments
 *      char    *argv[]     : arguments
 *
 * Returns      : int 0
 *
 *****************************************************************************/

main(argc, argv)
int         argc;
char    *argv[];
{
    efg_type    *efg;
    files_type  *files;
    char        inbuf[80];
    char        menu;

    efg = (efg_type *) malloc(sizeof(efg_type));
    files = (files_type *) malloc(sizeof(files_type));

    do
    {
        do
        {
            print_menu();
            gets(inbuf);
            menu = inbuf[0];
        } while (!('0' <= menu < '6'));

        switch(menu)
        {
        case '1':
                make_efg(efg, files);
                break;
        case '2':
                make_uio(efg, files);
                break;
        case '3':
                gen_dflow(efg, files);
                break;
        case '4':
                link_subseq(efg, files);
                break;
        case '5':
                to_ttcn(efg, files);
                break;
        case '0':
                break;
        default:
                break;
        }
    } while (menu != '0');
```

```
    free(efg);
    free(files);
    return(0);
}

/*****************************************************************************
 *
 * Function     : print_menu
 *
 * Description  :
 *          This function prints a menu for deriving testsequences on the screen.
 *
 * Arguments    : none
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
print_menu()
{
    printf("\n_____________________________________________________________\n");
    printf("\nAUTOMATIC TEST GENERATION FROM LOTOS SPECIFICATIONS\n\n");
    printf("\n1     Create Extended Flow Graph out of Smile EFSM");
    printf("\n2     Make UIO sequences for EFG flow states");
    printf("\n3     Generate Data Flow testsubsequences");
    printf("\n4     Generate linksequences");
    printf("\n5     Transform subsequences to TTCN");
    printf("\n\n0       Quit");
    printf("\n\nYour choice? ");
}

/*****************************************************************************
 *
 * Function     : err
 *
 * Description  :
 *          This function prints error messages and exits the program.
 *
 * Arguments    :
 *          module_type module      : module in which fault happened
 *          int         error       : number of error
 *                      parameter   : possible parameter to give extra info
 *
 *
 * Returns      : void
 *
 *****************************************************************************/

extern  void
err(module, error, parameter)
module_type module;
int         error,
            parameter;
{
    switch(error)
    {
    case 1:
            printf("\n%s cannot be opened", VAR_FILE_1);
            exit(1);
    case 2:
            printf("\n%s cannot be opened", TRANS_FILE_1);
            exit(2);
    case 3:
            printf("\n%s cannot be opened", STATE_FILE_1);
            exit(3);
    case 4:
            printf("\n%s cannot be opened", IN_FILE);
            exit(4);
    case 5:
            printf("\n%s cannot be opened", OUT_FILE);
            exit(5);
    case 6:
            printf("\n%s cannot be opened", STATE_FILE_2);
            exit(6);
    case 7:
            /* smile output file cannot be opened */
            exit(7);
    case 8:
            printf("\nInputfile not as expected:");
            printf("\n'State' not found after 'PROCESS'");
            exit(8);
    case 9:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File before specification name read");
            exit(9);
    case 10:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File before ports scanned");
            exit(10);
    case 11:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File before ports 'SPECIFICATION'");
            exit(11);
    case 12:
```

```
            printf("\nCalling write_state() with non-existing statetype");
            exit(12);
    case 13:
            printf("\n'STATE %d not found in varfile", parameter);
            exit(13);
    case 14:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File before ports 'BEHAVIOUR'");
            exit(14);
    case 15:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File before ports 'WHERE'");
            exit(15);
    case 16:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File at reading line in PROCESS State%d", parameter);
            exit(16);
    case 17:
            printf("\nCannot decide which action while reading line in State%d", parameter);
            exit(17);
    case 18:
            printf("\nInputfile not as expected:");
            printf("\nEnd Of File at reading rest header PROCESS State%d", parameter);
            exit(18);
    case 19:
            printf("\nInputfile not as expected:");
            printf("\n'ENDPROC' not found");
            exit(19);
    case 20:
            printf("\n%s cannot be opened", VAR_FILE_2);
            exit(20);
    case 21:
            printf("\n%s cannot be opened", BAS_SEQ_FILE);
            exit(21);
    case 22:
            printf("\n%s cannot be opened", BAS_ACT_FILE);
            exit(22);
    case 23:
            printf("\n%s cannot be opened", BAS_UNIQ_FILE);
            exit(23);
    case 24:
            printf("\n%s cannot be opened", TOT_SEQ_FILE);
            exit(24);
    case 25:
            printf("\n%s cannot be opened", TOT_ACT_FILE);
            exit(25);
    case 26:
            printf("\n%s cannot be opened", TOT_UNIQ_FILE);
            exit(26);
    case 27:
            printf("\n%s cannot be opened", UIO_FILE);
            exit(27);
    case 28:
            /* some line not found */
            exit(28);
    case 29:
            printf("\nEOF reached in %s when not expected", TRANS_FILE_1);
            exit(29);
    case 30:
            printf("\nEOF reached in %s when not expected", STATE_FILE_2);
            exit(30);
    case 31:
            printf("\n%s cannot be opened", DFLOW_FILE);
            exit(31);
    case 32:
            printf("\n%s cannot be opened", LINK_FILE);
            exit(32);
    case 33:
            printf("\n%s cannot be opened", CFLOW_TTCN_FILE);
            exit(33);
    case 34:
            printf("\n%s cannot be opened", UIO_TTCN_FILE);
            exit(34);
    case 35:
            printf("\n%s cannot be opened", DFLOW_TTCN_FILE);
            exit(35);
    case 36:
            printf("\n%s cannot be opened", LINK_TTCN_FILE);
            exit(36);
    case 37:
            printf("\nUnknown verdict");
            exit(37);
    default:
            printf("\nUnknown error, sorry, have a nice day.\n");
            exit(666);
    }
}
```

```
/*****************************************************************************
 *
 * File             : makeefg.c
 *
 * Description  :
 *          This module creates an Extended Flow Graph (EFG) out of a description of a protocol as an EFSM as
 *          created by Smile, the symbolic simulator of the LITE-toolset. If you choose this module from the menu
 *          in main, the name of the Smile EFSM will be asked, after which this EFSM will be scanned and transformed
 *          to some files of the suitable form for usage by the other modules. These files are:
 *          - VAR_FILE_1: This file contains for all flowstates (processes) in the EFSM the defined variables in the
 *          header. The structure of this file is as follows: For every flowstate a line 'STATE <num>', where <num>
 *          the number of the state is as indicated in the process. After such a line a new line is created for each
 *          defined variable and consisting only of the name of this variable.
 *          - VAR_FILE_2: This file contains all defined variables in the EFSM, each given on a separate line. This
 *          file is not used in this module.
 *          - STATE_FILE_1: This file is actually a temporary file to construct STATE_FILE_2. The structure is the
 *          same as the structure of STATE_FILE_2, except for the incoming and outgoing transitions of flowstates
 *          which are given in the files IN_FILE en OUT_FILE.
 *          - IN_FILE: This file contains the incoming transitions of the flowstates. During scanning each time
 *          such a call to a flowstate is found, a line '<statenum> <trans>' is written to this file; <statenum> is
 *          the number of the flowstate, <trans> is the internal number of the transition (see TRANS_FILE_1).
 *          - OUT_FILE: This file contains the outgoing transitions of the flowstates. The structure is the same as
 *          the structure of IN_FILE.
 *          - The files STATE_FILE_2 en TRANS_FILE_1 are the files that describe the EFG. In STATE_FILE_2 all
 *          flowstates, inputactions, outputactions, choice-actions and internal events are described as states.
 *          These states are connected by transitions which are given in TRANS_FILE_1. The structures of these
 *          files are as follows:
 *          - STATE_FILE_2: For each state the following lines are given:
 *              '@ST <int_num>' : <int_num> is the statenumber used to discern the various flowstates and actions.
 *                  This number is also used in TRANS_FILE_1.
 *              '<type> <num>' : <type> gives the type of the state. This can be S (startstate), F (flowstate),
 *                  I (inputaction), O (outputaction), X (internal event) and C (choice-action). <num> gives the
 *                  number for a certain type of state. For example for flowstates this is the number as indicated
 *                  in the process.
 *              '@ACT <action>': <action> contains in case of an inputaction or outputaction this action. In other
 *                  cases the line consists only of '@ACT'.
 *              '@VAR <var_type>' : In case of a flowstate is looked in VAR_FILE_1 if there are variables defined
 *                  in the header of this flowstate. If so, <var_type> is 'd' and this line is followed by lines
 *                  each containing such a variable. In case of an inputaction <var_type> is 'd' and the line is
 *                  followed by a line containing the inputvariable. In case of an outputaction is looked if there
 *                  are any defined variables used in this outputaction. If so, <var_type> becomes 'c' and the line
 *                  is followed by lines containing these outputvariables. In case of a choicestate <var_type> is
 *                  'd' and the line is followed by a line containing the choicevariable. In other cases the line
 *                  consists only of '@VAR'.
 *              '@IN' : This line is followed by a certain number of lines, each containing a transitionnumber
 *                  as defined in TRANS_FILE_1, which leads to this state (incoming transitions).
 *              '@OUT' : This line is followed by a certain number of lines, each containing a transitionnumber
 *                  as defined in TRANS_FILE_1, which leads from this state (outgoing transitions).
 *          - TRANS_FILE_1: For each transition the following lines are given:
 *              '@TR <tr_num>' : <tr_num> is the transitionnumber used to discern all transitions used in this file.
 *                  This number is also used as incoming and outgoing transitions in STATE_FILE_2.
 *              '@FROM <st_num>' : This line indicates in which state the transition starts. <st_num> is a
 *                  statenumber, as given in the '@ST <int_num>' line in STATE_FILE_2.
 *              '@TO <st_num>': This line indicates in which state the transition ends. <st_num> is a statenumber,
 *                  as given in the '@ST <int_num>' line in STATE_FILE_2.
 *              '@VAR <var_type>' : In case of a transition containing predicates ('[..] ->' in the Smile EFSM),
 *                  <var_type> is 'p' and the line is followed by a number of lines, each giving a variable used in
 *                  the predicates. In case of a call to a flowstate which defines variables in his header
 *                  (parameters), <var_type> is 'c'. This line is then followed by a line for each parameter. In
 *                  case that one or more variables are used to form a certain parameter, these variables are given
 *                  in this line seperated by tabs. In case a constant value is given to the parameter, the line
 *                  consists of a tab followed by '@CONST'. If the transition contains no predicates and there is
 *                  no call to a flowstate containing parameters, the line consists only of '@VAR'.
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : open_files_efg
 *
 * Description  :
 *          This function tries to open the necessary files in this module. Also it asks for the Smile EFSM file
 *          and tries to open it.
 *
 * Arguments    :
 *          files_type  *files : pointer to all filepointers
 *
 * Returns      : int
 *          0: all OK
 *          1: unable to open Smile EFSM file
 *
 *****************************************************************************/
```

```
static  int
open_files_efg(files)
files_type  *files;
{
    char    inbuf[80];

    if ((files->var1fp = fopen(VAR_FILE_1, "w+")) == NULL)
        err(m_efg, 1, 0);
    if ((files->trans1fp = fopen(TRANS_FILE_1, "w")) == NULL)
        err(m_efg, 2, 0);
    if ((files->state1fp = fopen(STATE_FILE_1, "w+")) == NULL)
        err(m_efg, 3, 0);
    if ((files->infp = fopen(IN_FILE, "w+")) == NULL)
        err(m_efg, 4, 0);
    if ((files->outfp = fopen(OUT_FILE, "w+")) == NULL)
        err(m_efg, 5, 0);
    if ((files->state2fp = fopen(STATE_FILE_2, "w")) == NULL)
        err(m_efg, 6, 0);
    if ((files->var2fp = fopen(VAR_FILE_2, "w")) == NULL)
        err(m_efg, 20, 0);

    printf("\nGive name of Smile output (LOTOS EFSM): ");
    gets(inbuf);
    if ((files->smilefp = fopen(inbuf, "r")) == NULL)
    {
        printf("\n%s cannot be opened", inbuf);
        fclose(files->var1fp);
        fclose(files->trans1fp);
        fclose(files->infp);
        fclose(files->state1fp);
        fclose(files->outfp);
        fclose(files->state2fp);
        fclose(files->var2fp);
        return(1);
    }
    return(0);
}

/*****************************************************************************
 *
 * Function     : close_files_efg
 *
 * Description  :
 *          This function closes all files used in this module.
 *
 * Arguments    :
 *          files_type  *files : pointer to all filepointers
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
close_files_efg(files)
files_type  *files;
{
    fclose(files->var1fp);
    fclose(files->trans1fp);
    fclose(files->infp);
    fclose(files->state1fp);
    fclose(files->outfp);
    fclose(files->state2fp);
    fclose(files->var2fp);
    fclose(files->smilefp);
}

/*****************************************************************************
 *
 * Function     : init_efg
 *
 * Description  :
 *          This function resets all num_xxx variables in the efg to 0.
 *
 * Arguments    :
 *          efg_type    *efg: pointer to efg to be resetted
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
init_efg(efg)
efg_type    *efg;
{
    efg->num_flow = 0;
    efg->num_input = 0;
    efg->num_output = 0;
    efg->num_intern = 0;
    efg->num_choice = 0;
    efg->num_states = 0;
    efg->num_trans = 0;
    efg->num_vars = 0;
    efg->num_basflow = 0;
}
```

```
/*****************************************************************************
 *
 * Function     : calc_vars
 *
 * Description  :
 *          This function calculates VAR_FILE_2. In this file all variables defined in de Smile EFSM file are given,
 *          each on a new line. This file is not used in this module.
 *
 * Arguments    :
 *          files_type  *files: pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
calc_vars(files)
files_type  *files;
{
    int         c,
                i = 0;
    char    variable[50];

    strcpy(variable, "");
    c = fgetc(files->smilefp);
    while (c != EOF)
    {
        if (isalnum(c) || (c == '_'))                                   /* possible variable */
        {
            variable[i++] = (char)c;
            variable[i] = '\0';
        }
        else    if ((c == ':') && (strcmp(variable, "") != 0))          /* variable */
                {
                    fprintf(files->var2fp, "%s\n", variable);
                    strcpy(variable, "");
                    i = 0;
                }
                else                                                    /* no part of variable */
                {
                    strcpy(variable, "");
                    i = 0;
                }

        c = fgetc(files->smilefp);
    }
}

/*****************************************************************************
 *
 * Function     : init_var
 *
 * Description  :
 *          This function creates a pointer to a variable of type var_type, initiates the various members of this
 *          type and returns the pointer.
 *
 * Arguments    : none
 *
 * Returns      : var_type * : pointer to initiated variable
 *
 *****************************************************************************/

static var_type *
init_var()
{
    var_type    *temp;

    temp = (var_type *) malloc(sizeof(var_type));
    strcpy(temp->name, "");
    temp->use = def;
    temp->io = nothing;
    temp->next = NULL;
    return(temp);
}

/*****************************************************************************
 *
 * Function     : init_port
 *
 * Description  :
 *          This function creates a pointer to a variable of type port_type, initiates the various members of this
 *          type and returns the pointer.
 *
 * Arguments    : none
 *
 * Returns      : port_type * : pointer to initiated variable
 *
 *****************************************************************************/

static port_type *
init_port()
{
    port_type   *temp;

    temp = (port_type *) malloc(sizeof(port_type));
```

```
    strcpy(temp->name, "");
    temp->next = NULL;
    return(temp);
}

/*****************************************************************************
 *
 * Function     : white
 *
 * Description  :
 *          This function checks if the character argument is equal to a space, newline or tab.
 *
 * Arguments    :
 *          char    c : character to be checked
 *
 * Returns      : int
 *          0: c is not equal to space, newline or tab
 *          1: c is equal to space, newline or tab
 *
 *****************************************************************************/

static  int
white(c)
char    c;
{
    if ((c == ' ') || (c == '\n') || (c == '\t'))
        return(1);
    else
        return(0);
}

/*****************************************************************************
 *
 * Function     : read_statevars
 *
 * Description  :
 *          This function creates VAR_FILE_1. It scans all headers of processes in de Smile EFSM, extracts the
 *          statenumbers of the flowstates and the defined variables in the header of the various flowstates.
 *          For the structure of this file: see Description module make_efg.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all used files in this module
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
read_statevars(files)
files_type  *files;
{
    char    s[80];
    int         statenum = 0;
    int         i, c;
    int         last;

    rewind(files->smilefp);
    while (fscanf(files->smilefp, "%s", s) != EOF)                      /* end of file reached */
    {
        if (strcmp(s, "PROCESS") == 0)
        {
            fscanf(files->smilefp, "%s", s);
            if ((s[0] != 'S') ||(s[1] != 't') ||(s[2] != 'a') ||(s[3] != 't') ||(s[4] != 'e'))
                err(m_efg, 8, 0);                                       /* check if word after PROCESS is State */

            strcpy(s, &s[5]);                                           /* remove 'State' */
            statenum = atoi(s);                                             /* number of State */
            fprintf(files->varlfp, "\nSTATE %d\n", statenum);

            while ((c = fgetc(files->smilefp)) != ']');                 /* ignore ports */
            while (white(c = fgetc(files->smilefp)));                   /* ignore possible white */
            if (c == '(')                                               /* start of variable declarations */
            {
                do
                {
                    last = 0;
                    fscanf(files->smilefp, "%s", s);
                    for (i = 0; s[i] != '\0'; i++)
                    {
                        if (s[i] == ')')
                            last = 1;
                        if (s[i] == ':')                                /* remove ':type' */
                            s[i] = '\0';
                    }
                    fprintf(files->varlfp, "%s\n", s);
                } while (last == 0);
            }
        }
    }
    printf("\nState variables written to %s", VAR_FILE_1);
}

/*****************************************************************************
 *
 * Function     : scan_name
```

```
 *
 * Description  :
 *          This function reads the name of the specification and returns it in name.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files used in this module
 *          char        name[80]: returns name of specification
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
scan_name(files, name)
files_type  *files;
char        name[80];
{
    int         c;
    int         i = 0;

    do
        c = fgetc(files->smilefp);                                  /* remove possible white */
    while ((white(c)) && (c != EOF));

    strcpy(name, "");
    while ((!white(c)) && (c != EOF) && (c != '['))
    {
        name[i++] = c;
        name[i] = '\0';
        c = fgetc(files->smilefp);
    }
    if (c == '[') ungetc(c, files->smilefp);

    printf("\n\nspec: %s", name);

    if (c == EOF) err(m_efg, 9, 0);
}

/******************************************************************************
 *
 * Function     : scan_ports
 *
 * Description  :
 *          This function scans the ports which are defined in the header of the specification and returns them
 *          via the variable ports.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          port_type   *ports  : pointer to variable containing all portnames
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
scan_ports(files, ports)
files_type  *files;
port_type   *ports;
{
    int         c;
    int             i = 0;
    port_type   *prt;

    do
        c = fgetc(files->smilefp);                                  /* remove possible white */
    while ((white(c)) && (c != EOF));

    if (c != EOF)                                                   /* c = '[' */
    {
        prt = ports;
        c = fgetc(files->smilefp);
        while ((c != ']') && (c != ',') && (c != EOF))
        {
            prt->name[i++] = c;                                         /* get first port */
            prt->name[i] = '\0';
            c = fgetc(files->smilefp);
        }
        printf("\nport: %s", prt->name);
        while ((c != ']') && (c != EOF))                            /* c = ',' => next port */
        {
            prt->next = init_port();
            prt = prt->next;
            i = 0;
            c = fgetc(files->smilefp);                              /* space after ',' */
            c = fgetc(files->smilefp);
            while ((c != ']') && (c != ',') && (c != EOF))
            {
                prt->name[i++] = c;                                     /* get next port */
                prt->name[i] = '\0';
                c = fgetc(files->smilefp);
            }
            printf("\nport: %s", prt->name);
        }
    }
```

```
    if ((c == EOF)) err(m_efg, 10, 0);
}

/*****************************************************************************
 *
 * Function     : read_spec
 *
 * Description  :
 *          This function reads the header of the specification, e.g. it calls scan_name() and scan_ports().
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to efg
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
read_spec(efg, files)
efg_type    *efg;
files_type  *files;
{
    char    s[80];

    rewind(files->smilefp);
    do
    {
        if (fscanf(files->smilefp, "%s", s) == EOF) err(m_efg, 11, 0);
    } while (strcmp(s, "SPECIFICATION") != 0);

    scan_name(files, efg->name);

    efg->ports = init_port();
    scan_ports(files, efg->ports);

    fgets(s, 79, files->smilefp);                                   /* scan rest of line */

    printf("\n\nSPECIFICATION-header scanned\n");
}

/*****************************************************************************
 *
 * Function     : exist_flow
 *
 * Description  :
 *          This function checks whether the flowstate with number <flowstate> already exists in the file designated
 *          by fp. If so, it assigns the internal statenumber of the found state to <intern_num> and returns the
 *          value 1. If not so, it returns 0.
 *
 * Arguments    :
 *          FILE    *fp         : pointer to file in which the states of the EFG are constructed
 *          int         flowstate   : number of flowstate as in 'PROCESS StateX'
 *          int         *intern_num     : number of flowstate as used in statefile, line '@ST X'
 *
 * Returns      : int
 *          0: flowstate not found
 *          1: flowstate found
 *
 *****************************************************************************/

static int
exist_flow(fp, flowstate, intern_num)
FILE    *fp;
int         flowstate,
        *intern_num;
{
    char    line[80],
            inbuf[80],
            word1[80],
            word2[80];

    rewind(fp);
    sprintf(line, "F %d\n", flowstate);
    do
    {
        fgets(inbuf, 79, fp);
        sscanf(inbuf, "%s", word1);
        if(strcmp(word1, "@ST") == 0)
        {
            sscanf(inbuf, "%s%s", word1, word2);
            *intern_num = atoi(word2);                              /* get possible intern statenumber */
        }
    } while ((!feof(fp)) && (strcmp(inbuf, line) != 0));            /* EOF or flowstate exists */

    fflush(fp);
    if (feof(fp))
    {
        return(0);
    }
    else                                                            /* flowstate exists */
    {
        fseek(fp, 0L, 2);
        return(1);
    }
```

```
}

/*****************************************************************************
 *
 * Function     : update_inout
 *
 * Description  :
 *          This function creates STATE_FILE_2 out of STATE_FILE_1, IN_FILE and OUT_FILE. It copies line by line
 *          STATE_FILE_1 until it reaches a the '@IN'-part of a flowstate. It searches IN_FILE for the right
 *          incoming transitions, OUT_FILE for the right outgoing transitions and adds them to STATE_FILE_2.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
update_inout(files)
files_type  *files;
{
    char    inbuf[200],
            word[80];
    int         flowstate,
            flowstate2,
            trans;

    printf("\n\nAdding incoming and outgoing transitions: Flowstate    ");
    rewind(files->state1fp);
    while (!feof(files->state1fp))
    {
        strcpy(word, "");
        do
        {
            fgets(inbuf, 199, files->state1fp);
            if (!feof(files->state1fp)) fputs(inbuf, files->state2fp);
            sscanf(inbuf, "%s", word);
        } while((!feof(files->state1fp)) && (strcmp(word, "F") != 0));              /* search first flowstate */

        if (!feof(files->state1fp))                                                 /* flowstate found */
        {
            sscanf(inbuf, "%s%d", word, &flowstate);
            printf("\b\b\b%-3d", flowstate);
            fflush(stdout);

            do
            {
                fgets(inbuf, 199, files->state1fp);
                if (!feof(files->state1fp)) fputs(inbuf, files->state2fp);
            } while((!feof(files->state1fp)) && (strcmp(inbuf, "@IN\n") != 0));             /* copy till line after
*/

            if (!feof(files->state1fp))
            {
                rewind(files->infp);
                rewind(files->outfp);
                while (!feof(files->infp))                                  /* add all incoming transitions */
                {
                    fgets(inbuf, 199, files->infp);
                    sscanf(inbuf, "%d%d", &flowstate2, &trans);
                    if ((flowstate == flowstate2) && (!feof(files->infp)))
                        fprintf(files->state2fp, "%d\n", trans);
                }
                fgets(inbuf, 199, files->state1fp);
                fputs(inbuf, files->state2fp);                              /* copy @OUT-line */
                while (!feof(files->outfp))                                 /* add all outgoing transitions */
                {
                    fgets(inbuf, 199, files->outfp);
                    sscanf(inbuf, "%d%d", &flowstate2, &trans);
                    if ((flowstate == flowstate2) && (!feof(files->outfp)))
                        fprintf(files->state2fp, "%d\n", trans);
                }
            }
        }
    }
    printf("\b\b\b\b\b\b\b\b\b\b\b\b ready      ");
}

/*****************************************************************************
 *
 * Function     : status_efg
 *
 * Description  :
 *          This function prints on standard output some statistics about the efg, e.g. number of various states.
 *
 * Arguments    :
 *          efg_type    *efg : pointer to efg
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
status_efg(efg)
```

```
efg_type    *efg;
{
    printf("\n\n%d processes scanned, End Of File (Smile EFSM) reached", efg->num_flow);
    printf("\nExtended Flow Graph consists of %d states en %d transitions", efg->num_states, efg->num_trans);
    printf("\nThese states can be subdivided into:");
    printf("\n1 Start State");
    printf("\n%d Control Flow States", efg->num_flow);
    printf("\n%d Input States", efg->num_input);
    printf("\n%d Output States", efg->num_output);
    printf("\n%d Choice States", efg->num_choice);
    printf("\n%d Internal States\n", efg->num_intern);
}

/*****************************************************************************
 *
 * Function     : write_state
 *
 * Description  :
 *          This function writes all necessary info about a state in an writefile. This info consists of the
 *          following lines:
 *              @ST <intern_statenum>
 *              <state_type> <statenum>
 *              @ACT <action>
 *              @VAR <var_type>
 *              lines containing a variable
 *              @IN
 *              lines containing an incoming transition
 *              @OUT
 *              lines containing outgoing transitions
 *          The exact structure is described in the Description module make_efg.
 *
 * Arguments    :
 *          FILE            *writefp        : pointer to file where state is to be written to
 *          state_sort_type type_of_state   : parameter indicating which type the state has (flow, input etc)
 *          int                 flowstate       : gives in case of a flowstate the statenumber as in PROCESS StateX
 *          char            act[80]             : gives in case of an input or output action the complete action
 *          var_type        *statevars      : in case of a flowstate: contains all defined variables in processheader
 *                                            in case of an inputstate: contains the inputvariable if existing
 *                                            in case of an outputstate: contains all variables used in outputstate
 *                                            in case of a choicestate: contains choicevariable
 *          int                 ins, outs       : gives for all statetypes except flowstates the incoming and the
outgoing
 *                                            transition
 *          efg_type        *efg            : pointer to efg
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
write_state(writefp, type_of_state, flowstate, act, statevars, ins, outs, efg)
FILE            *writefp;
state_sort_type type_of_state;
int                 flowstate;
char            act[80];
var_type        *statevars;
int                 ins, outs;
efg_type        *efg;
{
    var_type    *temp_var;
    int             intern_num,
                c_use_exist = 0;

    switch (type_of_state)
    {
    case start:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                fprintf(writefp, "S 1\n");
                fprintf(writefp, "@ACT\n");
                fprintf(writefp, "@VAR\n");
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    case flow:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                efg->num_flow++;
                fprintf(writefp, "F %d\n", flowstate);
                fprintf(writefp, "@ACT\n");
                fprintf(writefp, "@VAR");
                temp_var = statevars;                   /* statevars consists of all defined vars in processheader */
                if (strcmp(temp_var->name, "") != 0)                                    /* minimal one variable */
                {
                    fprintf(writefp, " d\n");
                    while (strcmp(temp_var->name, "") != 0)                         /* for all variables */
                    {
                        fprintf(writefp, "%s\n", temp_var->name);
                        temp_var = temp_var->next;
                    }
                }
                else                                                                /* no variables defined */
                {
                    fprintf(writefp, "\n");
```

```
                }
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    case input:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                efg->num_input++;
                fprintf(writefp, "I %d\n", efg->num_input);
                fprintf(writefp, "@ACT %s\n", act);                                    /* act: total inputaction */
                fprintf(writefp, "@VAR ");
                if (strcmp(statevars->name, "") != 0)                                  /* input = variable */
                {
                    fprintf(writefp, "d\n");
                    fprintf(writefp, "%s\n", statevars->name);                         /* statevars: inputvariable */
                }
                else fprintf(writefp, "\n");
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    case output:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                efg->num_output++;
                fprintf(writefp, "O %d\n", efg->num_output);
                fprintf(writefp, "@ACT %s\n", act);                                    /* act: total outputaction */
                fprintf(writefp, "@VAR");
                temp_var = statevars;                          /* statevars: all defined/used variables in process */
                while (strcmp(temp_var->name, "") != 0)
                {
                    if ((temp_var->use == c_use) && (c_use_exist == 0))        /* first time that a c_use is found */
                    {
                        c_use_exist = 1;
                        fprintf(writefp, " c\n");
                        fprintf(writefp, "%s\n", temp_var->name);
                    }
                    else if ((temp_var->use == c_use) && (c_use_exist == 1))       /* next times c_use found */
                        fprintf(writefp, "%s\n", temp_var->name);
                    temp_var = temp_var->next;
                }
                if (c_use_exist == 0) fprintf(writefp, "\n");                      /* c_use is not found */
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    case intern:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                efg->num_intern++;
                fprintf(writefp, "X %d\n", efg->num_intern);
                fprintf(writefp, "@ACT %s\n", act);
                fprintf(writefp, "@VAR\n");
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    case choice:
                fprintf(writefp, "@ST %d\n", ++efg->num_states);
                efg->num_choice++;
                fprintf(writefp, "C %d\n", efg->num_choice);
                fprintf(writefp, "@ACT\n");
                fprintf(writefp, "@VAR d\n");
                fprintf(writefp, "%s\n", statevars->name);                     /* statevars: choice variable */
                fprintf(writefp, "@IN\n");
                if (ins != 0) fprintf(writefp, "%d\n", ins);
                fprintf(writefp, "@OUT\n");
                if (outs != 0) fprintf(writefp, "%d\n", outs);
                fprintf(writefp, "\n");
                break;
    default:
                err(m_efg, 12, 0);
    }
}

/*****************************************************************************
 *
 * Function     : write_basic_trans
 *
 * Description  :
 *          This function writes the basic structure of a transition to the writefile. This structure consists of the
 *          following lines:
 *              @TR <tr_num>            : <tr_num> is number of transition
 *              @FROM <st_num>          : <st_num> is a number of a state in the statefile where the transition starts
 *              @TO <st_num>            : <st_num> is a number of a state in the statefile where the transition ends
 *              @VAR
 *
 * Arguments    :
 *          FILE        *writefp    : file where transition is to be written to
```

```
 *          efg_type    *efg        : pointer to efg
 *          int             fromstate   : number of state where transition starts
 *          int             tostate         : number of state where transition ends
 *
 * Returns      : void
 *
 ******************************************************************************/

static void
write_basic_trans(writefp, efg, fromstate, tostate)
FILE        *writefp;
efg_type    *efg;
int             fromstate;
int             tostate;
{
    fprintf(writefp, "@TR %d\n", ++efg->num_trans);
    fprintf(writefp, "@FROM %d\n", fromstate);
    fprintf(writefp, "@TO %d\n", tostate);
    fprintf(writefp, "@VAR");
}

/******************************************************************************
 *
 * Function     : scan_state
 *
 * Description  :
 *          This function scans the number of a flowstate X in StateX, and assigns it to parameter statenum. Also
 *          it ignores further characters until ], e.g. it ignores the ports.
 *
 * Arguments    :
 *          files_type  *files      : pointer to files
 *          int             *statenum   : pointer to integer variable which contains the flowstatenumber after scanning
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
scan_state(files, statenum)
files_type  *files;
int             *statenum;
{
    char    s[80];
    int         c;

    fscanf(files->smilefp, "%s", s);
    if ((s[0] != 'S') ||(s[1] != 't') ||(s[2] != 'a') ||(s[3] != 't') ||(s[4] != 'e'))
        err(m_efg, 8, 0);

    strcpy(s, &s[5]);                                           /* remove 'State' */
    *statenum = atoi(s);                                        /* get number State */
    while ((c = fgetc(files->smilefp)) != ']');                 /* ignore ports */
}

/******************************************************************************
 *
 * Function     : get_vars
 *
 ** Description :
 *          This function searches the readfile (varfile) for a given flowstate and fills statevars with the found
 *          variables for this flowstate
 *
 * Arguments    :
 *          FILE        *readfp         : pointer to file containing variables for flowstates
 *          int             flowstate   : number of flowstate as in StateX
 *          var_type    *statevars  : returns all variables of the flowstate
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
get_vars(readfp, flowstate, statevars)
FILE        *readfp;
int             flowstate;
var_type    *statevars;
{
    char        inbuf[80];
    char        compare[80];
    var_type    *temp_var;

    rewind(readfp);
    temp_var = statevars;
    sprintf(compare, "STATE %d\n", flowstate);

    do
    {
        fgets(inbuf, 79, readfp);
    } while ((!feof(readfp)) && (strcmp(compare, inbuf) != 0));             /* find STATE met statenumber */

    if (!feof(readfp))
    {
        if (fscanf(readfp, "%s", inbuf) != 0)                               /* EOF */
        {
            while ((strcmp(inbuf, "STATE") != 0) && (!feof(readfp)))
```

```
            {
                strcpy(temp_var->name, inbuf);                          /* get variable */
                temp_var->next = init_var();
                temp_var = temp_var->next;
                fscanf(readfp, "%s", inbuf);
            }
        )
    )
    else err(m_efg, 13, flowstate);
}

/*****************************************************************************
 *
 * Function     : free_var_chain
 *
 * Description  :
 *          This function frees a variable chain of type var_type
 *
 * Arguments    :
 *          var_type    *statevars : the variable to be freed
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
free_var_chain(statevars)
var_type    *statevars;
{
    if (statevars->next != NULL) free_var_chain(statevars->next);
    free(statevars);
}

/*****************************************************************************
 *
 * Function     : scan_pars
 *
 * Description  :
 *          This function scans the variables which are used while calling a flowstate and writes them to the
 *          writefile. If one or more variables are used for a parameter these variables are written on a line
 *          seperated and preceeded by a tab. If there are no variables used a tab followed by @CONST is written
 *          to the writefile.
 *
 * Arguments    :
 *          FILE        *smilefp    : pointer to Smile file
 *          FILE        *writefp    : pointer to transfile
 *          var_type    *statevars  : structure containing all defined variables in this process, so all variables
 *                                    which can be used while calling
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
scan_pars(smilefp, writefp, statevars)
FILE        *smilefp;
FILE        *writefp;
var_type    *statevars;
{
    var_type    *temp_var;
    char        parameter[80];
    int             i, c;
    int             parenthese = 1;
    int             const;

    fprintf(writefp, " c\n");

    do
    {
        temp_var = statevars;
        while (white(c = getc(smilefp)));                               /* remove possible white */

        /* c == , && par == 1: next parameter reached;       par == 0: end of parameters reached */
        for (i = 0; !(((c == ',') && (parenthese == 1)) || (parenthese == 0)); i++)
        {
            if (c == '(') parenthese++;
            if (c == ')') parenthese--;
            if (parenthese != 0) parameter[i] = c;
            c = fgetc(smilefp);
        }
        parameter[i] = '\0';

        const = 1;
        while (strcmp(temp_var->name, "") != 0)                         /* check all variables if they are used */
        {
            if (strstr(parameter, temp_var->name) != NULL)              /* variable is used */
            {
                const = 0;
                fprintf(writefp, "\t%s", temp_var->name);
            }
            temp_var = temp_var->next;
        }
        if (const == 1)                                                 /* no variables used */
            fprintf(writefp, "\t@CONST\n");
        else
```

```
            fprintf(writefp, "\n");
    } while (parenthese != 0);
    fprintf(writefp, "\n");
}

/*****************************************************************************
 *
 * Function     : read_behav
 *
 * Description  :
 *          Function that scans the behaviour-part of the Smile EFSM, e.g. it writes the startstate, it scans to
 *          which flowstate this startstate calls and writes this flowstate. Also it checks for possible
 *          parameter calling and constructs the first transition.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to efg
 *          files_type  *files  : pointer to files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
read_behav(efg, files)
efg_type    *efg;
files_type  *files;
{
    char        s[80];
    int             c;
    int             flowstate;
    char        act[80];
    var_type    *statevars;

    do
    {
        if (fscanf(files->smilefp, "%s", s) == EOF) err(m_efg, 14, 0);
    } while (strcmp(s, "BEHAVIOUR") != 0);

    strcpy(act, "");
    statevars = init_var();
    write_state(files->state1fp, start, 1, act, statevars, 0, 1, efg);                  /* write startstate */

    scan_state(files, &flowstate);
    get_vars(files->var1fp, flowstate, statevars);
    write_state(files->state1fp, flow, flowstate, act, statevars, 0, 0, efg);           /* write first called state */
    fprintf(files->infp, "%d 1\n", flowstate);                                          /* add incoming transition */
    free_var_chain(statevars);

    statevars = init_var();
    write_basic_trans(files->trans1fp, efg, 1, 2);                              /* write trans from start to first */
    while (((c = fgetc(files->smilefp)) != '\n') && (c != '('));
    if (c == '(')                                                               /* parameters are used */
        scan_pars(files->smilefp, files->trans1fp, statevars);
    else
        fprintf(files->trans1fp, "\n\n");
    free_var_chain(statevars);

    do
    {
        if (fscanf(files->smilefp, "%s", s) == EOF) err(m_efg, 15, 0);
    } while (strcmp(s, "WHERE") != 0);                                          /* ignore until WHERE */

    printf("\nBEHAVIOUR-part scanned\n");
}

/*****************************************************************************
 *
 * Function     : do_choice
 *
 * Description  :
 *          This function scans the choice variable, adds it to the chain of variables used in this process, writes
 *          the choicestate and writes the transition to this choicestate.
 *
 * Arguments    :
 *          files_type  *files      : pointer to files
 *          efg_type    *efg        : pointer to efg
 *          char        inbuf[80]   : line of Smile file containing choice
 *          var_type    *statevars  : chain of defined variables in this process (flowstate)
 *          int             intern_num  : statenumber of flowstate as in StateX
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
do_choice(files, efg, inbuf, statevars, intern_num)
files_type  *files;
efg_type    *efg;
char        inbuf[80];
var_type    *statevars;
int             intern_num;
{
    var_type    *temp_var;
    char        act[80],
                rest[40],
```

```
                ch_var[40];
    int             i, c;

    sscanf(inbuf, "%s%s", rest, ch_var);
    for (i = 0; ch_var[i] != '\0'; i++)                                    /* get choice variable */
    {
        if (ch_var[i] == ':') ch_var[i] = '\0';
    }

    temp_var = statevars;
    while (temp_var->next != NULL) temp_var = temp_var->next;
    strcpy(temp_var->name, ch_var);                                        /* add choice variable to varchain */
    temp_var->next = init_var();
    strcpy(act, "");

    write_state(files->state1fp, choice, 0, act, temp_var, efg->num_trans + 1, efg->num_trans + 2, efg);

    write_basic_trans(files->trans1fp, efg, intern_num, efg->num_states);
    fprintf(files->trans1fp, "\n\n");
}

/*****************************************************************************
 *
 * Function     : write_trans
 *
 * Description  :
 *          This function writes the the transition from the flowstate or choicestate to the actionstate including
 *          possible variables used in predicates for this transition.
 *
 * Arguments    :
 *          int             choice      : equals 1 if fromstate is choicestate, else 0
 *          int             pred        : equals 1 if predicates are used for this transition, else 0
 *          int             intern_num  : number of flowstate as in @ST <num>
 *          files_type  *files      : pointer to files
 *          efg_type    *efg        : pointer to efg
 *          var_type    *statevars  : chain of all defined and possibly p-used variables for this transition
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
write_trans(choice, pred, intern_num, files, efg, statevars)
int             choice,
            pred,
            intern_num;
files_type  *files;
efg_type    *efg;
var_type    *statevars;
{
    var_type    *temp_var;

    if (choice == 1)                                                       /* trans starts in choice state */
        write_basic_trans(files->trans1fp, efg, efg->num_states, efg->num_states + 1);
    else                                                                   /* trans starts in flowstate */
        write_basic_trans(files->trans1fp, efg, intern_num, efg->num_states + 1);

    if (pred == 1)                                                         /* trans contains predicates */
    {
        fprintf(files->trans1fp, " p\n");
        temp_var = statevars;
        while (strcmp(temp_var->name, "") != 0)
        {
            if (temp_var->use == p_use)                                    /* print predicate variable */
                fprintf(files->trans1fp, "%s\n", temp_var->name);
            temp_var = temp_var->next;
        }
        fprintf(files->trans1fp, "\n");
    }
    else fprintf(files->trans1fp, "\n\n");                                 /* trans contains no predicates */
}

/*****************************************************************************
 *
 * Function     : port_cmp
 *
 * Description  :
 *          This function checks whether a port is present in character string <inbuf>. If so, it returns 1, else 0.
 *
 * Arguments    :
 *          port_type   *ports      : ports used in this EFSM
 *          char        inbuf[400]  : string containing line of Smile file
 *
 * Returns      : int
 *          0: no port is present in the string
 *          1: at least one port is present in the string
 *
 *****************************************************************************/

static int
port_cmp(ports, inbuf)
port_type   *ports;
char        inbuf[400];
{
    port_type   *temp_port;
```

```
    int             end = 0;

    temp_port = ports;
    do
    {
        if (strstr(inbuf, temp_port->name) != NULL) return(1);
        if (temp_port->next != NULL) temp_port = temp_port->next;
        else end = 1;
    } while (!end);

    return(0);
}

/*****************************************************************************
 *
 * Function     : read_line
 *
 * Description  :
 *          This function scans an entire action line of a process. First it adds a transition to OUT_FILE, then it
 *          checks whether a choicestate has to be inserted and calls do_choice() if this is the case. Next it
 *          ignores possible (*..*) comment, looks which variables are used in possible predicates and assigns
 *          p-use to these variables. Then it calls write_trans() to write a transition. After this it checks
 *          which action is used in this line and calls write_state() to write this actionstate. Then it scans
 *          to which flowstate the action leads and writes the transition and if necessary this flowstate. Also it
 *          calls scan_pars to check if variables are used while calling a flowstate and writing these variables
 *          to TRANS_FILE_1 if this is the case. At the end it writes the outgoing transition to OUT_FILE and resets
 *          the chain of variables for the next actionline to be read.
 *
 * Arguments    :
 *          files_type  *files      : pointer to files
 *          efg_type    *efg        : pointer to efg
 *          int             flowstate   : number of flowstate as in StateX
 *          var_type    *statevars  : chain of used and defined variables
 *          int             intern_num  : number of flowstate as in @ST <num>
 *
 * Returns      : void
 *
 *****************************************************************************/

static void
read_line(files, efg, flowstate, statevars, intern_num)
files_type  *files;
efg_type    *efg;
int             flowstate;
var_type    *statevars;
int             intern_num;
{
    int             choice = 0,
                    pred = 0,
                    end = 0,
                    star = 0,
                    last_arrow = 0;
    char        inbuf[400],
                inbuf2[400],
                act[80],
                int_act[80],
                in_var[80];
    var_type    *temp_var,
                *old_last;
    char        c;
    int             i, j;

    printf("***");                                  /* shows number of scanned lines */
    fflush(stdout);

    fprintf(files->outfp, "%d %d\n", flowstate, efg->num_trans+1);              /* add outgoing transition */

    temp_var = statevars;
    do
    {
        old_last = temp_var;                                        /* old_last indicates end of original varchain. */
        if (old_last->next == NULL) end = 1;                        /* Is to be used to restore old varchain before */
        else temp_var = temp_var->next;                             /* scanning a new line.                         */
    } while (!end);

    fgets(inbuf, 399, files->smilefp);                                                  /* get line */
    if (feof(files->smilefp)) err(m_efg, 16, flowstate);

    if ((strstr(inbuf, "(choice ") != NULL) && (strstr(inbuf, "[]") != NULL))           /* line is choice line */
    {
        do_choice(files, efg, inbuf, statevars, intern_num);
        choice = 1;
        fgets(inbuf, 399, files->smilefp);                                              /* get new line */
        if (feof(files->smilefp)) err(m_efg, 16, flowstate);
    }

    for (i = 0; inbuf[i] == ' '; i++);
    strcpy(inbuf, &inbuf[i]);                                                           /* remove possible white */

    if ((strstr(inbuf, "(**") != NULL) && (strcmp(inbuf, "**)") != NULL))               /* line starts with (*..*) */
    {
        for (i = 0; (inbuf[i] != '*'); i++);
        strcpy(int_act, &inbuf[i+1]);                                                   /* get intern event action  */

        for (i = 0; (int_act[i] != '*'); i++);
```

```
    int_act[i] = '\0';                                                  /* remove rest action */

    for (i = 0; star < 2; i++)
    {
        if (inbuf[i] == '*') star++;
    }
    strcpy(inbuf, &inbuf[i+2]);                                         /* remove (*..*) */
}

for (i = 0; inbuf[i] != '\0'; i++)                                      /* check if predicates */
{
    if ((inbuf[i] == ']') && (inbuf[i+1] == ' ') && (inbuf[i+2] == '-') && (inbuf[i+3] == '>'))
        last_arrow = i+3;
}
for (i = 0; i != last_arrow; i++) inbuf2[i] = inbuf[i];
inbuf2[i] = '\0';                                                       /* fill inbuf2 with predicates */
if (last_arrow != 0) strcpy(inbuf, &inbuf[last_arrow+2]);               /* remove predicates from inbuf */

temp_var = statevars;
while (strcmp(temp_var->name, "") != 0)
{
    if (strstr(inbuf2, temp_var->name) != NULL)                 /* check which variables are used in predicates */
    {
        temp_var->use = p_use;
        pred = 1;
    }
    temp_var = temp_var->next;
}

write_trans(choice, pred, intern_num, files, efg, statevars);

/* check action */
sscanf(inbuf, "%s", inbuf2);
if (strcmp(inbuf2, "i") == 0)                                                       /* intern */
{
    write_state(files->state1fp, intern, 0, int_act, statevars, efg->num_trans, efg->num_trans + 1, efg);
}
else if ((port_cmp(efg->ports, inbuf2) == 1) && (strstr(inbuf2, ";") != NULL))      /* pure output */
{
    temp_var = init_var();
    for (i = 0; inbuf[i] != '\0'; i++)                                              /* remove ';' */
        if (inbuf[i] == ';') inbuf[i] = '\0';
    write_state(files->state1fp, output, 0, inbuf, temp_var, efg->num_trans, efg->num_trans + 1, efg);
    free(temp_var);
}
else if ((port_cmp(efg->ports, inbuf2) == 1) && (strstr(inbuf, "!") != NULL))       /* output */
{
    temp_var = statevars;
    while (strcmp(temp_var->name, "") != 0)                                 /* check which variables     */
    {                                                                       /* are used in output action */
        if (strstr(inbuf, temp_var->name) != NULL) temp_var->use = c_use;
        temp_var = temp_var->next;
    }
    for (i = 0; inbuf[i] != '\0'; i++)                                              /* remove ';' */
        if (inbuf[i] == ';') inbuf[i] = '\0';
    write_state(files->state1fp, output, 0, inbuf, statevars, efg->num_trans, efg->num_trans + 1, efg);
}
else if ((port_cmp(efg->ports, inbuf2) == 1) && (strstr(inbuf, "?") != NULL))       /* input */
{
    temp_var = statevars;
    while (strcmp(temp_var->name, "") != 0) temp_var = temp_var->next;
    if (strstr(inbuf, ":") != NULL)                                         /* input variable */
    {
        for (i = 0; inbuf[i] != '?'; i++);
        i++;
        for (j = 0; inbuf[i] != ':'; j++)
        {
            in_var[j] = inbuf[i++];                                         /* get input variable */
        }
        in_var[j] = '\0';
        strcpy(temp_var->name, in_var);                                     /* add inputvar aan varchain */
        temp_var->next = init_var();
    }
    for (i = 0; inbuf[i] != '\0'; i++)                                              /* remove ';' */
        if (inbuf[i] == ';') inbuf[i] = '\0';
    write_state(files->state1fp, input, 0, inbuf, temp_var, efg->num_trans, efg->num_trans + 1, efg);
}
else err(m_efg, 17, flowstate);                                                     /* fault */

scan_state(files, &flowstate);
if (!exist_flow(files->state1fp, flowstate, &intern_num)) intern_num = efg->num_states + 1;

write_basic_trans(files->trans1fp, efg, efg->num_states, intern_num);
while (((c = fgetc(files->smilefp)) != '\n') && (c != '('));
if (c == '(')                                                   /* parameters are given to State */
    scan_pars(files->smilefp, files->trans1fp, statevars);
else
    fprintf(files->trans1fp, "\n\n");

if (!exist_flow(files->state1fp, flowstate, &intern_num))               /* write flowstate if not exists */
{
    temp_var = init_var();
    get_vars(files->var1fp, flowstate, temp_var);
    write_state(files->state1fp, flow, flowstate, act, temp_var, 0, 0, efg);
    free_var_chain(temp_var);
```

```
    }
    fprintf(files->infp, "%d %d\n", flowstate, efg->num_trans);                /* add incoming transition */

    if (old_last->next != NULL)
    {
        free_var_chain(old_last->next);                          /* remove possible line-dependent tail of varchain */
        old_last->next = NULL;
    }
    strcpy(old_last->name, "");

    end = 0;
    temp_var = statevars;
    do
    {
        temp_var->use = def;                                     /* reset all uses of varchain to def */
        if (temp_var->next == NULL) end = 1;
        else temp_var = temp_var->next;
    } while (!end);
}

/*****************************************************************************
 *
 * Function     : read_proc
 *
 * Description  :
 *          This function scans a process by scanning 'PROCESS', calling scan_state() for scanning the number of
 *          the flowstate, calling get_vars() to get defined variables in this process, writing the flowstate if
 *          it doesn't exist yet, calling read_line() as long as new actionlines appear and finally scanning
 *          'ENDPROC'. if the starting PROCESS cannot be found, this function calls update_inout() and status_efg()
 *          to transform STATE_FILE_1 to STATE_FILE_2 and to print some statistical information about the efg.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to efg
 *          files_type  *files  : pointer to files
 *
 * Returns      : int
 *          0: PROCESS scanned
 *          1: End of Smile file reached
 *
 *****************************************************************************/

static  int
read_proc(efg, files)
efg_type    *efg;
files_type  *files;
{
    int             flowstate,
                intern_num;
    char        act[80];
    var_type    *statevars;
    char        inbuf[400],
                s[80];

    do
    {
        if (fscanf(files->smilefp, "%s", s) == EOF)                /* no next process, return with 1 */
        {
            update_inout(files);
            status_efg(efg);
            return(1);
        }
    } while (strcmp(s, "PROCESS") != 0);

    scan_state(files, &flowstate);                                             /* scan flowstate */
    printf("\nScanning process %d: ", flowstate);

    fgets(inbuf, 399, files->smilefp);
    if (feof(files->smilefp)) err(m_efg, 18, flowstate);

    strcpy(act, "");
    statevars = init_var();
    get_vars(files->var1fp, flowstate, statevars);

    if (!exist_flow(files->state1fp, flowstate, &intern_num))                  /* write flowstate if not exists */
    {
        write_state(files->state1fp, flow, flowstate, act, statevars, 0, 0, efg);
    }

    do                                                                         /* scan all lines */
    {
        read_line(files, efg, flowstate, statevars, intern_num);
        fscanf(files->smilefp, "%s", s);
    } while (strcmp(s, "[]") == 0);

    free_var_chain(statevars);

    if (strcmp(s, "ENDPROC") != 0) err(m_efg, 19, 0);
    else
    {
        printf(" Ready");
        return(0);
    }
}

/*****************************************************************************
```

```
 *
 * Function      : make_efg
 *
 * Description   :
 *          This function is the main function of this module; it sequently calls init_efg(), open_files_efg(),
 *          read_statevars(), read_spec(), read_behav(), read_proc() as much as necessary and close_files_efg().
 *
 * Arguments     :
 *          efg_type    *efg     : pointer to efg
 *          files_type  *files   : pointer to files
 *
 * Returns       : void
 *
 ******************************************************************************/

extern  void
make_efg(efg, files)
efg_type    *efg;
files_type  *files;
{
    int     end = 0;

    init_efg(efg);

    if (open_files_efg(files) == 1) return;

    calc_vars(files);

    read_statevars(files);

    read_spec(efg, files);

    read_behav(efg, files);

    do
    {
        end =   read_proc(efg, files);
    } while (end == 0);                                     /* end = 1: EOF reached; OK */

    close_files_efg(files);
}
```

```
/*****************************************************************************
 *
 * File             : makeuio.c
 *
 * Description  :
 *          This module calculates for each basic flow state if possible one or more UIO sequences and writes these
 *          sequences to the UIO_FILE. In order to derive this file other intermediate files are created and used.
 *          The calculation of the UIO sequences consists of the following steps:
 *          First calculate all basic sequences from all basic flow states (state from which input action takes place)
 *          to next basic flow states. These sequences are checked for not including variables in the IO sequence
 *          because this will complicate making statements about uniqueness of a sequence containing such variables.
 *          The basic sequences not containing variables will be checked for uniqueness (no other sequences having
 *          the same input/output sequence), half-uniqueness (other equal IO sequences exist but they lead to a
 *          different endstate) and non-uniqueness (other equal IO sequences to same endstate). If for a basic
 *          flowstate an UIO sequence is found this state is marked and in a next step no new sequences will be
 *          calculated. If for a certain flowstate no UIO sequence is found, all half unique sequences of the last
 *          step starting with this flowstate will be expanded with all possible basic sequences starting with
 *          the same basic flowstate as with which the half unique sequence ends. This algorithm continues until
 *          for each basic flowstate an UIO sequence is found or a certain limit is reached.
 *          The files used in this module are:
 *          - VAR_FILE_2: This file, constructed in the module makeefg, is used for checking if variables exist in
 *          a basic sequence.
 *          - BAS_SEQ_FILE: This file contains all calculated basic sequences, each on a separate line. Such a
 *          sequence consists of all states traversed, for example 'F1 I3 F4 O2 X6 F5\n'.
 *          - BAS_ACT_FILE: This file consists of all input/output sequences matching to the state sequences of
 *          BAS_SEQ_FILE, each on the same line as the matching state sequence. Example: 'L?SABM L!DM \n'.
 *          - BAS_UNIQ_FILE: This file contains for each matching input/output sequence of BAS_ACT_FILE a '0' if
 *          this sequence contains no variables, and a '3' if it contains variables.
 *          - TOT_SEQ_FILE: This file contains all calculated state sequences in the different steps in a same
 *          structure as BAS_SEQ_FILE.
 *          - TOT_ACT_FILE: This file contains all input/output sequences matching to the state sequences of
 *          TOT_SEQ_FILE.
 *          - TOT_UNIQ_FILE: This file contains for each sequence in the TOT-files a '0' if the sequence is
 *          non-unique, a '1' if the sequence is unique and a '2' if the sequence is half-unique.
 *          - UIO_FILE: This file is the main file of this module. It contains all calculated UIO sequences for
 *          all basic flow states. For each basic flow state, it has the following structure:
 *              '_____________________________'
 *              'Fx'                                        x is number of flowstate
 *              For each UIO sequence of this flow state:
 *              blank line
 *              state sequence as in TOT_SEQ_FILE           'NO UIO' if no UIO sequence found
 *              IO sequence as in TOT_ACT_FILE              'NO UIO' if no UIO sequence found
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : open_files_uio
 *
 * Description  :
 *          This function tries to open the necessary files in this module.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
open_files_uio(files)
files_type  *files;
{
    if ((files->state2fp = fopen(STATE_FILE_2, "r")) == NULL)
        err(m_uio, 6, 0);
    if ((files->trans1fp = fopen(TRANS_FILE_1, "r")) == NULL)
        err(m_uio, 2, 0);
    if ((files->var1fp = fopen(VAR_FILE_1, "r")) == NULL)
        err(m_uio, 1, 0);
    if ((files->var2fp = fopen(VAR_FILE_2, "r")) == NULL)
        err(m_uio, 20, 0);
    if ((files->basseqfp = fopen(BAS_SEQ_FILE, "w+")) == NULL)
        err(m_uio, 21, 0);
    if ((files->basactfp = fopen(BAS_ACT_FILE, "w+")) == NULL)
        err(m_uio, 22, 0);
    if ((files->basuniqfp = fopen(BAS_UNIQ_FILE, "w+")) == NULL)
        err(m_uio, 23, 0);
    if ((files->totseqfp = fopen(TOT_SEQ_FILE, "w+")) == NULL)
        err(m_uio, 24, 0);
    if ((files->totactfp = fopen(TOT_ACT_FILE, "w+")) == NULL)
        err(m_uio, 25, 0);
    if ((files->totuniqfp = fopen(TOT_UNIQ_FILE, "w+")) == NULL)
        err(m_uio, 26, 0);
```

```
    if ((files->uiofp = fopen(UIO_FILE, "w+")) == NULL)
        err(m_uio, 27, 0);
}

/*****************************************************************************
 *
 * Function     : close_files_uio
 *
 * Description  :
 *          This function closes all files used in this module.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
close_files_uio(files)
files_type  *files;
{
    fclose(files->state2fp);
    fclose(files->trans1fp);
    fclose(files->var1fp);
    fclose(files->var2fp);
    fclose(files->basseqfp);
    fclose(files->basactfp);
    fclose(files->basuniqfp);
    fclose(files->totseqfp);
    fclose(files->totactfp);
    fclose(files->totuniqfp);
    fclose(files->uiofp);
}

/*****************************************************************************
 *
 * Function     : find_line
 *
 * Description  :
 *          This function tries to find a line in the file with filepointer fp.
 *
 * Arguments    :
 *          char    line[80]    : line to search for
 *          FILE    *fp         : file in which to search for line
 *
 * Returns      : void
 *
 *****************************************************************************/

extern  void
find_line(line, fp)
char    line[80];
FILE    *fp;
{
    char    inbuf[80];

    do
    {
        fgets(inbuf, 79, fp);
    } while ((!feof(fp)) && (strcmp(inbuf, line) != 0));
    if (feof(fp))
    {
        printf("\n%s not found\n", line);
        err(m_uio, 28, 0);
    }
}

/*****************************************************************************
 *
 * Function     : find_st_line_next_state
 *
 * Description  :
 *          This function will find in TRANS_FILE_1 the transition with the number in parameter word. Next it will
 *          find the tostate of the transition and go to after the "@ST"-line of this tostate in STATE_FILE_2.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          char        *word   : string consisting of transition number
 *
 * Returns      : int   : internal number of next flowstate
 *
 *****************************************************************************/

extern  int
find_st_line_next_state(files, word)
files_type  *files;
char        *word;
{
    int         trans,
                tostate;
    char    inbuf[80],
            line[80],
            word2[50];
```

```
    trans = atoi(word);

    rewind(files->trans1fp);
    sprintf(line, "@TR %d\n", trans);
    find_line(line, files->trans1fp);                                   /* find transition */

    fgets(inbuf, 79, files->trans1fp);
    fgets(inbuf, 79, files->trans1fp);
    if (sscanf(inbuf, "%s%s", word, word2) == EOF) err(m_uio, 29, 0);
    tostate = atoi(word2);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", tostate);
    find_line(line, files->state2fp);                                   /* find tostate of transition */

    return(tostate);
}

/*****************************************************************************
 *
 * Function     : add_next_state
 *
 * Description  :
 *          This function will add the next state, which ought to be in the next line of STATE_FILE_2, to seq
 *          and will return the first word of this line (e.g. F or I or X or O or C).
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          char        *seq    : sequence which will be updated
 *
 * Returns      : char *    : first word of first encountered line in STATE_FILE_2
 *
 *****************************************************************************/

extern  char *
add_next_state(files, seq)
files_type  *files;
char        *seq;
{
    char    inbuf[80],
            word1[40],
            word2[40];
    int         i,
            seqlen;

    fgets(inbuf, 79, files->state2fp);
    if (sscanf(inbuf, "%s%s", word1, word2) == EOF) err(m_uio, 30, 0);
    strcat(seq, " ");
    for(i = 0; inbuf[i] != '\0'; i++)
    {
        if ((inbuf[i] != ' ') && (inbuf[i] != '\n'))
        {
            seqlen = strlen(seq);
            seq[seqlen] = inbuf[i];                                     /* add next state to sequence */
            seq[seqlen+1] = '\0';
        }
    }
    return(word1);
}

/*****************************************************************************
 *
 * Function     : add_next_state_begin
 *
 * Description  :
 *          This function will add the next state, which ought to be in the next line of STATE_FILE_2, to the
 *          beginning of seq and will return the first word of this line (e.g. F or I or X or O or C).
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          char        *seq    : sequence which will be updated
 *
 * Returns      : char *    : first word of first encountered line in STATE_FILE_2
 *
 *****************************************************************************/

extern  char *
add_next_state_begin(files, seq)
files_type  *files;
char        *seq;
{
    char    inbuf[80],
            *buf2,
            word1[40],
            word2[40];
    int         i,
            seqlen;

    buf2 = (char *) malloc((strlen(seq) + 120) * sizeof(char));

    fgets(inbuf, 79, files->state2fp);
    if (sscanf(inbuf, "%s%s", word1, word2) == EOF) err(m_uio, 30, 0);
    strcpy(buf2, "");
    for (i = 0; inbuf[i] != '\0'; i++)
    {
```

```
        if ((inbuf[i] != ' ') && (inbuf[i] != '\n'))
        {
            seqlen = strlen(buf2);
            buf2[seqlen] = inbuf[i];
            buf2[seqlen+1] = '\0';
        }
    }
    strcat(buf2, " ");                  /* expand seq with word and a space at beginning of seq */
    strcat(buf2, seq);
    strcpy(seq, buf2);

    return(word1);
}

/*****************************************************************************
 *
 * Function     : is_basic
 *
 * Description  :
 *          This function checks whether a flowstate is basic or not by looking if there is an inputaction from
 *          the flowstate.
 *
 * Arguments    :
 *          int             state       : internal number of flowstate to check
 *          files_type  *files      : pointer to files
 *
 * Returns      : int
 *          0: flowstate is not basic
 *          1: flowstate is basic
 *
 *****************************************************************************/

extern  int
is_basic(state, files)
int             state;
files_type  *files;
{
    char    line[80];
    char    word1[40];
    long    pos,
            pos_state;
    int         tostate;

    pos_state = ftell(files->state2fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);                                   /* find '@ST state' */

    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);                                   /* find '@OUT' */

    if (fscanf(files->state2fp, "%s", word1) == EOF) strcpy(word1, "@ST");          /* in case state is last state */
    pos = ftell(files->state2fp);                                                   /* in file                    */
    while (strcmp(word1, "@ST") != 0)                                   /* no next state reached */
    {
        tostate = find_st_line_next_state(files, word1);

        if (fscanf(files->state2fp, "%s", word1) == EOF) err(m_uio, 30, 0);
        if (strcmp(word1, "I") == 0)
        {
            fseek(files->state2fp, pos_state, 0);
            return(1);                          /* basic if at least one input from state */
        }
        fseek(files->state2fp, pos, 0);
        if (fscanf(files->state2fp, "%s", word1) == EOF) strcpy(word1, "@ST");
        pos = ftell(files->state2fp);
    }
    fseek(files->state2fp, pos_state, 0);
    return(0);                                                          /* not basic if no inputs from state */
}

/*****************************************************************************
 *
 * Function     : is_stable
 *
 * Description  :
 *          This functions checks if a flowstate is stable by looking if there are output, internal events or
 *          choice states leaving from the flowstate.
 *
 * Arguments    :
 *          int             state       : internal number of flowstate to check
 *          files_type  *files      : pointer to files
 *
 * Returns      : int
 *          0: flowstate is not stable
 *          1: flowstate is stable
 *
 *****************************************************************************/

extern  int
is_stable(state, files)
int             state;
files_type  *files;
{
    char    line[80];
```

```
    char    word1[40];
    long    pos,
            pos_state;
    int         tostate;

    pos_state = ftell(files->state2fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);;                                  /* find '@ST state' */

    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);;                                  /* find '@OUT' */

    if (fscanf(files->state2fp, "%s", word1) == EOF) strcpy(word1, "@ST");      /* in case state is last state */
    pos = ftell(files->state2fp);                                       /* in file                    */
    while (strcmp(word1, "@ST") != 0)                                   /* no next state reached */
    {
        tostate = find_st_line_next_state(files, word1);

        if (fscanf(files->state2fp, "%s", word1) == EOF) err(m_uio, 30, 0);
        if ((strcmp(word1, "X") == 0) || (strcmp(word1, "O") == 0) || (strcmp(word1, "C") == 0))
        {
            fseek(files->state2fp, pos_state, 0);
            return(0);                              /* not stable if output, intern event or choice from state */
        }
        fseek(files->state2fp, pos, 0);
        if (fscanf(files->state2fp, "%s", word1) == EOF) strcpy(word1, "@ST");
        pos = ftell(files->state2fp);
    }
    fseek(files->state2fp, pos_state, 0);
    return(1);                                  /* stable if only inputs from state */
}

/*****************************************************************************
 *
 * Function     : statenumber
 *
 * Description  :
 *          This function calculates the internal number of a flowstate from the number given as in PROCESS StateX.
 *
 * Arguments    :
 *          int             flowstate   : flowstatenumber as in PROCESS
 *          files_type  *files      : pointer to files
 *
 * Returns      : int   : statenumber as in EFG
 *
 *****************************************************************************/

extern  int
statenumber(flowstate, files)
int             flowstate;
files_type  *files;
{
    long    pos,
            pos_old,
            pos_new;
    char    line[80],
            inbuf[80],
            inbuf_old[80],
            word1[40],
            word2[40];

    pos = ftell(files->state2fp);

    sprintf(line, "F %d\n", flowstate);

    rewind(files->state2fp);
    strcpy(inbuf, "");
    do
    {
        strcpy(inbuf_old, inbuf);
        fgets(inbuf, 79, files->state2fp);
    } while ((!feof(files->state2fp)) && (strcmp(inbuf, line) != 0));           /* find 'F flowstate' */
    if (feof(files->state2fp))
    {
        printf("\n%s not found\n", line);
        err(m_uio, 28, 0);
    }

    sscanf(inbuf_old, "%s%s", word1, word2);                            /* scan internal number flowstate */
    fseek(files->state2fp, pos, 0);
    return(atoi(word2));
}

/*****************************************************************************
 *
 * Function     : flownumber
 *
 * Description  :
 *          This function calculates the number given as in PROCESS StateX from the internal number of a flowstate.
 *
 * Arguments    :
 *          int             state   : statenumber as in EFG
 *          files_type  *files  : pointer to files
 *
```

```
 * Returns      : int   : flowstatenumber as in PROCESS
 *
 ******************************************************************************/

extern  int
flownumber(state, files)
int             state;
files_type  *files;
{
    long    pos;
    char    line[80],
            inbuf[80],
            word1[40],
            word2[40];

    pos = ftell(files->state2fp);
    rewind(files->state2fp);

    sprintf(line, "@ST %d\n", state);
    do
    {
        fgets(inbuf, 79, files->state2fp);
    } while ((!feof(files->state2fp)) && (strcmp(inbuf, line) != 0));              /* find '@ST state' */
    if (feof(files->state2fp))
    {
        printf("\n%s not found\n", line);
        err(m_uio, 28, 0);
    }

    fscanf(files->state2fp, "%s%s", word1, word2);                          /* scan number flowstate as in PROCESS */
    fseek(files->state2fp, pos, 0);
    return(atoi(word2));
}

/******************************************************************************
 *
 * Function     : step_state
 *
 * Description  :
 *          This function adds the next state to a sequence and, depending of this next state does the following:
 *          If the state is a basic flow state: write sequence in basic sequence file.
 *          Else: If the state is a visited flowstate: ignore the sequence.
 *          Else: call step_state for a next step.
 *
 * Arguments    :
 *          char        *seq        : pointer to string containing sequence
 *          int             *basic      : pointer to structure giving which flowstates are basic
 *                      *visited    : pointer to structure giving which flowstates are already visited
 *          files_type  *files      : pointer to files
 *          efg_type    *efg        : pointer to EFG
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
step_state(seq, basic, visited, files, efg)
char        *seq;
int             *basic,
            *visited;
files_type  *files;
efg_type    *efg;
{
    char    word[40];
    long    pos;
    char    *seq2;
    int         tostate,
            toflow,
            i;
    int         *visited2;
    char    line[80];

    seq2 = (char *) malloc(sizeof(char) * 200);
    visited2 = (int *) malloc(sizeof(int) * (efg->num_flow+1));

    if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");            /* state is last state in file */
    pos = ftell(files->state2fp);
    while (strcmp(word, "@ST") != 0)                                                /* no next state */
    {
        tostate = find_st_line_next_state(files, word);
        toflow = flownumber(tostate, files);

        strcpy(seq2, seq);
        strcpy(word, add_next_state(files, seq2));

        if ((basic[toflow] == 1) && (strcmp(word, "F") == 0))           /* if basic flowstate reached, put sequence */
            fprintf(files->basseqfp, "%s\n", seq2);                         /* in basic sequence file
*/
        else
        {
            if ((visited[toflow]) && (strcmp(word, "F") == 0)) ;        /* else, if flowstate already visted: */
                                                                        /* do nothing, skip this sequence     */
            else
            {
                sprintf(line, "@OUT\n");                                /* else (flowstate not visited):  */
```

```
                find_line(line, files->state2fp);                    /* do step_state for next state   */
                for (i=1; i!=efg->num_flow+1; i++) visited2[i] = visited[i];
                step_state(seq2, basic, visited2, files, efg);
            )
        )
        fseek(files->state2fp, pos, 0);
        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
        pos = ftell(files->state2fp);
    }
    free(seq2);
    free(visited2);
}

/*****************************************************************************
 *
 * Function     : get_basic_seq
 *
 * Description  :
 *          Function calling step_state for each basic flowstate after initializing this basic sequence by the
 *          departing flowstate.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          efg_type    *efg    : pointer to EFG
 *          int             *basic  : pointer to structure giving which flowstates are basic
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
get_basic_seq(files, efg, basic)
files_type  *files;
efg_type    *efg;
int             *basic;
{
    int         *visited;
    int         flowstate,
            state,
            seqlen,
            i;
    char    *seq;
    char    line[80],
            inbuf[80];

    visited = (int *) malloc(sizeof(int) * (efg->num_flow+1));

    efg->num_basflow = 0;
    for (flowstate=1; flowstate!=efg->num_flow+1; flowstate++)
    (
        if ((basic[flowstate] = is_basic(statenumber(flowstate, files), files)) == 1)
            efg->num_basflow++;                                                          /* init basic[] */
        visited[flowstate] = 0;                                                      /* init visited[] */
    )
    seq = (char *) malloc(sizeof(char) * 200);
    strcpy(seq, "");

    printf("\nCalculating basic sequences for flowstate    ");
    for (flowstate=1; flowstate!=efg->num_flow+1; flowstate++)
    (
        if (basic[flowstate] == 1)                                       /* for each basic flowstate */
        (
            printf("\b\b\b%-3d", flowstate);
            fflush(stdout);

            state = statenumber(flowstate, files);
            visited[flowstate] = 1;                                          /* flowstate is visited */

            rewind(files->state2fp);
            sprintf(line, "@ST %d\n", state);
            find_line(line, files->state2fp);                            /* find '@ST state' */

            fgets(inbuf, 79, files->state2fp);
            strcpy(seq, "");
            for(i = 0; inbuf[i] != '\0'; i++)
            {
                if ((inbuf[i] != ' ') && (inbuf[i] != '\n'))
                {
                    seqlen = strlen(seq);
                    seq[seqlen] = inbuf[i];                              /* init sequence with flowstate */
                    seq[seqlen+1] = '\0';
                }
            )

            sprintf(line, "@OUT\n");
            find_line(line, files->state2fp);                            /* find '@OUT' */

            step_state(seq, basic, visited, files, efg);                 /* call step_state */
        }
    }
    printf("\n");
    free(seq);
    free(visited);
}
```

```
/*****************************************************************************
 *
 * Function     : get_basic_act
 *
 * Description  :
 *          This function makes the basic action file out of the basic sequence file by traversing each line of
 *          the sequence file and writing the action of a input or output states encountered in this line into
 *          the basic action file. From these actions type parts (:..)  and unnecessary spaces are removed.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          efg_type    *efg    : pointer to EFG
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
get_basic_act(files, efg)
files_type  *files;
efg_type    *efg;
{
    int         i,
                j,
                c,
                action;
    char        word[10],
                line[10],
                inbuf[400],
                inbuf2[400];

    rewind(files->basseqfp);
    printf("\nCalculating actions of basic sequences\n");

    do
    {
        action = 0;
        strcpy(word, "");
        c = fgetc(files->basseqfp);
        for (i = 0; (c != EOF) && (c != ' ') && (c != '\n'); i++)
        {
            word[i] = c;                                        /* get statetype and number */
            c = fgetc(files->basseqfp);
        }
        word[i] = '\0';

        if ((word[0] == 'I') || (word[0] == 'O'))               /* if statetype is input or output */
        {
            action = 1;
            line[0] = word[0];
            line[1] = ' ';
            line[2] = '\0';
            strcpy(word, &word[1]);
            strcat(line, word);
            strcat(line, "\n");
            rewind(files->state2fp);
            find_line(line, files->state2fp);                   /* find state */

            fgets(inbuf, 399, files->state2fp);                     /* get '@ACT ...' */
            strcpy(inbuf, &inbuf[5]);                           /* remove '@ACT ' */
            j = 0;
            for (i = 0; inbuf[i] != '\n'; i++)
            {
                if (inbuf[i] != ' ')                            /* remove spaces */
                {
                    if (inbuf[i] == ':')
                        inbuf2[j++] = '\0';                         /* remove variable type parts (: type) */
                    else
                        inbuf2[j++] = inbuf[i];
                }
            }
            inbuf2[j] = '\0';

            fprintf(files->basactfp, "%s", inbuf2);
        }

        if ((c == '\n') || ((c == ' ') && (action == 1))) fputc(c, files->basactfp);

    } while (c != EOF);
}

/*****************************************************************************
 *
 * Function     : init_uio_found
 *
 * Description  :
 *          This function initiates the structure uio_found with all numbers of basic flowstates and sets the variable
 *          found for each state to 0.
 *
 * Arguments    :
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          efg_type        *efg        : pointer to EFG
 *          int                 *basic      : pointer to structure giving which flowstates are basic
 *
```

```
 * Returns      : void
 *
 ******************************************************************************/

static  void
init_uio_found(uio_found, efg, basic)
uio_found_type  *uio_found;
efg_type        *efg;
int                 *basic;
{
    int     flowstate,
        i = 0;

    for (flowstate=1; flowstate!=efg->num_flow+1; flowstate++)
    {
        if (basic[flowstate] == 1)
        {
            (&uio_found[i])->fstate = flowstate;
            (&uio_found[i])->found = 0;
            i++;
        }
    }
}

/******************************************************************************
 *
 * Function     : init_tot_files
 *
 * Description  :
 *          Function making total sequence and action files by looking in the basic uniqueness file which actions
 *          contain no variables and putting these actions and sequences into the total files.
 *
 * Arguments    :
 *          files_type      *files  : pointer to files
 *          efg_type        *efg    : pointer to EFG
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
init_tot_files(files, efg)
files_type      *files;
efg_type        *efg;
{
    char    *inact,
            *inseq;
    int         c,
            uiofnd;

    inact = (char *) malloc(sizeof(char) * 200);
    inseq = (char *) malloc(sizeof(char) * 200);

    rewind(files->basseqfp);
    rewind(files->basactfp);
    rewind(files->basuniqfp);
    c = getc(files->basseqfp);
    while (c != EOF)
    {
        ungetc(c, files->basseqfp);
        fgets(inseq, 199, files->basseqfp);
        fgets(inact, 199, files->basactfp);
        fscanf(files->basuniqfp, "%d", &uiofnd);
        if (uiofnd != 3)                                    /* if action consists no variables: */
        {
            fputs(inseq, files->totseqfp);                  /* copy sequence to total sequence file */
            fputs(inact, files->totactfp);                  /* copy action to total action file */
        }
        c = getc(files->basseqfp);
    }
    clearerr(files->basseqfp);
    clearerr(files->basactfp);

    free(inact);
    free(inseq);
}

/******************************************************************************
 *
 * Function     : all_uios_found
 *
 * Description  :
 *          This function checks if there is a basic flowstate for which uio_found->found iequals zero.
 *
 * Arguments    :
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          efg_type        *efg        : pointer to EFG
 *
 * Returns      : int
 *          0: not all uio sequences found
 *          1: all uio sequences found
 *
 ******************************************************************************/
```

```
static  int
all_uios_found(uio_found, efg)
uio_found_type  *uio_found;
efg_type        *efg;
{
    int     flowstate;

    for (flowstate = 0; flowstate != efg->num_basflow; flowstate++)
        if ((&uio_found[flowstate])->found == 0) return(0);

    return(1);
}

/*****************************************************************************
 *
 * Function     : contains_var
 *
 * Description  :
 *          This function checks if inbuf contains one of the variables written in VAR_FILE_2.
 *
 * Arguments    :
 *          char        *inbuf  : pointer to string to check
 *          files_type  *files  : pointer to files
 *
 * Returns      : int
 *          0: inbuf contains no variables
 *          1: inbuf contains variables
 *
 *****************************************************************************/

static  int
contains_var(inbuf, files)
char        *inbuf;
files_type  *files;
{
    char    word[40];

    rewind(files->var2fp);

    while (!feof(files->var2fp))
    {
        fscanf(files->var2fp, "%s", word);
        if (strstr(inbuf, word) != NULL) return(1);
    }

    return(0);
}

/*****************************************************************************
 *
 * Function     : check_vars
 *
 * Description  :
 *          This function checks all lines in the basic action file if it contains a variable from VAR_FILE_2.
 *          If so, it writes '3\n' in the basic uniqueness file for this line. Otherwise, it writes '0\n'.
 *          Also, it initiates uio_found with 0's or with a 3 if there are no actions without variables for a
 *          given flowstate.
 *
 * Arguments    :
 *          files_type      *files      : pointer to files
 *          efg_type        *efg        : pointer to EFG
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
check_vars(files, efg, uio_found)
files_type      *files;
efg_type        *efg;
uio_found_type  *uio_found;
{
    char    *inact,
            *inseq,
            word[40];
    int         c,
            flowstate,
            end;

    printf("\nChecking variable existence in basic sequences\n");
    fflush(stdout);

    inact = (char *) malloc(sizeof(char) * 200);
    inseq = (char *) malloc(sizeof(char) * 200);

    rewind(files->basseqfp);
    rewind(files->basactfp);
    rewind(files->basuniqfp);

    c = getc(files->basactfp);
    while (!feof(files->basactfp))                          /* for each action line */
    {
        ungetc(c, files->basactfp);
```

```
        fgets(inact, 199, files->basactfp);
        if (contains_var(inact, files))                          /* if action contains variables */
        {
            fprintf(files->basuniqfp, "3\n");
        }
        else                                                     /* if action contains no variables */
        {
            fprintf(files->basuniqfp, "0\n");
        }
        c = getc(files->basactfp);
    }
    clearerr(files->basactfp);

    for (flowstate = 0; flowstate != efg->num_basflow; flowstate++)
        (&uio_found[flowstate])->found = 3;                      /* init uio_found with 3's */

    rewind(files->basseqfp);
    rewind(files->basuniqfp);

    c = getc(files->basuniqfp);
    while (!feof(files->basuniqfp))                                  /* for each basic sequence */
    {
        fgets(inseq, 199, files->basseqfp);
        if (c == '0')                                            /* if action contains no variables */
        {
            sscanf(inseq, "%s", word);
            end = 0;
            for (flowstate = 0; ((flowstate != efg->num_basflow) && (end == 0)); flowstate++)
            {
                if (atoi(&word[1]) == (&uio_found[flowstate])->fstate)
                {
                    (&uio_found[flowstate])->found = 0;          /* set uio_found of state that begins in */
                    end = 1;                                     /* sequence to 0                         */
                }
            }
        }
        c = getc(files->basuniqfp);
        c = getc(files->basuniqfp);
    }
    clearerr(files->basuniqfp);

    rewind(files->basseqfp);
    rewind(files->basactfp);
    rewind(files->basuniqfp);

    free(inact);
    free(inseq);
}

/*****************************************************************************
 *
 * Function     : get_maxstep
 *
 * Description  :
 *          This function asks for the maximal number of steps for calculating UIO sequences. Default is n*n, with
 *          n equaling the number of basic flowstates.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to EFG
 *
 * Returns      : int   : number of steps
 *
 *****************************************************************************/

static  int
get_maxstep(efg)
efg_type    *efg;
{
    int         steps;
    char    inbuf[200];

    printf("\nGive maximal number of steps for UIO-length (default %d): ", efg->num_basflow * efg->num_basflow);
    gets(inbuf);
    if (sscanf(inbuf, "%d", &steps) != 1) steps = efg->num_basflow * efg->num_basflow;
    return(steps);
}

/*****************************************************************************
 *
 * Function     : first_state
 *
 * Description  :
 *          This function returns the number of the first flowstate in a string 'inbuf' having the structure of a
 *          line from a sequence file.
 *
 * Arguments    :
 *          char    *inseq  : pointer to a string containing a sequence
 *
 * Returns      : int   : first state of sequence
 *
 *****************************************************************************/

static  int
first_state(inseq)
char    *inseq;
```

```
{
    int         i;
    char    word[10];

    for(i = 0; inseq[i] != ' '; i++) word[i] = inseq[i];
    word[i] = '\0';
    return(atoi(&word[1]));
}

/*****************************************************************************
 *
 * Function     : last_state
 *
 * Description  :
 *          This function returns the number of the last flowstate in a string 'inbuf' having the structure of a
 *          line from a sequence file.
 *
 * Arguments    :
 *          char    *inseq  : pointer to a string containing a sequence
 *
 * Returns      : int   : last state of sequence
 *
 *****************************************************************************/

static  int
last_state(inseq)
char    *inseq;
{
    char    word[10];
    int         i, j;

    i = strlen(inseq);
    while (inseq[i] != 'F') i--;
    for (j = 0; inseq[++i] != '\n'; j++)
    {
        word[j] = inseq[i];
    }
    word[j] = '\0';
    return(atoi(word));
}

/*****************************************************************************
 *
 * Function     : check_line_uniq
 *
 * Description  :
 *              This function checks of a certain action sequence if it is the same as some other action sequence in th
 *          total actions file. If so, and the endstates of these sequences are also the same, the action is marked
 *          as non unique and '0\n' is written to the total uniqueness file. If the endstates are not the same,
 *          the sequence is marked as half unique. If after comparing the sequence to the whole total actions file
 *          the action sequence is still marked half unique, '2\n' is written to the total uniqueness file. If it
 *          is still marked unique, '1\n' is written.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          int             steps   : step in which calculating UIO sequences algorithm is
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
check_line_uniq(files, steps)
files_type  *files;
int             steps;
{
    char    *inseq,
            *inact,
            *bufseq,
            *bufact;
    long    posseq,
            posact;
    int         uniq,
            half,
            non;

    inseq = (char *) malloc(sizeof(char) * 200 * steps);
    inact = (char *) malloc(sizeof(char) * 200 * steps);
    bufseq = (char *) malloc(sizeof(char) * 200 * steps);
    bufact = (char *) malloc(sizeof(char) * 200 * steps);

    fgets(inseq, (200 * steps)-1, files->totseqfp);
    fgets(inact, (200 * steps)-1, files->totactfp);
    posseq = ftell(files->totseqfp);
    posact = ftell(files->totactfp);
    rewind(files->totseqfp);
    rewind(files->totactfp);

    uniq = 1;
    half = 0;
    non = 0;
    while ((fgets(bufact, (200 * steps)-1, files->totactfp) != NULL) && (non == 0))        /* while no nonunique  */
    {                                                                                      /* actions to same state found */
        fgets(bufseq, (200 * steps)-1, files->totseqfp);
        if ((strcmp(inact, bufact) == 0) && (strcmp(inseq, bufseq) != 0))                  /* if same action found */
```

```
        {
            uniq = 0;
            if (last_state(inseq) == last_state(bufseq))                    /* if action leads to same state */
            {
                half = 0;
                non = 1;
            }
            else if (non == 0) half = 1;
        }
    }
    if (non == 1) fprintf(files->totuniqfp, "0\n");
    if (uniq == 1) fprintf(files->totuniqfp, "1\n");
    if (half == 1) fprintf(files->totuniqfp, "2\n");

    fseek(files->totseqfp, posseq, 0);
    fseek(files->totactfp, posact, 0);

    free(inseq);
    free(inact);
    free(bufseq);
    free(bufact);
}

/*****************************************************************************
 *
 * Function     : check_uniq
 *
 * Description  :
 *          This function calls for each line from the last end of the total files upto the present end of the total
 *          files the function check_line_uniq to construct the total uniqueness file. Also it sets uio_found->found
 *          for a given flowstate to 1 if in this step a UIO sequence is found.
 *
 * Arguments    :
 *          files_type      *files      : pointer to files
 *          efg_type        *efg        : pointer to EFG
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          ends_type       *ends       : pointer to structure giving positions of last ends of total files
 *          int                 steps       : step in which calculating UIO sequences algorithm is
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
check_uniq(files, efg, uio_found, ends, steps)
files_type      *files;
efg_type        *efg;
uio_found_type  *uio_found;
ends_type       *ends;
int                 steps;
{
    int         i, c;
    char    *inseq;

    fseek(files->totseqfp, ends->seq, 0);
    fseek(files->totactfp, ends->act, 0);
    fseek(files->totuniqfp, ends->uniq, 0);

    c = getc(files->totactfp);
    while (!feof(files->totactfp))
    {
        ungetc(c, files->totactfp);
        check_line_uniq(files, steps);                                  /* check all lines for uniqueness */
        c = getc(files->totactfp);
    }

    fseek(files->totuniqfp, ends->uniq, 0);
    fseek(files->totseqfp, ends->seq, 0);
    inseq = (char *) malloc(sizeof(char) * 200 * steps);

    c = getc(files->totuniqfp);
    while (!feof(files->totuniqfp))
    {
        fgets(inseq, (200 * steps)-1, files->totseqfp);
        if (c == '1')                                                   /* if an action is unique */
        {
            for(i = 0; i != efg->num_basflow; i++)
            {
                if ((&uio_found[i])->fstate == first_state(inseq))
                    (&uio_found[i])->found = 1;                         /* set uio_found of startstate to 1 */
            }
        }
        c = getc(files->totuniqfp);
        c = getc(files->totuniqfp);
    }
    free(inseq);
}

/*****************************************************************************
 *
 * Function     : cat_seq
 *
 * Description  :
 *          This function makes the string newseq out of inseq and bufseq by concatenating these strings and removing
```

```
 *          the last flowstate of string inseq.
 *
 * Arguments    :
 *          char    *newseq : pointer to new formed string
 *                  *inseq  : pointer to start string
 *                  *bufseq : pointer to end string
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
cat_seq(newseq, inseq, bufseq)
char    *newseq,
        *inseq,
        *bufseq;
{
    int i;

    strcpy(newseq, inseq);
    i = strlen(newseq);
    while (newseq[i] != 'F') i--;
    newseq[i] = '\0';
    strcat(newseq, bufseq);
}

/******************************************************************************
 *
 * Function     : cat_act
 *
 * Description  :
 *          This function makes the string newact out of inact and bufact by concatenating these strings.
 *
 * Arguments    :
 *          char    *newact : pointer to new formed string
 *                  *inact  : pointer to start string
 *                  *bufact : pointer to end string
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
cat_act(newact, inact, bufact)
char    *newact,
        *inact,
        *bufact;
{
    strcpy(newact, inact);
    newact[strlen(newact) - 1] = '\0';
    strcat(newact, bufact);
}

/******************************************************************************
 *
 * Function     : add_basic
 *
 * Description  :
 *          This function forms for given action and state sequences new sequences by concatenating these sequences
 *          with basic sequences not containing variables and starting with the same flowstate as with which the
 *          old sequences end.
 *
 * Arguments    :
 *          files_type  *files  : pointer to files
 *          char        *inseq  : string containing state sequence to be expanded
 *                      *inact  : string containing action sequence to be expanded
 *          int             steps   : step in which calculating UIO sequences algorithm is
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
add_basic(files, inseq, inact, steps)
files_type  *files;
char        *inseq,
            *inact;
int             steps;
{
    char    *bufseq,
            *bufact,
            *newseq,
            *newact;
    int         c;
    long    posseq,
            posact;

    bufseq = (char *) malloc(sizeof(char) * 200);
    bufact = (char *) malloc(sizeof(char) * 200);
    newseq = (char *) malloc(sizeof(char) * 200 * (steps+1));
    newact = (char *) malloc(sizeof(char) * 200 * (steps+1));

    posseq = ftell(files->totseqfp);
    posact = ftell(files->totactfp);
    rewind(files->basseqfp);
```

```
    rewind(files->basactfp);
    rewind(files->basuniqfp);
    fseek(files->totseqfp, 0L, 2);
    fseek(files->totactfp, 0L, 2);

    c = getc(files->basuniqfp);
    while (!feof(files->basuniqfp))                                    /* for all lines in basic files */
    {
        fgets(bufseq, 199, files->basseqfp);
        fgets(bufact, 199, files->basactfp);

        if ((c != '3') && (last_state(inseq) == first_state(bufseq)))  /* if action contains no variables   */
        {                                                              /* and first state of basic sequence */
            cat_seq(newseq, inseq, bufseq);                            /* is last state of sequence to be
*/
            cat_act(newact, inact, bufact);                            /* expanded: put total sequence and
*/
            fprintf(files->totseqfp, "%s", newseq);                    /* action in total files
*/
            fprintf(files->totactfp, "%s", newact);
        }
        c = getc(files->basuniqfp);
        c = getc(files->basuniqfp);
    }

    fseek(files->totseqfp, posseq, 0);
    fseek(files->totactfp, posact, 0);

    free(bufseq);
    free(bufact);
    free(newseq);
    free(newact);
}

/*****************************************************************************
 *
 * Function     : calc_next_seq
 *
 * Description  :
 *          This function checks for each flowstate for which not yet an UIO sequence is found which line in the
 *          total sequence files (from the last end position upto the present end position) starts with this flowstate
 *          and is halfunique. All these sequences are expanded to new sequences by calling add_basic.
 *
 * Arguments    :
 *          files_type      *files      : pointer to files
 *          efg_type        *efg        : pointer to EFG
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          ends_type       *ends       : pointer to structure giving positions of last ends of total files
 *          int                 steps       : step in which calculating UIO sequences algorithm is
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
calc_next_seq(files, efg, uio_found, ends, steps)
files_type      *files;
efg_type        *efg;
uio_found_type  *uio_found;
ends_type       *ends;
int                 steps;
{
    char    *inseq,
            *inact;
    int         i, c;
    long    old_pos_seq,
            old_pos_act,
            old_pos_uniq;

    inseq = (char *) malloc(sizeof(char) * 200 * steps);
    inact = (char *) malloc(sizeof(char) * 200 * steps);

    old_pos_seq = ends->seq;
    old_pos_act = ends->act;
    old_pos_uniq = ends->uniq;
    fseek(files->totseqfp, 0L, 2);
    fseek(files->totactfp, 0L, 2);
    fseek(files->totuniqfp, 0L, 2);
    ends->seq = ftell(files->totseqfp);
    ends->act = ftell(files->totactfp);
    ends->uniq = ftell(files->totuniqfp);

    for(i = 0; i < efg->num_basflow; i++)
    {
        if ((&uio_found[i])->found == 0)            /* for all basic flowstates for which no uio sequence is found */
        {
            fseek(files->totseqfp, old_pos_seq, 0);
            fseek(files->totactfp, old_pos_act, 0);
            fseek(files->totuniqfp, old_pos_uniq, 0);

            c = getc(files->totuniqfp);
            while (!feof(files->totuniqfp))                                    /* for all lines in total files */
            {
                fgets(inseq, (200 * steps)-1, files->totseqfp);
```

```
                fgets(inact, (200 * steps)-1, files->totactfp);

                if ((c == '2') && (first_state(inseq) == (&uio_found[i])->fstate))      /* if action is half unique */
                {                                                                       /* and starts with basic flowstate  */
                    add_basic(files, inseq, inact, steps);
                }
                c = getc(files->totuniqfp);
                c = getc(files->totuniqfp);
            }
        }
    }

    free(inseq);
    free(inact);
}

/*****************************************************************************
 *
 * Function     : print_uio_message
 *
 * Description  :
 *          This function prints on the screen fopr which basic flow states no UIO sequence is found or that for all
 *          basic flowstates UIO sequences are found.
 *
 * Arguments    :
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          efg_type        *efg        : pointer to EFG
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
print_uio_message(uio_found, efg)
uio_found_type  *uio_found;
efg_type        *efg;
{
    int i;

    if(!all_uios_found(uio_found, efg))
    {
        printf("\nNo UIO sequence found for state ");
        for(i = 0; i < efg->num_basflow; i++)
        {
            if ((&uio_found[i])->found == 0)
            {
                printf("%d, ", (&uio_found[i])->fstate);
            }
        }
        printf("\b\b.\n");

    }
    else printf("\nAll uio's found\n");
}

/*****************************************************************************
 *
 * Function     : get_uio_file
 *
 * Description  :
 *          This function makes the UIO_FILE by writing in this file for each basic flowstate a horizontal line, on
 *          the next line the flowstate (for example 'F12\n') followed by a blank line. After this blank line all
 *          found UIO state and action sequences for this flowstate are written, separated by a blank line. If no
 *          UIO sequence is found, twice the line 'NO UIO\n' is written.
 *
 * Arguments    :
 *          files_type      *files      : pointer to files
 *          efg_type        *efg        : pointer to EFG
 *          uio_found_type  *uio_found  : pointer to structure giving all basic flowstates and telling if a uio
 *                                        sequence is found for these states
 *          int                 steps       : step in which calculating UIO sequences algorithm is
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
get_uio_file(files, efg, uio_found, steps)
files_type      *files;
efg_type        *efg;
uio_found_type  *uio_found;
int                 steps;
{
    char    *inseq,
            *inact;
    int         i, c;

    inseq = (char *) malloc(sizeof(char) * 200 * steps);
    inact = (char *) malloc(sizeof(char) * 200 * steps);

    for(i = 0; i < efg->num_basflow; i++)                                   /* for all basic flowstates */
    {
        fprintf(files->uiofp, "_____________________________\n");
        fprintf(files->uiofp, "F%d\n\n", (&uio_found[i])->fstate);
```

```
        if ((&uio_found[i])->found == 1)                                    /* if unique seq exists for this state */
        {
            rewind(files->totseqfp);
            rewind(files->totactfp);
            rewind(files->totuniqfp);

            c = getc(files->totuniqfp);
            while (!feof(files->totuniqfp))                                 /* for all lines in total files */
            {
                fgets(inseq, (200 * steps)-1, files->totseqfp);
                fgets(inact, (200 * steps)-1, files->totactfp);

                if ((c == '1') && (first_state(inseq) == (&uio_found[i])->fstate))          /* if action is unique */
                {                                                           /* and starts with basic flowstate */
                    fprintf(files->uiofp, "%s", inseq);
                    fprintf(files->uiofp, "%s\n", inact);
                }
                c = getc(files->totuniqfp);
                c = getc(files->totuniqfp);
            }
        }
        else                                                                /* no unique seq exists for this state */
        {
            fprintf(files->uiofp, "NO UIO\n");
            fprintf(files->uiofp, "NO UIO\n\n");
        }
    }
    free(inseq);
    free(inact);
}

/*****************************************************************************
 *
 * Function     : make_uio
 *
 * Description  :
 *          This file is the main file of this module. It calls other functions to construct the basic state sequence,
 *          action sequence, uniqueness files and the total state sequence, action sequence and uniqueness files.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to EFG
 *          files_type  *files  : pointer to files
 *
 * Returns      : void
 *
 *****************************************************************************/

extern  void
make_uio(efg, files)
efg_type    *efg;
files_type  *files;
{
    int                 *basic;
    uio_found_type  *uio_found;
    ends_type       *ends;
    int                 steps,
                    max_step;

    open_files_uio(files);

    basic = (int *) malloc(sizeof(int) * (efg->num_flow+1));
    get_basic_seq(files, efg, basic);                                   /* make basic sequence file */
    get_basic_act(files, efg);                                          /* make basic action file */

    uio_found = (uio_found_type *) malloc(sizeof(uio_found_type) * efg->num_basflow);
    init_uio_found(uio_found, efg, basic);                              /* init struct for telling if uio is found */
    check_vars(files, efg, uio_found);      /* make basic unique file, telling for which action variables are used */
    init_tot_files(files, efg);                                         /* make total sequence and total action file */

    ends = (ends_type *) malloc(sizeof(ends_type));
    ends->seq = 0L;
    ends->act = 0L;
    ends->uniq = 0L;

    max_step = get_maxstep(efg);                    /* ask for maximal number of steps for calculating uio sequence */
    for (steps = 1; (steps <= max_step) && (!all_uios_found(uio_found, efg)); steps++)      /* for each basic state  */
    {                                                                                       /* not having an uio seq */
        printf("\nCalculating step %d of determination of UIO sequences\n", steps);
        check_uniq(files, efg, uio_found, ends, steps);                                 /* expand total unique file */
        calc_next_seq(files, efg, uio_found, ends, steps);              /* expand total sequence and action file */
    }

    print_uio_message(uio_found, efg);                                  /* give message whether all uio's are found */
    get_uio_file(files, efg, uio_found, steps);                         /* make uio file */

    free(basic);
    free(uio_found);
    free(ends);
    close_files_uio(files);
}
```

```
/*****************************************************************************
 *
 * File             : gendflow.c
 *
 * Description  :
 *          This module calculates all data flow subsequences for the EFG and writes them to DFLOW_FILE. A data
 *          flow chain is a state sequence from an 'input' to an 'output'. An 'input' can be a real input of a
 *          variable but can also be a call to a flowstate with only constants. An 'output' can be a variable
 *          in a variable output or a variable in a predicate that leads to a constant output before a basic
 *          flowstate is encountered. In the calculated data flow chains the 'output' must be influenced by the
 *          'input', e.g. the value of the 'output' depends in some way of the 'input'. The written data flow
 *          sequences are always extended at the begin and end to the nearest basic flow state. Stop criteria for
 *          calculating data flow chains are, besides the situations when there are no new definitions of variables
 *          in a new flowstate influenced by the input variable, the existence of three subsequent flowstate loops
 *          in the state sequence and reaching a maximal length in states of the data flow chain without extension
 *          to basic flow states. This maximal length is asked before calculating the data flow chains. Also is
 *          asked if data flow chains have to be calculated for variable inputs, constant calls or for both.
 *          The constructed file in this module, DFLOW_FILE, has the following structure:
 *          - A line "----------------- Real Inputs -----------------\n"
 *          - For each data flow chain where the 'input' is a real variable input:
 *                  - a line "var_out input_var output_var sequence\n" where input_var is the variable in the input,
 *                  output_var is the variable in the variable output en sequence is the total data flow chain with
 *                  extensions to the nearest basic flow state.
 *                  - a line "const_out input_var output_var sequence\n" where input_var is the variable in the input,
 *                  output_var is the variable used in a predicate leading to a constant output en sequence is the
 *                  total data flow chain with extensions to the nearest basic flow state.
 *          - A line "----------------- Constant Calls -----------------\n"
 *          - For each data flow chain where the 'input' is a constant call:
 *                  - a line "var_out input_var output_var sequence\n" where input_var is the variable in the input,
 *                  output_var is the variable in the variable output en sequence is the total data flow chain with
 *                  extensions to the nearest basic flow state.
 *                  - a line "const_out input_var output_var sequence\n" where input_var is the variable in the input,
 *                  output_var is the variable used in a predicate leading to a constant output en sequence is the
 *                  total data flow chain with extensions to the nearest basic flow state.
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : open_files_dflow
 *
 * Description  :
 *              This function opens the necessary files for this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
open_files_dflow(files)
files_type  *files;
{
    if ((files->state2fp = fopen(STATE_FILE_2, "r")) == NULL)
        err(gen_d, 6, 0);
    if ((files->trans1fp = fopen(TRANS_FILE_1, "r")) == NULL)
        err(gen_d, 2, 0);
    if ((files->dflowfp = fopen(DFLOW_FILE, "w+")) == NULL)
        err(gen_d, 31, 0);
}

/*****************************************************************************
 *
 * Function     : close_files_dflow
 *
 * Description  :
 *          This function closes the necessary files in this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
close_files_dflow(files)
files_type  *files;
{
    fclose(files->state2fp);
    fclose(files->trans1fp);
```

```
    fclose(files->dflowfp);
}

/*****************************************************************************
 *
 * Function     : var_input
 *
 * Description  :
 *          This function tries to find a next variable input in STATE_FILE_2.
 *
 * Arguments    :
 *          FILE    *fp             : STATE_FILE_2
 *          int         *input_state    : state in which input takes place
 *          char    *variab         : variable in input
 *
 * Returns      : int :
 *          0: no next variable input found
 *          1: next variable input found
 *
 *****************************************************************************/

static  int
var_input(fp, input_state, variab)
FILE    *fp;
int         *input_state;
char    *variab;
{
    char    word1[50],
            word2[50],
            word3[50],
            word4[50],
            inbuf[200],
            line[80];

    sprintf(word1, "@ST");
    while (!feof(fp))
    {
        fscanf(fp, "%s", word2);
        if (strcmp(word1, word2) == 0)
        {
            fscanf(fp, "%s", word3);
            fscanf(fp, "%s", word4);
            if (strcmp(word4, "I") == 0)
            {
                fgets(inbuf, 199, fp);                          /* rest line */
                fgets(inbuf, 199, fp);                          /* @ACT-line */
                fgets(inbuf, 199, fp);                          /* @VAR-line */
                sprintf(line, "@VAR d\n");
                if (strcmp(inbuf, line) == 0)
                {
                    fscanf(fp, "%s", variab);
                    *input_state = atoi(word3);
                    return(1);
                }
            }
        }
    }
    clearerr(fp);
    return(0);
}

/*****************************************************************************
 *
 * Function     : const_call
 *
 * Description  :
 *          This function tries to find the next call with only constants to a flowstate in STATE_FILE_2.
 *
 * Arguments    :
 *          files_type  *files      : pointer to all files
 *          int             *from_state : pointer to state from which constant call starts
 *                      *to_state   : pointer to state to which constant call takes place
 *          char        *variab         : variable defined by constant call
 *
 * Returns      : int :
 *          0: no next constant call found
 *          1: next constant call found
 *
 *****************************************************************************/

static  int
const_call(files, from_state, to_state, variab)
files_type  *files;
int             *from_state,
            *to_state;
char        *variab;
{
    char    word1[50],
            word2[50],
            word3[50],
            word4[50],
            inbuf1[80],
            inbuf2[80],
            inbuf3[80],
            line[80];
```

```
    fgets(inbuf3, 79, files->trans1fp);
    fscanf(files->state2fp, "%s", variab);
    if (strcmp(inbuf3, "\n") == 0)                          /* end of transition reached, get next @TR-line */
        fgets(inbuf3, 79, files->trans1fp);

    if (strstr(inbuf3, "@TR") == NULL) goto nextvar_label;  /* if inbuf3 is no @TR-line, skip FROM/TO/VAR-part */

    while (!feof(files->trans1fp))
    {
        if (strstr(inbuf3, "@TR") != NULL)
        {
            fgets(inbuf1, 79, files->trans1fp);                 /* @FROM-line */
            fgets(inbuf2, 79, files->trans1fp);                 /* @TO-line */
            fgets(inbuf3, 79, files->trans1fp);                 /* @VAR-line */
            sprintf(line, "@VAR c\n");
            if (strcmp(inbuf3, line) == 0)
            {
                sscanf(inbuf1, "%s%s", word1, word2);
                *from_state = atoi(word2);
                sscanf(inbuf2, "%s%s", word1, word2);
                *to_state = atoi(word2);

                rewind(files->state2fp);
                sprintf(line, "@ST %d\n", *to_state);
                find_line(line, files->state2fp);
                fgets(inbuf3, 79, files->state2fp);             /* F X-line */
                fgets(inbuf3, 79, files->state2fp);             /* @ACT-line */
                fgets(inbuf3, 79, files->state2fp);             /* @VAR-line */

                fgets(inbuf3, 79, files->trans1fp);
                fscanf(files->state2fp, "%s", variab);

                nextvar_label:                                  /* get next variable of transition */
                while (strcmp(inbuf3, "\n") != 0)
                {
                    if (strstr(inbuf3, "@CONST") != NULL)
                        return(1);

                    fgets(inbuf3, 79, files->trans1fp);
                    fscanf(files->state2fp, "%s", variab);
                }
            }
        }
        fgets(inbuf3, 79, files->trans1fp);
    }
    clearerr(files->trans1fp);
    return(0);
}

/*****************************************************************************
 *
 * Function     : is_flowstate
 *
 * Description  :
 *          This function checks if the given state is a flowstate.
 *
 * Arguments    :
 *          int             state   : state to be checked
 *          files_type  *files  : pointer to all files
 *
 * Returns      : int :
 *          0: state is no flowstate
 *          1: state is flowstate
 *
 *****************************************************************************/

static  int
is_flowstate(state, files)
int             state;
files_type  *files;
{
    char    line[80];
    char    word[40];
    long    pos_state;

    pos_state = ftell(files->state2fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);;                                  /* find '@ST state' */

    if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
    if(strcmp(word, "F") == 0)
    {
        fseek(files->state2fp, pos_state, 0);
        return(1);
    }
    else
    {
        fseek(files->state2fp, pos_state, 0);
        return(0);
    }
}

/*****************************************************************************
```

```
 *
 * Function      : is_inputstate
 *
 * Description   :
 *          This function checks if the given state is a inputstate.
 *
 * Arguments     :
 *          int             state   : state to be checked
 *          files_type  *files  : pointer to all files
 *
 * Returns       : int :
 *          0: state is no inputstate
 *          1: state is inputstate
 *
 ******************************************************************************/

extern  int
is_inputstate(state, files)
int             state;
files_type  *files;
{
    char    line[80];
    char    word[40];
    long    pos_state;

    pos_state = ftell(files->state2fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);;                                  /* find '@ST state' */

    if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
    if(strcmp(word, "I") == 0)
    {
        fseek(files->state2fp, pos_state, 0);
        return(1);
    }
    else
    {
        fseek(files->state2fp, pos_state, 0);
        return(0);
    }
}

/******************************************************************************
 *
 * Function      : is_startstate
 *
 * Description   :
 *          This function checks if the given state is a startstate.
 *
 * Arguments     :
 *          int             state   : state to be checked
 *          files_type  *files  : pointer to all files
 *
 * Returns       : int :
 *          0: state is no startstate
 *          1: state is startstate
 *
 ******************************************************************************/

static  int
is_startstate(state, files)
int             state;
files_type  *files;
{
    char    line[80];
    char    word[40];
    long    pos_state;

    pos_state = ftell(files->state2fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);;                                  /* find '@ST state' */

    if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
    if(strcmp(word, "S") == 0)
    {
        fseek(files->state2fp, pos_state, 0);
        return(1);
    }
    else
    {
        fseek(files->state2fp, pos_state, 0);
        return(0);
    }
}

/******************************************************************************
 *
 * Function      : start_seq_inputs
 *
 * Description   :
 *          This function constructs a state sequence starting in a basic flowstate or startstate and ending in
 *          the inputstate.
 *
```

```
 * Arguments    :
 *          files_type  *files      : pointer to all files
 *          char        *seq        : state sequence to be constructed
 *          int             input_state : endstate of sequence
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
start_seq_inputs(files, seq, input_state)
files_type  *files;
char        *seq;
int             input_state;
{
    long    pos_state,
            pos_trans;
    int         state,
            end,
            i,
            trans;
    char    line[80],
            word[50];

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);
    state = input_state;
    strcpy(seq, "");
    end = 0;

    do
    {
        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);

        add_next_state_begin(files, seq);

        if ((is_basic(state, files) && is_flowstate(state, files)) || (is_startstate(state, files)))
            end = 1;
        else
        {
            sprintf(line, "@IN\n");
            find_line(line, files->state2fp);
            if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
            trans = atoi(word);
            rewind(files->trans1fp);
            sprintf(line, "@TR %d\n", trans);
            find_line(line, files->trans1fp);
            if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);
            if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);
            state = atoi(word);                                         /* get previous state */
        }
    } while (!end);
    seq[strlen(seq) - 1] = '\0';                                    /* remove last space */

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
}

/******************************************************************************
 *
 * Function     : start_seq_const_calls
 *
 * Description  :
 *          This function constructs a state sequence starting in a basic flowstate or startstate and ending in
 *          the the flowstate called by a constant call. The state from which the constant call takes place
 *          is also included in the sequence.
 *
 * Arguments    :
 *          files_type  *files      : pointer to all files
 *          char        *seq        : state sequence to be constructed
 *          int             from_state  : second last state of startsequence
 *                      to_state    : last state of startsequence
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
start_seq_const_calls(files, seq, from_state, to_state)
files_type  *files;
char        *seq;
int             from_state,
            to_state;
{
    long    pos_state,
            pos_trans;
    int         state,
            end,
            i,
            trans;
    char    line[80],
            word[50];
```

```
    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);
    state = from_state;
    strcpy(seq, "");
    end = 0;

    do
    {
        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);

        add_next_state_begin(files, seq);

        if ((is_basic(state, files) && is_flowstate(state, files)) || (is_startstate(state, files)))
            end = 1;
        else
        {
            sprintf(line, "@IN\n");
            find_line(line, files->state2fp);
            if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
            trans = atoi(word);
            rewind(files->trans1fp);
            sprintf(line, "@TR %d\n", trans);
            find_line(line, files->trans1fp);
            if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);
            if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);
            state = atoi(word);                                          /* get previous state */
        }
    } while (!end);
    seq[strlen(seq) - 1] = '\0';                                        /* remove last space */

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", to_state);
    find_line(line, files->state2fp);

    add_next_state(files, seq);

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : get_prev_seq_state
 *
 * Description  :
 *          This function returns the second last state of a sequence with a space inserted and a '\n' at the end.
 *
 * Arguments    :
 *          char        *seq    : sequence from which to derive second last state
 *
 * Returns      : char * :
 *          second last state with space inserted and \n at end
 *
 *****************************************************************************/

static  char *
get_prev_seq_state(seq)
char        *seq;
{
    int         i, j;
    char    word[10];

    i = strlen(seq);
    while (seq[i] != ' ') i--;
    i--;
    while ((seq[i] != ' ') && (i >= 0)) i--;                /* goto begin of previous state */

    word[0] = seq[++i];
    word[1] = ' ';
    for (j = 2; seq[++i] != ' '; j++)
    {
        word[j] = seq[i];
    }
    word[j] = '\n';
    word[j+1] = '\0';                                       /* get previous state with space inserted and \n at end */
    return(word);
}

/*****************************************************************************
 *
 * Function     : no_new_def
 *
 * Description  :
 *          This function checks whether the given state is a flowstate defining a variable that is influenced
 *          by the variable in the previous sequence state.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          int             state   : state to be checked
 *          char        *seq    : present sequence
 *                      *variab     : variable to be checked in call to state
 *
 * Returns      : int :
```

```
 *          0: state is no flowstate or flowstate is called with variable
 *          1: state is not called with variable
 *
 ******************************************************************************/

static  int
no_new_def(files, state, seq, variab)
files_type  *files;
int             state;
char        *seq,
            *variab;
{
    long    pos_state,
            pos_trans;
    char    line[80],
            inbuf[200],
            word[40];
    int         trans;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    if (is_flowstate(state, files))
    {
        rewind(files->state2fp);
        strcpy(line, get_prev_seq_state(seq));
        find_line(line, files->state2fp);
        sprintf(line, "@OUT\n");
        find_line(line, files->state2fp);
        if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
        trans = atoi(word);                                         /* get outgoing trans */
        rewind(files->trans1fp);
        sprintf(line, "@TR %d\n", trans);
        find_line(line, files->trans1fp);
        fgets(inbuf, 199, files->trans1fp);                         /* @FROM-line */
        fgets(inbuf, 199, files->trans1fp);                         /* @TO-line */
        fgets(inbuf, 199, files->trans1fp);                         /* @VAR-line */

        if (fscanf(files->trans1fp, "%s", word) == EOF) strcpy(word, "@TR");
        while (strcmp(word, "@TR") != 0)
        {
            if (strcmp(word, variab) == 0)                  /* check if variab is used in call to flowstate */
            {
                fseek(files->state2fp, pos_state, 0);
                fseek(files->trans1fp, pos_trans, 0);
                return(0);
            }

            if (fscanf(files->trans1fp, "%s", word) == EOF) strcpy(word, "@TR");        }

        fseek(files->state2fp, pos_state, 0);
        fseek(files->trans1fp, pos_trans, 0);
        return(1);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
    return(0);
}

/******************************************************************************
 *
 * Function     : three_loops
 *
 * Description  :
 *          This function checks whether the sequence contains three subsequent same loops starting and ending
 *          in some flowstate.
 *
 * Arguments    :
 *          char        *seq    : present state sequence
 *
 * Returns      : int :
 *          0: no three loops
 *          1: three loops
 *
 ******************************************************************************/

static  int
three_loops(seq)
char        *seq;
{
    char    *seq2,
            *seq3,
            *seq4,
            word[50],
            *ptr;
    int         i, j,
            seqlen;
    char    c;

    seq2 = (char *) malloc (strlen(seq) * sizeof(char));
    seq3 = (char *) malloc (strlen(seq) * sizeof(char));
    seq4 = (char *) malloc (strlen(seq) * sizeof(char));

    for (i = 0; seq[i] != '\0'; i++)                                /* check total sequence */
```

```
    {
        if (seq[i] == 'F')
        {
            for (j = 0; isalnum(seq[i]); j++)
            {
                word[j] = seq[i];
                i++;
            }
            word[j] = ' ';
            word[j+1] = '\0';                                   /* get flowstate sequence */

            strcpy(seq2, &seq[i - strlen(word) + 1]);           /* remove states until flowstate */
            seqlen = 0;
            while ((ptr = strstr(&seq2[seqlen+1], word)) != NULL)   /* check total rest of sequence */
            {
                seqlen = strlen(seq2) - strlen(ptr);
                c = *ptr;
                *ptr = '\0';
                strcpy(seq3, seq2);                                 /* seq3 becomes loop part to be checked
*/
                *ptr = c;
                strcpy(seq4, ptr);                              /* seq4 becomes rest of seq2 */

                if (strstr(seq4, seq3) == seq4)                     /* check if two loops */
                {
                    strcpy(seq4, &seq4[strlen(seq3)]);
                    if (strstr(seq4, seq3) == seq4)             /* check if three loops */
                    {
                        strcpy(seq4, &seq4[strlen(seq3)]);
                        word[j] = '\0';
                        if ((strstr(seq4, word) == seq4) && (!isdigit(seq4[strlen(word)])))
                        {                                           /* check also last state */
                            free(seq2);
                            free(seq3);
                            free(seq4);
                            return(1);
                        }
                        word[j] = ' ';
                        word[j+1] = '\0';
                    }
                }
            }
        }
    }
    free(seq2);
    free(seq3);
    free(seq4);
    return(0);
}

/*****************************************************************************
 *
 * Function     : flow_step
 *
 * Description  :
 *          This function executes a next step in deriving data flow chains for the given flowstate, e.g. this
 *          function calls step_seq for the flowstate for all variables defined in this flowstate that are
 *          influenced by the start variable of the data flow chains.
 *
 * Arguments    :
 *          files_type  *files          : pointer to all files
 *          char        *seq            : present state sequence
 *          int             fromstate       : previous state of step
 *                      state           : present state for step
 *          char        *start_variab   : start variable for data flow chain
 *                      *variab             : present variable for data flow chain
 *          int             maxstep             : maximul number of steps to be taken
 *                      step            : present step
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
flow_step(files, seq, fromstate, state, start_variab, variab, maxstep, step)
files_type  *files;
char        *seq;
int             fromstate,
            state;
char        *start_variab,
            *variab;
int             maxstep,
            step;
{
    static  void    step_seq();

    long    pos_state,
            pos_trans,
            pos_state2,
            pos_trans2;
    char    line[80],
            word[40],
            inbuf[200],
            new_variab[30];
    int         trans;
```

```
    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", fromstate);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);
    if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 29, 0);
    trans = atoi(word);
    rewind(files->trans1fp);
    sprintf(line, "@TR %d\n", trans);                    /* go to outgoing trans of fromstate */
    find_line(line, files->trans1fp);
    fgets(inbuf, 199, files->trans1fp);                  /* @FROM-line */
    fgets(inbuf, 199, files->trans1fp);                  /* @TO-line */
    fgets(inbuf, 199, files->trans1fp);                  /* @VAR-line */
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);                    /* go to (to)state */
    find_line(line, files->state2fp);
    fgets(inbuf, 199, files->state2fp);                  /* F X-line */
    fgets(inbuf, 199, files->state2fp);                  /* @ACT-line */
    fgets(inbuf, 199, files->state2fp);                  /* @VAR-line */

    fgets(inbuf, 199, files->trans1fp);
    fscanf(files->state2fp, "%s", new_variab);
    pos_state2 = ftell(files->state2fp);
    pos_trans2 = ftell(files->trans1fp);
    while (strcmp(inbuf, "\n") != 0)                     /* check all calls for variable existence */
    {
        if (strstr(inbuf, variab) != NULL)
            step_seq(files, seq, state, start_variab, new_variab, variab, maxstep, step);

        fseek(files->state2fp, pos_state2, 0);
        fseek(files->trans1fp, pos_trans2, 0);
        fgets(inbuf, 199, files->trans1fp);
        fscanf(files->state2fp, "%s", new_variab);
        pos_state2 = ftell(files->state2fp);
        pos_trans2 = ftell(files->trans1fp);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
}
/*****************************************************************************
 *
 * Function     : finish_seq
 *
 * Description  :
 *          This function finishes the given sequence to the first basic flowstate to be found.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          char        *seq    : state sequence to be finished
 *          int             state   : start state for finishing
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
finish_seq(files, seq, state)
files_type  *files;
char        *seq;
int             state;
{
    long    pos_state,
            pos_trans;
    int         i,
            tmp_state,
            trans;
    char    word[80],
            inbuf[80],
            line[80];

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    tmp_state = state;
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", tmp_state);
    find_line(line, files->state2fp);

    while (!(is_basic(tmp_state, files) && is_flowstate(tmp_state, files)))
    {
        sprintf(line, "@OUT\n");
        find_line(line, files->state2fp);
        if (fscanf(files->state2fp, "%s", word) == EOF) err(gen_d, 30, 0);
        trans = atoi(word);
        rewind(files->trans1fp);
        sprintf(line, "@TR %d\n", trans);
        find_line(line, files->trans1fp);                                   /* go to outgoing trans of state */
        fgets(inbuf, 79, files->trans1fp);                                  /* @FROM-line */
        if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);  /* @TO */
        if (fscanf(files->trans1fp, "%s", word) == EOF) err(gen_d, 29, 0);  /* tostate */
```

```
        tmp_state = atoi(word);

        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", tmp_state);
        find_line(line, files->state2fp);

        add_next_state(files, seq);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : var_in_output
 *
 * Description  :
 *          This function checks if the variable is used in the output to be checked upon.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          int             state   : outputstate to be checked
 *          char        *variab     : variable to be checked if it is used in output
 *
 * Returns      : int :
 *          0: variable not in output
 *          1: variable in output
 *
 *****************************************************************************/

static  int
var_in_output(files, state, variab)
files_type  *files;
int             state;
char        *variab;
{
    long    pos_state;
    char    line[80],
            inbuf[400];

    pos_state = ftell(files->state2fp);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);
    fgets(inbuf, 399, files->state2fp);                                 /* O X-line */

    if (strstr(inbuf, variab) != NULL)
    {
        fseek(files->state2fp, pos_state, 0);
        return(1);
    }
    else
    {
        fseek(files->state2fp, pos_state, 0);
        return(0);
    }
}

/*****************************************************************************
 *
 * Function     : write_seq
 *
 * Description  :
 *          This function writes a line containing a data flow chain to DFLOW_FILE, starting with 'var_out' if the
 *          output, is a variable output, 'const_out' if it is a chain for a constant output. Next the start
 *          variable and the end variable of the data flow chain are written and the state sequence leading from
 *          an input to an output starting and ending in a basic flowstate.
 *
 * Arguments    :
 *          files_type  *files          : pointer to all files
 *          char        *seq            : sequence to be written
 *                      *start_variab   : startvariable of data flow chain
 *                      *variab             : endvariable of data flow chain
 *          int             output_sort     : integer telling whether output is variable or constant
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_seq(files, seq, start_variab, variab, output_sort)
files_type  *files;
char        *seq,
            *start_variab,
            *variab;
int             output_sort;
{
    if (output_sort == 1)
        fprintf(files->dflowfp, "var_out %s %s %s\n", start_variab, variab, seq);
    else        /* output_sort == 2 */
        fprintf(files->dflowfp, "const_out %s %s %s\n", start_variab, variab, seq);
}
```

```
/*****************************************************************************
 *
 * Function     : cut_last_seq_state
 *
 * Description  :
 *          This function returns the last state of a sequence with a space inserted. Also it returns the sequence
 *          without this last state.
 *
 * Arguments    :
 *          char    *seq    : sequence to be cutted
 *
 * Returns      : char * :
 *          last sequence state with space inserted
 *
 *****************************************************************************/

static  char *
cut_last_seq_state(seq)
char    *seq;
{
    int         i, j,
                cut;
    char        word[10];

    i = strlen(seq);
    while ((seq[i] != ' ') && (i >= 0)) i--;                /* go to point before start of last state */
    cut = i;

    word[0] = seq[++i];
    word[1] = ' ';
    for (j = 2; isalnum(seq[++i]); j++)
    {
        word[j] = seq[i];
    }
    word[j] = '\0';                                         /* word becomes last state with space */
    if (cut != -1) seq[cut] = '\0';                         /* remove last state */
    return(word);
}

/*****************************************************************************
 *
 * Function     : intern_num
 *
 * Description  :
 *          This function returns the internal number, so the number of the @ST-line, of a sequence state returned
 *          by cut_last_seq_state.
 *
 * Arguments    :
 *          files_type  *files      : pointer to all files
 *          char        *seq_state  : sequence state to be checked
 *
 * Returns      : int :
 *          internal number of sequence state
 *
 *****************************************************************************/

static  int
intern_num(files, seq_state)
files_type  *files;
char        *seq_state;
{
    long    pos_state;
    char    line[200],
            inbuf_old[200],
            inbuf_new[200],
            word1[20],
            word2[20];

    pos_state = ftell(files->state2fp);

    sprintf(line, "%s\n", seq_state);
    rewind(files->state2fp);
    fgets(inbuf_new, 199, files->state2fp);

    do
    {
        strcpy(inbuf_old, inbuf_new);
        fgets(inbuf_new, 199, files->state2fp);
    } while (strcmp(line, inbuf_new) != 0);                 /* find seq_state-line */
    if (feof(files->state2fp)) err(gen_d, 30, 0);

    sscanf(inbuf_old, "%s%s", word1, word2);                /* inbuf_old is @ST-line */

    fseek(files->state2fp, pos_state, 0);
    return(atoi(word2));
}

/*****************************************************************************
 *
 * Function     : pred_trans
 *
 * Description  :
 *          This function checks if the transition from the from_state to the to_state contains predicates.
 *
 * Arguments    :
```

```
 *          files_type  *files      : pointer to all files
 *          int             from_state  : fromstate of transition
 *                      to_state    : tostate of transition
 *
 * Returns      : int :
 *          0: no predicate transition
 *          1: predicate transition
 *
 ******************************************************************************/

static  int
pred_trans(files, from_state, to_state)
files_type  *files;
int             from_state,
            to_state;
{
    long    pos_state,
            pos_trans;
    char    line[200],
            inbuf[200],
            seq_state[20],
            word[50],
            word2[50];
    int         trans;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", from_state);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);

    if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
    while (strcmp(word, "@ST") != 0)                        /* check all outgoing transitions of from_state */
    {
        trans = atoi(word);
        rewind(files->trans1fp);
        sprintf(line, "@TR %d\n", trans);
        find_line(line, files->trans1fp);
        fgets(inbuf, 199, files->trans1fp);                 /* @FROM-line */
        fgets(inbuf, 199, files->trans1fp);                 /* @TO-line */
        sscanf(inbuf, "%s%s", word, word2);
        if (atoi(word2) == to_state)                        /* if to_state of trans is the right one */
        {
            fgets(inbuf, 199, files->trans1fp);                 /* @VAR-line */
            sprintf(line, "@VAR p\n");
            if (strcmp(line, inbuf) == 0)
            {
                fseek(files->state2fp, pos_state, 0);
                return(1);
            }
            else
            {
                fseek(files->state2fp, pos_state, 0);
                fseek(files->trans1fp, pos_trans, 0);
                return(0);
            }
        }

        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
    }
}

/******************************************************************************
 *
 * Function     : output_step
 *
 * Description  :
 *          This function checks if data flow chains can be written to the DFLOW_FILE because of the existence of
 *          outputs influenced by an input in the present outputstate. If the output is a variable output it is
 *          checked whether the variable in the output is the variable influenced by the input. If so, this data
 *          flow chain is written. If the output is a constant output this function searches back for predicate
 *          transitions with the influenced variable before a basic flow state is encountered. If this is the case,
 *          the data flow chain is written.
 *
 * Arguments    :
 *          files_type  *files          : pointer to all files
 *          char        *seq            : present state sequence
 *          int             state           : outputstate
 *          char        *start_variab   : startvariable of data flow chain
 *                      *variab             : present variable of data flow chain
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
output_step(files, seq, state, start_variab, variab)
files_type  *files;
char        *seq;
int             state;
char        *start_variab,
            *variab;
{
```

```
    long    pos_state,
            pos_trans;
    char    *seq2,
            *seq3,
            word[50],
            seq_state[20],
            new_variab[50];
    int         to_state,
            from_state,
            end;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    seq2 = (char *) malloc((strlen(seq) + 120) * sizeof(char));
    strcpy(seq2, seq);
    finish_seq(files, seq2, state);

    if (var_in_output(files, state, variab))                /* variable in output */
    {
        write_seq(files, seq2, start_variab, variab, 1);
    }
    else                                                    /* constant output or variable output without 'variab' */
    {
        seq3 = (char *) malloc((strlen(seq)) * sizeof(char));
        strcpy(seq3, seq);

        strcpy(seq_state, cut_last_seq_state(seq3));
        from_state = intern_num(files, seq_state);          /* get number last state */
        end = 0;
        do
        {
            to_state = from_state;
            strcpy(seq_state, cut_last_seq_state(seq3));
            from_state = intern_num(files, seq_state);              /* get number last state */
            if (pred_trans(files, from_state, to_state))            /* if transition contains predicates */
            {
                if (fscanf(files->trans1fp, "%s", new_variab) == EOF) strcpy(word, "@TR");
                while (strcmp(new_variab, "@TR") != 0)              /* for each predicate use of new_variab */
                {
                    if (strcmp(variab, new_variab) == 0)            /* if variable is the influenced variable */
                    {
                        write_seq(files, seq2, start_variab, new_variab, 2);
                        end = 1;
                    }
                    if (fscanf(files->trans1fp, "%s", new_variab) == EOF) strcpy(word, "@TR");
                }
            }
        } while ((!(is_flowstate(from_state, files) && is_basic(from_state, files))) && (end == 0));
        free(seq3);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
    free(seq2);
}

/*****************************************************************************
 *
 * Function     : step_seq
 *
 * Description  :
 *          This function performs a next step in deriving data flow chains by getting the next state. Next, if this
 *          state is a flowstate, flow_step is called. If the state is an outputstate output_step is called to
 *          write possible data flow chains. after this the next step of step_seq is called. If the state is no
 *          flowstate or outputstate the next step of step_seq is called.
 *
 * Arguments    :
 *          files_type  *files          : pointer to all files
 *          char        *seq            : present state sequence
 *          int             state           : present state for step
 *          char        *start_variab   : start variable of data flow chain
 *                      *variab             : present variable of data flow chain
 *                      *prev_variab    : in case of new flowstate, variable used in last flowstate
 *          int             maxstep             : maximal number of steps to be taken
 *                      step            : present step
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
step_seq(files, seq, state, start_variab, variab, prev_variab, maxstep, step)
files_type  *files;
char        *seq;
int             state;
char        *start_variab,
            *variab,
            *prev_variab;
int             maxstep,
            step;
{
    long    pos_state,
            pos_trans,
            pos_state2;
```

```
    char    *seq2,
            *seq3,
            line[200],
            word[50];
    int         tostate;

    printf("\b\b\b\b%-4d", step);
    fflush(stdout);

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);
    seq2 = (char *) malloc((strlen(seq) + 6) * sizeof(char));
    seq3 = (char *) malloc((strlen(seq) + 6) * sizeof(char));
    strcpy(seq2, seq);

    /* if state contains no new defs influenced by variable or three loops in seq or maximum number of steps */
    /* reached : skip the sequence and return                                                                 */
    if (((no_new_def(files, state, seq2, prev_variab)) && (step != 1)) || (three_loops(seq2)) || (step >= maxstep));
    else
    {
        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);
        sprintf(line, "@OUT\n");
        find_line(line, files->state2fp);

        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
        pos_state2 = ftell(files->state2fp);
        strcpy(seq3, seq2);
        while (strcmp(word, "@ST") != 0)                                    /* for each outgoing trans */
        {
            strcpy(seq2, seq3);
            tostate = find_st_line_next_state(files, word);
            strcpy(word, add_next_state(files, seq2));

            if (strcmp(word, "F") == 0)
            {
                flow_step(files, seq2, state, tostate, start_variab, variab, maxstep, step+1);
            }
            else
            {
                if (strcmp(word, "O") == 0)
                {
                    output_step(files, seq2, tostate, start_variab, variab);
                    step_seq(files, seq2, tostate, start_variab, variab, variab, maxstep, step+1);
                }
                else
                {
                    step_seq(files, seq2, tostate, start_variab, variab, variab, maxstep, step+1);
                }
            }

            fseek(files->state2fp, pos_state2, 0);
            if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
            pos_state2 = ftell(files->state2fp);
        }
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
    free(seq2);
    free(seq3);
}

/*****************************************************************************
 *
 * Function     : gen_for_inputs
 *
 * Description  :
 *          This function calls var_input to get next variable inputs. Next, it calls start_seq to get an initial
 *          sequence and it calls step_seq for the first step.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          efg_type    *efg    : pointer to EFG
 *          int             maxstep     : maximal number of steps to be taken
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
gen_for_inputs(files, efg, maxstep)
files_type  *files;
efg_type    *efg;
int             maxstep;
{
    char    *seq;
    char    variab[50];
    int         input_state;

    seq = (char *) malloc(120 * sizeof(char));

    rewind(files->state2fp);
```

```
    printf("\nCalculating Data Flow subsequences for real inputs   , length of sequence : none");
    fprintf(files->dflowfp, "------------------ Real Inputs ------------------\n");

    while (var_input(files->state2fp, &input_state, variab))                    /* get next variable input */
    {
        start_seq_inputs(files, seq, input_state);
        step_seq(files, seq, input_state, variab, variab, variab, maxstep, 1);
    }

    free(seq);
}

/*****************************************************************************
 *
 * Function     : gen_for_const_calls
 *
 * Description  :
 *          This function calls const_call to get next variable constant calls. Next, it calls start_seq_const_call
 *          to construct a initial sequence and it calls step_seq for the first step.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          efg_type    *efg    : pointer to EFG
 *          int             maxstep     : maximal number of steps to be taken
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
gen_for_const_calls(files, efg, maxstep)
files_type  *files;
efg_type    *efg;
int             maxstep;
{
    char    *seq;
    char    variab[50];
    int         from_state,
            to_state;

    seq = (char *) malloc(120 * sizeof(char));

    rewind(files->trans1fp);

    printf("\nCalculating Data Flow subsequences for constant calls, length of sequence : none");
    fprintf(files->dflowfp, "------------------ Constant Calls ------------------\n");

    while (const_call(files, &from_state, &to_state, variab))                   /* get next variable in call */
    {
        start_seq_const_calls(files, seq, from_state, to_state);
        step_seq(files, seq, to_state, variab, variab, variab, maxstep, 1);
    }

    free(seq);
}

/*****************************************************************************
 *
 * Function     : gen_dflow
 *
 * Description  :
 *          This function is the main function of this module. It asks the maximal number of steps to be performed,
 *          it asks is you want to derive data flow chains for variable inputs and constant calls and calls the
 *          functions gen_for_inputs or gen_for_const_calls depending on the answer of this last question.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to EFG
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

extern  void
gen_dflow(efg, files)
efg_type    *efg;
files_type  *files;
{
    int         maxstep;
    char    inbuf[80];
    int         menu;

    open_files_dflow(files);

    do
    {
        printf("\nGive maximal length of sequences to look for (max 500 states) : ");
        gets(inbuf);
        sscanf(inbuf, "%d", &maxstep);
    } while ((maxstep < 1) || (maxstep > 500));

    printf("\nFor which 'inputs' do you want to derive Data Flow testsubsequences?");
    printf("\n\n1       Inputs of Variables and Constant Calls");
    printf("\n2     Only Inputs of Variables");
    printf("\n3     Only Constant Calls");
```

```
    printf("\n\n0       No derivation Data Flow testsubsequences\n");
    do
    {
        printf("\nYour choice? ");
        gets(inbuf);
        sscanf(inbuf, "%d", &menu);
    } while ((menu < 0) || (menu > 3));


    if ((menu == 1) || (menu == 2))
        gen_for_inputs(files, efg, maxstep);

    if ((menu == 1) || (menu == 3))
        gen_for_const_calls(files, efg, maxstep);

    close_files_dflow(files);
}
```

```
/*****************************************************************************
 *
 * File             : ttcn.c
 *
 * Description  :
 *          This module constructs TTCN test cases for the control flow and it transforms the UIO sequences and
 *          the data flow sequences formed in makeuio.c and gendflow.c to TTCN. The transformation of link
 *          subsequences is not yet implementated. The TTCN test cases for control flow are written, with a
 *          header giving the purpose of the test case, its identifier and some comment lines giving information
 *          about the test case, to CFLOW_TTCN_FILE. The algorithm to construct TTCN control flow test cases
 *          is als follows:
 *
 *          port!fs
 *              possible I's
 *                  -if reception before basic state
 *                  port?fr
 *                      possible I's
 *                          +UIO sequence basic flow state                                  pass
 *                  port?other_frames
 *                      possible I's
 *                          +UIO sequences paths basic flow states                          pass
 *                  ?OTHERWISE                                                              fail
 *                  ?EXPIRATION OF TIMER                                                    fail
 *
 *                  -if no reception and basic flow state is unstable
 *                  +UIO sequence basic flow state(s)                                       pass
 *
 *                  -if no reception and basic flow state is stable
 *                  START TIMER
 *                      ?TIME_OUT
 *                          +UIO sequence basic flow state                                  pass
 *                      ?OTHERWISE                                                          fail
 *
 *          The UIO sequences and data flow subsequences are read from DFLOW_FILE and UIO_FILE (only one UIO
 *          sequence for each basic state) and transformed to TTCN and written, preceeded by a header and
 *          followed by comment lines, to DFLOW_TTCN_FILE and UIO_TTCN_FILE. The following algorithm is used:
 *
 *          actions
 *
 *              -at any unstable flow state:
 *              -for all paths leading to a reception (output in EFG)
 *              possible I's
 *                  port?fr                                                             inconclusive
 *                  ?OTHERWISE                                                          fail
 *                  ?EXPIRATION OF TIMER                                                fail
 *              -for all paths leading to a stable flow state via I's
 *              possible I's, at least one
 *                  ?EXPIRATION OF TIMER                                                inconclusive
 *
 *                      actions                                                         pass
 *
 *          ____________________
 *          -for each intended reception
 *          port?fr
 *          ?OTHERWISE                                                                  fail
 *          ?EXPIRATION OF TIMER                                                        fail
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : open_files_ttcn
 *
 * Description  :
 *              This function opens the necessary files for this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
open_files_ttcn(files)
files_type  *files;
{
    if ((files->state2fp = fopen(STATE_FILE_2, "r")) == NULL)
        err(ttcn, 6, 0);
    if ((files->trans1fp = fopen(TRANS_FILE_1, "r")) == NULL)
        err(ttcn, 2, 0);
    if ((files->uiofp = fopen(UIO_FILE, "r")) == NULL)
        err(ttcn, 27, 0);
    if ((files->dflowfp = fopen(DFLOW_FILE, "r")) == NULL)
```

```
        err(ttcn, 31, 0);
    if ((files->linkfp = fopen(LINK_FILE, "r")) == NULL)
        err(ttcn, 32, 0);
    if ((files->cflowttcnfp = fopen(CFLOW_TTCN_FILE, "w")) == NULL)
        err(ttcn, 33, 0);
    if ((files->uiottcnfp = fopen(UIO_TTCN_FILE, "w")) == NULL)
        err(ttcn, 34, 0);
    if ((files->dflowttcnfp = fopen(DFLOW_TTCN_FILE, "w")) == NULL)
        err(ttcn, 35, 0);
    if ((files->linkttcnfp = fopen(LINK_TTCN_FILE, "w")) == NULL)
        err(ttcn, 36, 0);
}

/*****************************************************************************
 *
 * Function     : close_files_ttcn
 *
 * Description  :
 *          This function closes the necessary files in this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
close_files_ttcn(files)
files_type  *files;
{
    fclose(files->state2fp);
    fclose(files->trans1fp);
    fclose(files->uiofp);
    fclose(files->dflowfp);
    fclose(files->linkfp);
    fclose(files->cflowttcnfp);
    fclose(files->uiottcnfp);
    fclose(files->dflowttcnfp);
    fclose(files->linkttcnfp);
}

/*****************************************************************************
 *
 * Function     : init_ttcn_el
 *
 * Description  :
 *          This function allocates memory for a ttcn element and resets all elements of it.
 *
 * Arguments    : none
 *
 * Returns      : ttcn_el_type * :
 *          pointer to initialized ttcn element
 *
 *****************************************************************************/

static  ttcn_el_type *
init_ttcn_el()
{
    ttcn_el_type    *elem;

    elem = (ttcn_el_type *) malloc(sizeof(ttcn_el_type));

    strcpy(elem->action,"");
    strcpy(elem->comment,"");
    elem->verdict = none;
    elem->nl = NULL;
    elem->np = NULL;
    elem->indent = 0;
    elem->state = 0;
    return(elem);
}

/*****************************************************************************
 *
 * Function     : inv_inout
 *
 * Description  :
 *          This function replaces '?' by '!' and vice versa.
 *
 * Arguments    :
 *          char    *action : pointer to action to be inverted
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
inv_inout(action)
char    *action;
{
    int     i;

    for (i=0; i<strlen(action); i++)
        if (action[i] == '?') action[i] = '&';
```

```
    for (i=0; i<strlen(action); i++)
        if (action[i] == '!') action[i] = '?';

    for (i=0; i<strlen(action); i++)
        if (action[i] == '&') action[i] = '!';

}
/*****************************************************************************
 *
 * Function     : fill_first_cflow
 *
 * Description  :
 *          This function fills first element of cflow test case with info about the input to be tested.
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          ttcn_el_type    *elem       : pointer to ttcn element to be updated
 *          int                 flowstate   : number of flowstate as in F X
 *                          inputstate  : intern number of inputstate
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
fill_first_cflow(files, elem, flowstate, inputstate)
files_type      *files;
ttcn_el_type    *elem;
int                 flowstate,
                inputstate;
{
    long    pos_state,
            pos_trans;
    char    line[200],
            inbuf[200];

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", inputstate);
    find_line(line, files->state2fp);
    fgets(inbuf, 199, files->state2fp);                                     /* I X-line */
    fgets(inbuf, 199, files->state2fp);                                     /* @ACT-line */
    inbuf[strlen(inbuf)-1] = '\0';                                          /* remove '\n' */
    strcpy(elem->action, &inbuf[5]);                                        /* remove @ACT */
    inv_inout(elem->action);                                                /* inverse ?! */

    sprintf(elem->comment, "Start in F%d, check input '%s'", flowstate, &inbuf[5]);

    elem->state = inputstate;

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : create_next_elem
 *
 * Description  :
 *          This function creates if necessary and returns the last empty element of the present level of a
 *          ttcn element if first is 1, else it creates if necessary and returns the last element of the next
 *          level of the ttcn element.
 *
 * Arguments    :
 *          ttcn_el_type    *elem   : pointer to ttcn element to be expanded
 *          int                 first   : is 1 if elem is first element
 *
 * Returns      : ttcn_el_type * :
 *          pointer to created ttcn element
 *
 *****************************************************************************/

static  ttcn_el_type *
create_next_elem(elem, first)
ttcn_el_type    *elem;
int                 first;
{
    ttcn_el_type    *tmp;

    if (first == 1)                                                 /* no previous element, so create on this lev
*/
    {
        if (strcmp(elem, "") == 0)                              /* elem not used */
            return(elem);
        else                                                    /* find last used next possibility */
        {
            tmp = elem;
            while (tmp->np != NULL)
                tmp = tmp->np;
            tmp->np = init_ttcn_el();
            return(tmp->np);
        }
    }
```

```
    else                                                        /* elem is previous level */
    {
        if (strcmp(elem, "") == 0)
            return(elem);

        if (elem->nl == NULL)                                   /* first next element is empty */
        {
            elem->nl = init_ttcn_el();
            return(elem->nl);
        }
        else                                                    /* find last next possibility of next level */
        {
            tmp = elem->nl;
            while (tmp->np != NULL)
                tmp = tmp->np;
            tmp->np = init_ttcn_el();
            return(tmp->np);
        }
    }
}

/*****************************************************************************
 *
 * Function     : intern_fill
 *
 * Description  :
 *          This function fills nelem with info about the internal event state 'state'.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be used for updating nelem
 *                          *nelem  : pointer to ttcn element to be filled
 *          int                 state   : internal number of internal event state
 *                          sort    : is 1 if state has to be searched else 0
 *                          first   : is 1 if nelem is first element
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
intern_fill(files, elem, nelem, state, sort, first)
files_type      *files;
ttcn_el_type    *elem,
                *nelem;
int                 state,
                sort,
                first;
{
    char    inbuf[200],
            line[200];
    long    pos_state;

    if (sort == 1)                                                  /* state has to be searched */
    {
        pos_state = ftell(files->state2fp);

        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);
        fgets(inbuf, 199, files->state2fp);
    }

    strcpy(nelem->action, "I");

    fgets(inbuf, 199, files->state2fp);                             /* @ACT-line */
    inbuf[strlen(inbuf)-1] = '\0';                                  /* remove '\n' */
    strcpy(inbuf, &inbuf[5]);                                       /* remove @ACT */
    sprintf(nelem->comment, "Internal Event : %s", inbuf);

    if (first == 0)
        nelem->indent = elem->indent + 1;

    nelem->state = state;

    if (sort == 1)
        fseek(files->state2fp, pos_state, 0);
}

/*****************************************************************************
 *
 * Function     : output_fill
 *
 * Description  :
 *          This function fills nelem with info about the output state 'state'.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be used for updating nelem
 *                          *nelem  : pointer to ttcn element to be filled
 *          int                 state   : internal number of outputstate
 *                          sort    : is 1 if state has to be searched, 2 if output is inconclusive sidepath, else 0
 *                          first   : is 1 if nelem is first element
 *
 * Returns      : void
```

```
 *
 ******************************************************************************/

static  void
output_fill(files, elem, nelem, state, sort, first)
files_type      *files;
ttcn_el_type    *elem,
                *nelem;
int                 state,
                sort,
                first;
{
    char    inbuf[200],
            line[200];
    long    pos_state;

    if (sort == 1)                                                  /* state has to be searched */
    {
        pos_state = ftell(files->state2fp);

        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);
        fgets(inbuf, 199, files->state2fp);
    }

    fgets(inbuf, 199, files->state2fp);                             /* @ACT-line */
    inbuf[strlen(inbuf)-1] = '\0';                                  /* remove '\n' */
    strcpy(nelem->action, &inbuf[5]);                               /* remove @ACT */
    inv_inout(nelem->action);                                       /* inverse ?! */

    if (first == 0)
        nelem->indent = elem->indent + 1;

    nelem->state = state;

    if (sort == 2)                          /* output is last element of possible sidepath from unstable state */
    {
        sprintf(line, "Possible sidepath to output");
        strcpy(nelem->comment, line);
        nelem->verdict = inconclusive;
    }

    if (sort == 1)
        fseek(files->state2fp, pos_state, 0);
}

/******************************************************************************
 *
 * Function     : input_fill
 *
 * Description  :
 *          This function fills nelem with info about the input state 'state'.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be used for updating nelem
 *                          *nelem  : pointer to ttcn element to be filled
 *          int                 state   : internal number of inputstate
 *                          sort    : is 1 if state has to be searched, else 0
 *                          first   : is 1 if nelem is first element
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
input_fill(files, elem, nelem, state, sort, first)
files_type      *files;
ttcn_el_type    *elem,
                *nelem;
int                 state,
                sort,
                first;
{
    char    inbuf[200],
            line[200];
    long    pos_state;

    if (sort == 1)                                                  /* state has to be searched */
    {
        pos_state = ftell(files->state2fp);

        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);
        fgets(inbuf, 199, files->state2fp);
    }

    fgets(inbuf, 199, files->state2fp);                             /* @ACT-line */
    inbuf[strlen(inbuf)-1] = '\0';                                  /* remove '\n' */
    strcpy(nelem->action, &inbuf[5]);                               /* remove @ACT */
    inv_inout(nelem->action);

    if (first == 0)
```

```
        nelem->indent = elem->indent + 1;

    nelem->state = state;

    if (sort == 1)
        fseek(files->state2fp, pos_state, 0);
}

/*****************************************************************************
 *
 * Function     : choice_fill
 *
 * Description  :
 *          This function fills nelem with info about the choice state 'state'.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be used for updating nelem
 *                          *nelem  : pointer to ttcn element to be filled
 *          int                 state   : internal number of choice state
 *                          sort    : is 1 if state has to be searched, else 0
 *                          first   : is 1 if nelem is first element
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
choice_fill(files, elem, nelem, state, sort, first)
files_type      *files;
ttcn_el_type    *elem,
                *nelem;
int                 state,
                sort,
                first;
{
    char    inbuf[200],
            line[200];
    long    pos_state;

    if (sort == 1)                                                  /* state has to be searched */
    {
        pos_state = ftell(files->state2fp);

        rewind(files->state2fp);
        sprintf(line, "@ST %d\n", state);
        find_line(line, files->state2fp);
        fgets(inbuf, 199, files->state2fp);
    }

    sprintf(nelem->action, "C");

    fgets(inbuf, 199, files->state2fp);                             /* @ACT-line */
    fgets(inbuf, 199, files->state2fp);                             /* @VAR-line */
    fgets(inbuf, 199, files->state2fp);                             /* choice variable-line */
    inbuf[strlen(inbuf)-1] = '\0';                                  /* remove '\n' */
    sprintf(nelem->comment, "Choicevar : %s", inbuf);

    if (first == 0)
        nelem->indent = elem->indent + 1;

    nelem->state = state;

    if (sort == 1)
        fseek(files->state2fp, pos_state, 0);
}

/*****************************************************************************
 *
 * Function     : strec_flow_fill
 *
 * Description  :
 *          This function finishes elem for a stable flowstate if a reception has taken place. It adds at the next
 *          level :
 *          START TIMER
 *              ?TIME OUT
 *                  +UIO sequence state 'state'
 *              ?OTHERWISE
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be filled
 *          int                 state   : internal number of flowstate to be checked
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
strec_flow_fill(files, elem, state)
files_type      *files;
ttcn_el_type    *elem;
int                 state;
{
    int         flowstate;
```

```
    char    line[200];

    flowstate = flownumber(state, files);

    elem->nl = init_ttcn_el();
    strcpy(elem->nl->action, "START TIMER");
    sprintf(line, "Checking if state F%d is stable", flowstate);
    strcpy(elem->nl->comment, line);
    elem->nl->indent = elem->indent + 1;

    elem->nl->nl = init_ttcn_el();
    strcpy(elem->nl->nl->action, "?TIME OUT");
    elem->nl->nl->indent = elem->indent + 2;

    elem->nl->nl->nl = init_ttcn_el();
    sprintf(line, "+UIO sequence state F%d", flowstate);
    strcpy(elem->nl->nl->nl->action, line);
    elem->nl->nl->nl->verdict = pass;
    elem->nl->nl->nl->indent = elem->indent + 3;

    elem->nl->nl->np = init_ttcn_el();
    strcpy(elem->nl->nl->np->action, "?OTHERWISE");
    elem->nl->nl->np->verdict = fail;
    elem->nl->nl->np->indent = elem->indent + 2;
}

/*****************************************************************************
 *
 * Function     : unstrec_flow_fill
 *
 * Description  :
 *          This function finishes elem for a unstable flowstate or if no reception has taken place. Only the
 *          UIO sequence is added at the next level.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be filled
 *          int                 state   : internal number of flowstate which UIO sequence has to be added
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
unstrec_flow_fill(files, elem, state)
files_type      *files;
ttcn_el_type    *elem;
int                 state;
{
    int         flowstate;
    char    line[200];

    elem->nl = init_ttcn_el();
    flowstate = flownumber(state, files);
    sprintf(line, "+UIO sequence state F%d", flowstate);
    strcpy(elem->nl->action, line);
    elem->nl->verdict = pass;
    elem->nl->indent = elem->indent + 1;
}

/*****************************************************************************
 *
 * Function     : step_cflow
 *
 * Description  :
 *          This function makes a next step in forming a control flow test case for an input. If the next state
 *          is no flowstate, an ttcn element will be filled with the appropiate info and added to elem.
 *          If a basic flowstate is reached, the element elem will be finished in a way depending on whether
 *          a stable flowstate is reached and a reception has taken place. If a not basic flowstate is reached,
 *          a next step will be called. Next steps will be called for all possible states departing from the
 *          present state.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *          int                 rec         : is 1 if reception has taken place before this step
 *                          state   : internal number of state on which step takes place
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
step_cflow(files, elem, rec, state)
files_type      *files;
ttcn_el_type    *elem;
int                 rec,
                state;
{
    long            pos_state,
                    pos_state2,
                    pos_trans;
    char            line[200],
                    word[50];
    int                 tostate,
```

```
                flowstate,
                state2;
    ttcn_el_type    *nelem;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);

    if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
    pos_state2 = ftell(files->state2fp);
    while (strcmp(word, "@ST") != 0)                                    /* for each outgoing trans */
    {
        tostate = find_st_line_next_state(files, word);
        fgets(line, 199, files->state2fp);                              /* F/X/O/I/C X-line */

        switch(line[0])
        {
        case 'X':
                nelem = create_next_elem(elem, 0);
                intern_fill(files, elem, nelem, tostate, 0, 0);
                step_cflow(files, nelem, rec, tostate);
                break;
        case 'O':
                nelem = create_next_elem(elem, 0);
                output_fill(files, elem, nelem, tostate, 0, 0);
                step_cflow(files, nelem, 1, tostate);
                break;
        case 'I':
                nelem = create_next_elem(elem, 0);
                input_fill(files, elem, nelem, tostate, 0, 0);
                step_cflow(files, nelem, rec, tostate);
                break;
        case 'C':
                nelem = create_next_elem(elem, 0);
                choice_fill(files, elem, nelem, tostate, 0, 0);
                step_cflow(files, nelem, rec, tostate);
                break;
        case 'F':
                if (is_basic(tostate, files))                           /* basic: finish sequence */
                {
                    if ((is_stable(tostate, files)) && (rec == 1))
                        strec_flow_fill(files, elem, tostate);
                    else
                        unstrec_flow_fill(files, elem, tostate);
                }
                else                                                    /* not basic: next step */
                    step_cflow(files, elem, rec, tostate);

                break;
        }

        fseek(files->state2fp, pos_state2, 0);
        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
        pos_state2 = ftell(files->state2fp);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : add_otherw
 *
 * Description  :
 *          This function adds '?OTHERWISE   FAIL' and 'EXPIRATION OF TIMER     FAIL' as a next possibility
 *          for levels where an output from the IUT is expected.
 *
 * Arguments    :
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
add_otherw(elem)
ttcn_el_type    *elem;
{
    ttcn_el_type    *tmp;

    if ((strstr(elem->action, "?") != NULL) && (strcmp(elem->action, "?TIME OUT") != 0))
    {                                                                   /* if output action on this level */
        tmp = elem;
        while (tmp->np != NULL)
        {
            tmp = tmp->np;                                              /* find last possibility */
        }

        if ((strcmp(tmp->action, "?EXPIRATION OF TIMER") != 0) && (strcmp(tmp->action, "?OTHERWISE") != 0))
```

```
        {                                                                   /* not already added */
            tmp->np = init_ttcn_el();
            strcpy(tmp->np->action, "?OTHERWISE");
            tmp->np->indent = tmp->indent;
            tmp->np->verdict = fail;

            tmp->np->np = init_ttcn_el();
            strcpy(tmp->np->np->action, "?EXPIRATION OF TIMER");
            tmp->np->np->indent = tmp->indent;
            tmp->np->np->verdict = fail;
        }
    }

    if (elem->nl != NULL)                                                   /* traverse depth first */
    {
        add_otherw(elem->nl);
    }

    if (elem->np != NULL)
    {
        add_otherw(elem->np);
    }
}

/*****************************************************************************
 *
 * Function     : write_head_cflow
 *
 * Description  :
 *          This function writes the test case header for a control flow test case.
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          int                 flowstate   : number of starting flowstate (as in F X)
 *                          inputstate  : internal number of inputstate whose action is basic part of test case
 *          case_type       *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_head_cflow(files, flowstate, inputstate, cases)
files_type      *files;
int                 flowstate,
                inputstate;
case_type       *cases;
{
    long    pos_state,
            pos_trans;
    char    line[200],
            inbuf[200];

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    fprintf(files->cflowttcnfp, "_____________________________________________________________________________\n");
    fprintf(files->cflowttcnfp, "-----------------------------------------------------------------------------\n");
    fprintf(files->cflowttcnfp, "                          Test Case Control Flow\n");
    fprintf(files->cflowttcnfp, "-----------------------------------------------------------------------------\n");

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", inputstate);
    find_line(line, files->state2fp);
    fgets(inbuf, 199, files->state2fp);                                 /* I X-line */
    fgets(inbuf, 199, files->state2fp);                                 /* @ACT-line */
    inbuf[strlen(inbuf)-1] = '\0';                                      /* remove '\n' */
    strcpy(inbuf, &inbuf[5]);
    fprintf(files->cflowttcnfp, "Purpose:    Testing input '%s' from flowstate F%d\n", inbuf, flowstate);

    fprintf(files->cflowttcnfp, "Identifier: CFL_%d\n", ++(cases->num_cflow));
    fprintf(files->cflowttcnfp, "-----------------------------------------------------------------------------\n");

    fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : find_maxlen
 *
 * Description  :
 *          This function finds the maximal length necessary to write an action with indentations on a test case
 *          body line.
 *
 * Arguments    :
 *          ttcn_el_type    *elem       : pointer to ttcn element to be traversed
 *          int                 *maxlength  : maximal length of action part + indentations
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
find_maxlen(elem, maxlength)
```

```
ttcn_el_type    *elem;
int                 *maxlength;
{
    int         total_max1,
            total_max2,
            local_max;

    total_max1 = 0;
    total_max2 = 0;

    local_max = strlen(elem->action) + (4 * elem->indent);                          /* calculate local length */

    if (elem->nl != NULL)                                                           /* traverse depth first */
    {
        find_maxlen(elem->nl, &total_max1);
    }

    if (elem->np != NULL)
    {
        find_maxlen(elem->np, &total_max2);
    }

    if ((total_max1 >= local_max) && (total_max1 >= total_max2))
        *maxlength = total_max1;
    if ((total_max2 >= local_max) && (total_max2 >= total_max1))
        *maxlength = total_max2;
    if ((local_max >= total_max1) && (local_max >= total_max2))
        *maxlength = local_max;
}

/*****************************************************************************
 *
 * Function     : write_body_line
 *
 * Description  :
 *          This function writes a line, e.g. a next ttcn element, for the body of a test case to a file with
 *          file pointer fp. It contains the line number, the indented action and the verdict. This function
 *          writes all lines recursively.
 *
 * Arguments    :
 *          FILE            *fp             : pointer to write file
 *          ttcn_el_type    *elem       : pointer to element to be traversed
 *          int                 maxlength   : maximal length of action part + indentations
 *                          *line       : line number
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_body_line(fp, elem, maxlength, line)
FILE            *fp;
ttcn_el_type    *elem;
int                 maxlength,
                *line;
{
    int             spacelength,
                spaces,
                i;

    fprintf(fp, " %4d |    ", (*line)++);                                         /* print line number */
    for (i=0; i < elem->indent; i++) fprintf(fp, "    ");
    fprintf(fp, "%s", elem->action);
    if (maxlength < 21)                                 /* minimal necessary length of line (Behaviour description) */
        spacelength = 21;
    else
        spacelength = maxlength;
    spaces = spacelength - (4 * elem->indent) - strlen(elem->action);
    for (i=0; i < spaces; i++) fprintf(fp, " ");                                    /* add until maxlength */
    fprintf(fp, "  |  ");

    switch(elem->verdict)
    {
    case pass:
            fprintf(fp, "PASS\n");
            break;
    case fail:
            fprintf(fp, "FAIL\n");
            break;
    case inconclusive:
            fprintf(fp, "INCONCLUSIVE\n");
            break;
    case none:
            fprintf(fp, "\n");
            break;
    default:
            err(ttcn, 37, 0);
    }

    if (elem->nl != NULL)                                                           /* traverse depth first */
    {
        write_body_line(fp, elem->nl, maxlength, line);
    }

    if (elem->np != NULL)
```

```
    {
        write_body_line(fp, elem->np, maxlength, line);
    }
}

/*****************************************************************************
 *
 * Function     : write_body
 *
 * Description  :
 *          This function writes the body of a test case, e.g. the first two standard lines and it calls the
 *          recursive function write_body_line.
 *
 * Arguments    :
 *          FILE            *fp         : pointer to write file
 *          ttcn_el_type    *elem   : pointer to ttcn element to be traversed
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_body(fp, elem)
FILE            *fp;
ttcn_el_type    *elem;
{
    int         maxlength,
            div1,
            div2,
            i,
            line = 1;

    find_maxlen(elem, &maxlength);

    div1 = maxlength / 2;
    div2 = div1 + (maxlength % 2);
    fprintf(fp, " Line |    ");
    for (i=0; i < div1-10; i++) fprintf(fp, " ");
    fprintf(fp, "Behaviour Description");
    for (i=0; i < div2-11; i++) fprintf(fp, " ");
    fprintf(fp, "  |  Verdict\n");

    fprintf(fp, "      |    ");
    if (maxlength > 21)
        for (i=0; i < maxlength; i++) fprintf(fp, " ");
    else
        for (i=0; i < 21; i++) fprintf(fp, " ");
    fprintf(fp, "  |\n");

    write_body_line(fp, elem, maxlength, &line);
}

/*****************************************************************************
 *
 * Function     : write_comm_line
 *
 * Description  :
 *          This function writes recursively comment lines to fp.
 *
 * Arguments    :
 *          FILE            *fp         : pointer to write file
 *          ttcn_el_type    *elem   : pointer to ttcn element to be traversed
 *          int                 *line   : line number of element
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_comm_line(fp, elem, line)
FILE            *fp;
ttcn_el_type    *elem;
int                 *line;
{
    if (strcmp(elem->comment, "") != 0)
    {
        fprintf(fp, "Line %3d : %s\n", *line, elem->comment);
    }

    (*line)++;

    if (elem->nl != NULL)                                   /* write comment, depth first */
    {
        write_comm_line(fp, elem->nl, line);
    }

    if (elem->np != NULL)
    {
        write_comm_line(fp, elem->np, line);
    }
}

/*****************************************************************************
 *
 * Function     : write_comm
```

```
 *
 * Description  :
 *          This function writes the comment part of a test case to fp by writing the first to standard lines
 *          and calling the recursive function write_comm_line.
 *
 * Arguments    :
 *          FILE            *fp         : pointer to write file
 *          ttcn_el_type    *elem   : pointer to ttcn element to be traversed
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
write_comm(fp, elem)
FILE            *fp;
ttcn_el_type    *elem;
{
    int         line = 1;

    fprintf(fp, "-------------------------------------------------------------------------------\n");
    fprintf(fp, "COMMENTS:\n");

    write_comm_line(fp, elem, &line);

    fprintf(fp, "_____________________________________________________________________________\n\n\n");
}

/*****************************************************************************
 *
 * Function     : free_total
 *
 * Description  :
 *          This function frees all elements of a ttcn test case.
 *
 * Arguments    :
 *          ttcn_el_type    *elem   : pointer of ttcn element to be freed
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
free_total(elem)
ttcn_el_type    *elem;
{
    if (elem->nl != NULL)                                       /* traverse depth first, free all */
    {
        free_total(elem->nl);
    }

    if (elem->np != NULL)
    {
        free_total(elem->np);
    }

    free(elem);
}

/*****************************************************************************
 *
 * Function     : cflow_case
 *
 * Description  :
 *          This function makes a control flow ttcn test case by filling the first element with the input state,
 *          calling the recursive function step_cflow to make all elements, adding otherwise and exp. of time and
 *          calling the functions to write the test case to the appropiate file.
 *
 * Arguments    :
 *          int             flowstate   : number of flowstate
 *                      inputstate  : internal number of input state
 *          files_type  *files      : pointer to all files
 *          case_type   *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
cflow_case(flowstate, inputstate, files, cases)
int             flowstate,
            inputstate;
files_type  *files;
case_type   *cases;
{
    ttcn_el_type    *elem;

    elem = init_ttcn_el();

    fill_first_cflow(files, elem, flowstate, inputstate);

    step_cflow(files, elem, 0, elem->state);

    add_otherw(elem);
```

```
    write_head_cflow(files, flowstate, inputstate, cases);
    write_body(files->cflowttcnfp, elem);
    write_comm(files->cflowttcnfp, elem);

    free_total(elem);
}

/*****************************************************************************
 *
 * Function     : ttcn_cflow
 *
 * Description  :
 *          This function calls the function cflow_case for all inputs.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          efg_type    *efg    : pointer to efg
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
ttcn_cflow(files, efg)
files_type  *files;
efg_type    *efg;
{
    long        pos;
    int             flowstate,
                tostate,
                state;
    char        line[80],
                word[50];
    case_type   *cases;

    cases = (case_type *) calloc(1, sizeof(case_type));

    printf("\nTransformation of Control Flow subsequences to TTCN");

    for (flowstate=1; flowstate != efg->num_flow+1; flowstate++)
    {
        state = statenumber(flowstate, files);
        if (is_basic(state, files))                                     /* for all basic flowstates */
        {
            rewind(files->state2fp);
            sprintf(line, "@ST %d\n", state);
            find_line(line, files->state2fp);
            sprintf(line, "@OUT\n");
            find_line(line, files->state2fp);

            if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
            pos = ftell(files->state2fp);
            while (strcmp(word, "@ST") != 0)                            /* for each outgoing trans */
            {
                tostate = find_st_line_next_state(files, word);
                if(is_inputstate(tostate, files))                       /* if input from flowstate */
                {
                    cflow_case(flowstate, tostate, files, cases);
                }
                fseek(files->state2fp, pos, 0);
                if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
                pos = ftell(files->state2fp);
            }
        }
    }

    free(cases);
}

/*****************************************************************************
 *
 * Function     : num_next_state
 *
 * Description  :
 *          This function tries to find the next state in seq. It will return in 'word' the type of the next state,
 *          e.g. F or X or I or O or C and will return the internal number of this next state.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *          char        *seq    : sequence string
 *                      *word   : first letter of next state in seq
 *
 * Returns      : int   :
 *          intern number of first state in seq
 *
 *****************************************************************************/

static  int
num_next_state(files, seq, word)
files_type  *files;
char        *seq,
            *word;
{
    long    pos_state;
    char    line[80],
```

```
            inbuf[80],
            inbuf_old[80],
            word1[20],
            word2[20];

    pos_state = ftell(files->state2fp);

    if (strcmp(seq, "") == 0) return(0);                                    /* all states removed */

    *word = seq[0];
    line[0] = seq[0];
    line[1] = ' ';
    line[2] = '\0';
    strcpy(seq, &seq[1]);                                                   /* remove F/X/O/I/C */
    sscanf(seq, "%s", word1);                                               /* scan number of state */
    if (strcmp(seq, word1) == 0)                                            /* last state */
        strcpy(seq, "");
    else
        strcpy(seq, &seq[strlen(word1) + 1]);
    sprintf(word2, "%s\n", word1);
    strcat(line, word2);

    rewind(files->state2fp);
    strcpy(inbuf, "");
    do
    {
        strcpy(inbuf_old, inbuf);
        fgets(inbuf, 79, files->state2fp);
    } while ((!feof(files->state2fp)) && (strcmp(inbuf, line) != 0));       /* find 'F/O/I/X/C state' */
    if (feof(files->state2fp))
    {
        printf("\n%s not found\n", line);
        err(ttcn, 30, 0);
    }

    sscanf(inbuf_old, "%s%s", word1, word2);                            /* scan internal number state */

    fseek(files->state2fp, pos_state, 0);

    return(atoi(word2));
}

/*****************************************************************************
 *
 * Function     : stab_flow_fill
 *
 * Description  :
 *          This function will add the element '?EXPIRATION OF TIMER          INCONCLUSIVE' to elem. This takes
 *          place at a stable flowstate in possible sidepaths from unstable flowstates.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *          int                 state   : internal number of flowstate to be checked
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
stab_flow_fill(files, elem, state)
files_type      *files;
ttcn_el_type    *elem;
int                 state;
{
    char    line[200];
    int         flowstate;

    elem->nl = init_ttcn_el();
    strcpy(elem->nl->action, "?EXPIRATION OF TIMER");
    flowstate = flownumber(state, files);
    sprintf(line, "Checking stability of possible sidepath to stable flowstate F%d", flowstate);
    strcpy(elem->nl->comment, line);
    elem->nl->indent = elem->indent + 1;
    elem->nl->verdict = inconclusive;
    elem->nl->state = state;
}

/*****************************************************************************
 *
 * Function     : step_uio_flow
 *
 * Description  :
 *          This function makes a next step in forming a UIO sequence test case for all outgoing states except
 *          the intended one. The construction will be stopped at reaching an input, finished at reaching an
 *          output or stable flowstate and continued in all other cases. This function takes place recursively
 *          for not intended sidepaths from unstable flowstates of the UIO sequence.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *          char            *seq2   : state sequence string
 *          int                 state   : internal number of state to be expanded in this step
 *                          first   : is 1 if elem is first element, else 0
 *
```

```
 * Returns      : void
 *
 ******************************************************************************/

static  void
step_uio_flow(files, elem, seq2, state, first)
files_type      *files;
ttcn_el_type    *elem;
char            *seq2;
int                 state,
                first;
{
    long            pos_state,
                    pos_state2,
                    pos_trans;
    char            line[200],
                    word[50],
                    word1[10],
                    word2[10];
    int                 tostate;
    ttcn_el_type    *nelem;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);

    if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
    pos_state2 = ftell(files->state2fp);
    while (strcmp(word, "@ST") != 0)                                    /* for each outgoing trans */
    {
        tostate = find_st_line_next_state(files, word);
        fgets(line, 199, files->state2fp);                              /* F X-line */
        sscanf(line, "%s %s", word1, word2);
        strcat(word1, word2);

        if ((strstr(seq2, word1) != seq2) && (strcmp(seq2, "") != 0))
        {                                                               /* if not PASS-path */
            switch(line[0])
            {
            case 'X':
                    nelem = create_next_elem(elem, first);
                    intern_fill(files, elem, nelem, tostate, 0, first);
                    step_uio_flow(files, nelem, seq2, tostate, 0);
                    break;
            case 'O':
                    nelem = create_next_elem(elem, first);
                    output_fill(files, elem, nelem, tostate, 2, first);     /* finish at output */
                    break;
            case 'I':
                    break;                                              /* wrong path */
            case 'C':
                    nelem = create_next_elem(elem, first);
                    choice_fill(files, elem, nelem, tostate, 0, first);
                    step_uio_flow(files, nelem, seq2, tostate, 0);
                    break;
            case 'F':
                    if (!is_stable(tostate, files))                     /* not stable: next step */
                    {
                        step_uio_flow(files, elem, seq2, tostate, first);
                    }
                    else                                                /* stable : finish */
                    {
                        stab_flow_fill(files, elem, tostate);
                    }
                    break;
            }
        }
        fseek(files->state2fp, pos_state2, 0);
        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
        pos_state2 = ftell(files->state2fp);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/******************************************************************************
 *
 * Function     : step_uio
 *
 * Description  :
 *          This function takes a next step in forming a test case for an UIO sequence. It takes the next state
 *          of the sequence and fills a new element with the appropiate info. For instable flowstates it calls
 *          the function step_uio_flow to form all possible inconclusive sidepaths.
 *
 * Arguments    :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated in this step
 *          char            *seq    : state sequence string
 *          int                 state   : internal number of state to be expanded in this step
```

```
 *                          first   : is 1 if elem is first element, else 0
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
step_uio(files, elem, seq, state, first)
files_type      *files;
ttcn_el_type    *elem;
char            *seq;
int             state,
                first;
{
    long            pos_state,
                    pos_trans;
    char            word,
                    *seq2;
    int             tostate;
    ttcn_el_type    *nelem;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    seq2 = (char *) malloc((strlen(seq)+1) * sizeof(char));

    tostate = num_next_state(files, seq, &word);                                    /* get next state of sequence */

    if (tostate != 0)
    {
        switch(word)
        {
        case 'X':
                nelem = create_next_elem(elem, first);
                intern_fill(files, elem, nelem, tostate, 1, first);
                step_uio(files, nelem, seq, tostate, 0);
                break;
        case 'O':
                nelem = create_next_elem(elem, 0);
                output_fill(files, elem, nelem, tostate, 1, first);
                step_uio(files, nelem, seq, tostate, 0);
                break;
        case 'I':
                nelem = create_next_elem(elem, 0);
                input_fill(files, elem, nelem, tostate, 1, first);
                step_uio(files, nelem, seq, tostate, 0);
                break;
        case 'C':
                nelem = create_next_elem(elem, 0);
                choice_fill(files, elem, nelem, tostate, 1, first);
                step_uio(files, nelem, seq, tostate, 0);
                break;
        case 'F':
                if (is_stable(tostate, files))
                    step_uio(files, elem, seq, tostate, first);
                else
                {
                    strcpy(seq2, seq);
                    step_uio(files, elem, seq, tostate, first);
                    step_uio_flow(files, elem, seq2, tostate, first);       /* not stable : form possible sidepaths */
                }
                break;
        }
    }

  . fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/******************************************************************************
 *
 * Function     : add_pass
 *
 * Description  :
 *          This function adds the verdict PASS at the last level of a ttcn element if there is not yet an
 *          other verdict.
 *
 * Arguments    :
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
add_pass(elem)
ttcn_el_type    *elem;
{
    ttcn_el_type    *tmp;

    if (elem->nl != NULL)                                       /* traverse depth first */
    {
        add_pass(elem->nl);
    }
    else
```

```
    {
        if (elem->verdict == none)
            elem->verdict = pass;                                   /* add PASS at last level if no verdict */
    }

    if (elem->np != NULL)
    {
        add_pass(elem->np);
    }
}

/*****************************************************************************
 *
 * Function     : write_head_uio
 *
 * Description  :
 *          This function writes the header for a UIO sequence test case to the appropiate file.
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          int                 flowstate   : internal number of flowstate to be checked for uniqueness
 *          case_type       *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_head_uio(files, flowstate, cases)
files_type      *files;
int                 flowstate;
case_type       *cases;
{
    int         state;

    state = flownumber(flowstate, files);

    fprintf(files->uiottcnfp, "_____________________________________________________________________________\n");
    fprintf(files->uiottcnfp, "-----------------------------------------------------------------------------\n");
    fprintf(files->uiottcnfp, "                          Test Case Part UIO Sequence\n");
    fprintf(files->uiottcnfp, "-----------------------------------------------------------------------------\n");

    fprintf(files->uiottcnfp, "Purpose:    Testing uniqueness flowstate F%d\n", state);

    fprintf(files->uiottcnfp, "Identifier: UIO_%d\n", ++(cases->num_uio));
    fprintf(files->uiottcnfp, "-----------------------------------------------------------------------------\n");
}

/*****************************************************************************
 *
 * Function     : uio_case
 *
 * Description  :
 *          This function makes a UIO sequence test case by calling the recursive function step_uio to
 *          make all elements, adding otherwise and exp. of time, adding verdicts PASS where necessary and calling
 *          the functions to write the test case to the appropiate file.
 *
 * Arguments    :
 *          int             flowstate   : number of flowstate to be checked for uniqueness
 *          char        *seq        : state sequence string
 *          files_type  *files      : pointer to all files
 *          case_type   *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
uio_case(flowstate, seq, files, cases)
int             flowstate;
char        *seq;
files_type  *files;
case_type   *cases;
{
    ttcn_el_type    *elem;

    elem = init_ttcn_el();

    step_uio(files, elem, seq, flowstate, 1);

    add_otherw(elem);

    add_pass(elem);

    write_head_uio(files, flowstate, cases);
    write_body(files->uiottcnfp, elem);
    write_comm(files->uiottcnfp, elem);

    free_total(elem);
}

/*****************************************************************************
 *
 * Function     : ttcn_uio
 *
```

```
 * Description   :
 *          This function calls the function uio_case for the first UIO sequence of all basic states containing
 *          such a sequence.
 *
 * Arguments     :
 *          files_type  *files   : pointer to all files
 *
 * Returns       : void
 *
 ******************************************************************************/

static  void
ttcn_uio(files)
files_type  *files;
{
    char        line[3000];
    int             c,
                flowstate;
    case_type   *cases;

    cases = (case_type *) calloc(1, sizeof(case_type));

    printf("\nTransformation of UIO sequences to TTCN");

    rewind(files->state2fp);
    fgets(line, 2999, files->uiofp);
    while (!feof(files->uiofp))
    {
        c = getc(files->uiofp);                                         /* 'F' */
        fscanf(files->uiofp, "%d", &flowstate);
        flowstate = statenumber(flowstate, files);
        fgets(line, 2999, files->uiofp);                                /* rest line */
        fgets(line, 2999, files->uiofp);                                /* empty line */
        fgets(line, 2999, files->uiofp);                                /* sequence line or striped line */
        line[strlen(line)-1] = '\0';                                    /* remove '\n' */

        if (strcmp(line, "NO UIO") != 0)
            uio_case(flowstate, line, files, cases);

        while ((line[0] != '_') && (!feof(files->uiofp)))               /* goto next flowstate */
        {
            /* uio_case(flowstate, line, files); here if uiosequences for all basic flowstates necessary */

            fgets(line, 2999, files->uiofp);                            /* action line */
            fgets(line, 2999, files->uiofp);                            /* empty line */
            fgets(line, 2999, files->uiofp);                            /* sequence line or striped line */
        }
    }
    clearerr(files->uiofp);
    free(cases);
}

/******************************************************************************
 *
 * Function      : step_dflow_flow
 *
 * Description   :
 *          This function makes a next step in forming a data flow test case for all outgoing states except
 *          the intended one. The construction will be stopped at reaching an input, finished at reaching an
 *          output or stable flowstate and continued in all other cases. This function takes place recursively
 *          for not intended sidepaths from unstable flowstates of a data flow chain.
 *
 * Arguments     :
 *          files_type      *files  : pointer to all files
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *          char            *seq2   : state sequence string
 *          int                 state   : internal number of state to be expanded in this step
 *                          first   : is 1 if elem is first element, else 0
 *
 * Returns       : void
 *
 ******************************************************************************/

static  void
step_dflow_flow(files, elem, seq2, state, first)
files_type      *files;
ttcn_el_type    *elem;
char            *seq2;
int                 state,
                first;
{
    long            pos_state,
                    pos_state2,
                    pos_trans;
    char            line[200],
                    word[50],
                    word1[10],
                    word2[10];
    int                 tostate;
    ttcn_el_type    *nelem;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    rewind(files->state2fp);
```

```
    sprintf(line, "@ST %d\n", state);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);

    if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
    pos_state2 = ftell(files->state2fp);
    while (strcmp(word, "@ST") != 0)                                    /* for each outgoing trans */
    {
        tostate = find_st_line_next_state(files, word);
        fgets(line, 199, files->state2fp);                              /* F X-line */
        sscanf(line, "%s %s", word1, word2);
        strcat(word1, word2);

        if ((strstr(seq2, word1) != seq2) && (strcmp(seq2, "") != 0))
        {                                                               /* if not PASS-path */
            switch(line[0])
            {
            case 'X':
                    nelem = create_next_elem(elem, first);
                    intern_fill(files, elem, nelem, tostate, 0, first);
                    step_dflow_flow(files, nelem, seq2, tostate, 0);
                    break;
            case 'O':
                    nelem = create_next_elem(elem, first);
                    output_fill(files, elem, nelem, tostate, 2, first);  /* stop at output */
                    break;
            case 'I':
                    break;                                              /* wrong path */
            case 'C':
                    nelem = create_next_elem(elem, first);
                    choice_fill(files, elem, nelem, tostate, 0, first);
                    step_dflow_flow(files, nelem, seq2, tostate, 0);
                    break;
            case 'F':
                    if (!is_stable(tostate, files))                     /* not stable : next step */
                    {
                        step_dflow_flow(files, elem, seq2, tostate, first);
                    }
                    else                                                /* stable : finish */
                    {
                        stab_flow_fill(files, elem, tostate);
                    }
                    break;
            }
        }
        fseek(files->state2fp, pos_state2, 0);
        if (fscanf(files->state2fp, "%s", word) == EOF) strcpy(word, "@ST");
        pos_state2 = ftell(files->state2fp);
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : write_call
 *
 * Description  :
 *          This function writes comment about a defined variable in a flowstate to elem->comment.
 *
 * Arguments    :
 *          ttcn_el_type    *elem   : pointer to ttcn element to be updated
 *          char            *inbuf  : string containing call variables
 *                          *variab     : influenced variable
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
write_call(elem, inbuf, variab)
ttcn_el_type    *elem;
char            *inbuf,
                *variab;
{
    char    line[80];

    if (strstr(inbuf, "@CONST") == NULL)                    /* variable defined by call with other variables */
    {
        sprintf(line, "          '%s' influenced by variable(s)", variab);
        strcat(elem->comment, line);

        while (sscanf(inbuf, "%s", variab) != EOF)                      /* for every used variable */
        {
            sprintf(line, " '%s',", variab);
            strcat(elem->comment, line);
            strcpy(inbuf, &inbuf[strlen(variab)+1]);
        }
        elem->comment[strlen(elem->comment)] = '\0';
        elem->comment[strlen(elem->comment)-1] = '\n';
    }
    else
    {
```

```
        sprintf(line, "          '%s' formed by constant call\n", variab);
        strcat(elem->comment, line);
    }
}

/*****************************************************************************
 *
 * Function     : add_calls_comm
 *
 * Description  :
 *          This function adds comment about calls to flowstates to the comment part of a data flow test case.
 *          This is done by calling the function add_calls for all defined variables in a flowstate.
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          ttcn_el_type    *elem       : pointer to ttcn element to be updated
 *          int                 from_state  : state at which call starts
 *                          state       : state at which call ends
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
add_calls_comm(files, elem, from_state, state)
files_type      *files;
ttcn_el_type    *elem;
int                 from_state,
                state;
{
    long    pos_state,
            pos_trans,
            pos_state2,
            pos_trans2;
    char    line[80],
            new_variab[30],
            word[40],
            inbuf[200];
    int         trans,
            flowstate;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    if (from_state == 0) return;

    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", from_state);
    find_line(line, files->state2fp);
    sprintf(line, "@OUT\n");
    find_line(line, files->state2fp);
    if (fscanf(files->state2fp, "%s", word) == EOF) err(ttcn, 29, 0);
    trans = atoi(word);
    rewind(files->trans1fp);
    sprintf(line, "@TR %d\n", trans);                                   /* go to outgoing trans of fromstate */
    find_line(line, files->trans1fp);
    fgets(inbuf, 199, files->trans1fp);                                 /* @FROM-line */
    fgets(inbuf, 199, files->trans1fp);                                 /* @TO-line */
    fgets(inbuf, 199, files->trans1fp);                                 /* @VAR-line */
    rewind(files->state2fp);
    sprintf(line, "@ST %d\n", state);                                   /* go to (to)state */
    find_line(line, files->state2fp);
    fgets(inbuf, 199, files->state2fp);                                 /* F X-line */
    fgets(inbuf, 199, files->state2fp);                                 /* @ACT-line */
    fgets(inbuf, 199, files->state2fp);                                 /* @VAR-line */

    if (strcmp(inbuf, "@VAR d\n") == 0)
    {
        flowstate = flownumber(state, files);
        if (strlen(elem->comment) > 0)
            strcat(elem->comment, "\n          ");
        sprintf(line, "Call to flowstate %d in which the following variables are defined:\n", flowstate);
        strcat(elem->comment, line);

        fgets(inbuf, 199, files->trans1fp);
        fscanf(files->state2fp, "%s", new_variab);                      /* defined variable in flowstate */
        pos_state2 = ftell(files->state2fp);
        pos_trans2 = ftell(files->trans1fp);
        while (strcmp(inbuf, "\n") != 0)                            /* for all defined variable in flowstate */
        {
            write_call(elem, inbuf, new_variab);

            fseek(files->state2fp, pos_state2, 0);
            fseek(files->trans1fp, pos_trans2, 0);
            fgets(inbuf, 199, files->trans1fp);
            fscanf(files->state2fp, "%s", new_variab);
            pos_state2 = ftell(files->state2fp);
            pos_trans2 = ftell(files->trans1fp);
        }

        elem->comment[strlen(elem->comment)-1] = '\0';
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->trans1fp, pos_trans, 0);
```

```
}

/*****************************************************************************
 *
 * Function     : step_dflow
 *
 * Description  :
 *          This function takes a next step in forming a test case for an data flow chain. It takes the next state
 *          of the sequence and fills a new element with the appropiate info. For instable flowstates it calls
 *          the function step_dflow_flow to form all possible inconclusive sidepaths.
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          ttcn_el_type    *elem       : pointer to ttcn element to be updated in this step
 *          char            *seq        : state sequence string
 *          int                 state       : number of previous state
 *                          first       : is 1 if elem is first element, else 0
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
step_dflow(files, elem, seq, state, first)
files_type      *files;
ttcn_el_type    *elem;
char            *seq;
int                 state,
                first;
{
    long            pos_state,
                    pos_trans;
    char            word,
                    *seq2;
    int                 tostate;
    ttcn_el_type    *nelem;

    pos_state = ftell(files->state2fp);
    pos_trans = ftell(files->trans1fp);

    seq2 = (char *) malloc((strlen(seq)+1) * sizeof(char));

    tostate = num_next_state(files, seq, &word);                    /* get internal number next state of sequence */

    if (tostate != 0)
    {
        switch(word)
        {
        case 'X':
                nelem = create_next_elem(elem, first);
                intern_fill(files, elem, nelem, tostate, 1, first);
                step_dflow(files, nelem, seq, tostate, 0);
                break;
        case 'O':
                nelem = create_next_elem(elem, 0);
                output_fill(files, elem, nelem, tostate, 1, first);
                step_dflow(files, nelem, seq, tostate, 0);
                break;
        case 'I':
                nelem = create_next_elem(elem, 0);
                input_fill(files, elem, nelem, tostate, 1, first);
                step_dflow(files, nelem, seq, tostate, 0);
                break;
        case 'C':
                nelem = create_next_elem(elem, 0);
                choice_fill(files, elem, nelem, tostate, 1, first);
                step_dflow(files, nelem, seq, tostate, 0);
                break;
        case 'F':
                if (first != 1)                                                 /* if previous element alraedy exists *
                    add_calls_comm(files, elem, state, tostate);

                if (is_stable(tostate, files))
                    step_dflow(files, elem, seq, tostate, first);
                else
                {
                    strcpy(seq2, seq);
                    step_dflow(files, elem, seq, tostate, first);
                    step_dflow_flow(files, elem, seq2, tostate, first);     /* not stable : make possible sidepaths */
                }
                break;
        }
    }

    fseek(files->state2fp, pos_state, 0);
    fseek(files->state2fp, pos_trans, 0);
}

/*****************************************************************************
 *
 * Function     : write_head_dflow
 *
 * Description  :
 *          This function writes the header part of a data flow test case to the appropiate file. The purpose
 *          part depends on the type of 'input' (variable input or constant call) and the type of 'output'
```

```
 *          (constant or variable).
 *
 * Arguments    :
 *          files_type      *files      : pointer to all files
 *          int                 flowstate   : internal number of flowstate at which case starts
 *                          type_in         : 1 if real input, 2 if constant call
 *          char            *type_out   : type of output variable
 *                          *start_var  : 'input' of data flow chain
 *                          *out_var    : 'output' of data flow chain
 *          case_type       *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
write_head_dflow(files, flowstate, type_in, type_out, start_var, out_var, cases)
files_type      *files;
int                 flowstate,
                type_in;
char            *type_out,
                *start_var,
                *out_var;
case_type       *cases;
{
    int         state;

    state = flownumber(flowstate, files);

    fprintf(files->dflowttcnfp, "_____________________________________________________________________________\n");
    fprintf(files->dflowttcnfp, "-----------------------------------------------------------------------------\n");
    fprintf(files->dflowttcnfp, "                          Test Case Data Flow\n");
    fprintf(files->dflowttcnfp, "-----------------------------------------------------------------------------\n");

    if ((type_in == 1) && (strcmp(type_out, "const_out") == 0))
    {
        fprintf(files->dflowttcnfp, "Purpose:    Testing Data Flow Chain from real input variable '%s'\n", start_var);
        fprintf(files->dflowttcnfp, "            to predicate variable '%s' (constant output)\n", out_var);
    }

    if ((type_in == 1) && (strcmp(type_out, "var_out") == 0))
    {
        fprintf(files->dflowttcnfp, "Purpose:    Testing Data Flow Chain from real input variable '%s'\n", start_var);
        fprintf(files->dflowttcnfp, "            to output variable '%s' (variable output)\n", out_var);
    }

    if ((type_in == 2) && (strcmp(type_out, "const_out") == 0))
    {
        fprintf(files->dflowttcnfp, "Purpose:    Testing Data Flow Chain from constant call variable '%s'\n", start_var);
        fprintf(files->dflowttcnfp, "            to predicate variable '%s' (constant output)\n", out_var);
    }

    if ((type_in == 2) && (strcmp(type_out, "var_out") == 0))
    {
        fprintf(files->dflowttcnfp, "Purpose:    Testing Data Flow Chain from constant call variable '%s'\n", start_var);
        fprintf(files->dflowttcnfp, "            to output variable '%s' (variable output)\n", out_var);
    }

    fprintf(files->dflowttcnfp, "            Start in flowstate F%d\n", state);

    fprintf(files->dflowttcnfp, "Identifier: DFL_%d\n", ++(cases->num_dflow));
    fprintf(files->dflowttcnfp, "-----------------------------------------------------------------------------\n");
}

/******************************************************************************
 *
 * Function     : dflow_case
 *
 * Description  :
 *          This function makes a data flow sequence test case by calling the recursive function step_dflow to
 *          make all elements, adding otherwise and exp. of time, adding verdicts PASS where necessary and calling
 *          the functions to write the test case to the appropiate file.
 *
 * Arguments    :
 *          int             flowstate   : internal number of flowstate at which case starts
 *          char        *seq        : state sequence string
 *          int             type_in         : 1 if real input, 2 if constant call
 *          char        *type_out   : type of output variable
 *                      *start_var  : 'input' of data flow chain
 *                      *out_var    : 'output' of data flow chain
 *          files_type  *files      : pointer to all files
 *          case_type   *cases      : pointer to structure counting number of test cases
 *
 * Returns      : void
 *
 ******************************************************************************/

static  void
dflow_case(flowstate, seq, type_in, type_out, start_var, out_var, files, cases)
int             flowstate;
char        *seq;
int             type_in;
char        *type_out,
            *start_var,
            *out_var;
```

```
files_type  *files;
case_type   *cases;
{
    ttcn_el_type    *elem;

    elem = init_ttcn_el();

    step_dflow(files, elem, seq, flowstate, 1);

    add_otherw(elem);

    add_pass(elem);

    write_head_dflow(files, flowstate, type_in, type_out, start_var, out_var, cases);
    write_body(files->dflowttcnfp, elem);
    write_comm(files->dflowttcnfp, elem);

    free_total(elem);
}

/*****************************************************************************
 *
 * Function     : ttcn_dflow
 *
 * Description  :
 *          This function calls the function dflow_case for all data flow chains in DFLOW_FILE.
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
ttcn_dflow(files)
files_type  *files;
{
    char    line[3000],
            type_out[15],
            start_var[50],
            out_var[50],
            word[10];
    int         flowstate,
            c;
    case_type   *cases;

    cases = (case_type *) calloc(1, sizeof(case_type));

    printf("\nTransformation of Data Flow subsequences to TTCN");

    fgets(line, 2999, files->dflowfp);                                  /* --------Real Input--------- line */
    fscanf(files->dflowfp, "%s", type_out);
    while (type_out[0] != '-')                                          /* for all real input lines */
    {
        fscanf(files->dflowfp, "%s", start_var);
        fscanf(files->dflowfp, "%s", out_var);
        c = getc(files->dflowfp);                                       /* space */
        fgets(line, 2999, files->dflowfp);
        line[strlen(line)-1] = '\0';                                    /* remove '\n' */

        sscanf(line, "%s", word);                                       /* 'F' and number flowstate */
        flowstate = atoi(&word[1]);
        flowstate = statenumber(flowstate, files);

        dflow_case(flowstate, line, 1, type_out, start_var, out_var, files, cases);

        fscanf(files->dflowfp, "%s", type_out);
    }

    fgets(line, 2999, files->dflowfp);                                  /* rest ---------Constant Call--------- line */
    fscanf(files->dflowfp, "%s", type_out);
    while (!feof(files->dflowfp))                                       /* for all constant call lines */
    {
        fscanf(files->dflowfp, "%s", start_var);
        fscanf(files->dflowfp, "%s", out_var);
        c = getc(files->dflowfp);                                       /* space */
        fgets(line, 2999, files->dflowfp);
        line[strlen(line)-1] = '\0';                                    /* remove '\n' */

        sscanf(line, "%s", word);                                       /* 'F' and number flowstate */
        flowstate = atoi(&word[1]);
        flowstate = statenumber(flowstate, files);

        dflow_case(flowstate, line, 2, type_out, start_var, out_var, files, cases);

        fscanf(files->dflowfp, "%s", type_out);
    }
    free(cases);
}

/*****************************************************************************
 *
 * Function     : ttcn_link
 *
```

```
 * Description   :
 *          This function should be the main function to transform link subsequences to TTCN.
 *
 * Arguments     :
 *          files_type  *files  : pointer to all files
 *
 * Returns       : void
 *
 ******************************************************************************/

static  void
ttcn_link(files)
files_type  *files;
{
    printf("\nTransformation of link subsequences to TTCN not yet implemented");
}

/******************************************************************************
 *
 * Function      : to_ttcn
 *
 * Description   :
 *          This function is the main function of this module. It asks if you want to transform control flow, data
 *          flow or link subsequences (or all) to TTCN. If you choose control flow, also UIO sequence parts
 *          will be automatically constructed.
 *
 * Arguments     :
 *          efg_type    *efg    : pointer to efg
 *          files_type  *files  : pointer to all files
 *
 * Returns       : void
 *
 ******************************************************************************/

extern  void
to_ttcn(efg, files)
efg_type    *efg;
files_type  *files;
{
    char    inbuf[80];
    int         menu;

    open_files_ttcn(files);

    printf("\nWhich sequences do you want to have transformed to TTCN?");
    printf("\n\n1       Control Flow, Data Flow and Link subsequences");
    printf("\n2     Only Control Flow subsequences");
    printf("\n3     Only Data Flow subsequences");
    printf("\n4     Only Link subsequences");
    printf("\n\n0       No transformation to TTCN\n");
    do
    {
        printf("\nYour choice? ");
        gets(inbuf);
        sscanf(inbuf, "%d", &menu);
    } while ((menu < 0) || (menu > 4));


    if ((menu == 1) || (menu == 2))
    {
        ttcn_cflow(files, efg);
        ttcn_uio(files);
    }

    if ((menu == 1) || (menu == 3))
        ttcn_dflow(files);

    if ((menu == 1) || (menu == 4))
        ttcn_link(files);

    close_files_ttcn(files);
}
```

```
/*****************************************************************************
 *
 * File             : linksubs.c
 *
 * Description  :
 *          This module ought to calculate link subsequences and write them to LINK_FILE, but is not yet
 *          implemented.
 *
 * Editor       : M.F. van Opstal
 *
 * Version      : 30-7-1993
 *
 *****************************************************************************/

/* INCLUDE FILES */

#include <stdio.h>
#include "typedef.h"
#include "efghead.h"

/* FUNCTION DEFINITIONS */

/*****************************************************************************
 *
 * Function     : open_files_links
 *
 * Description  :
 *              This function opens the necessary files for this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
open_files_links(files)
files_type  *files;
{
    if ((files->state2fp = fopen(STATE_FILE_2, "r")) == NULL)
        err(link_ss, 6, 0);
    if ((files->trans1fp = fopen(TRANS_FILE_1, "r")) == NULL)
        err(link_ss, 2, 0);
    if ((files->linkfp = fopen(LINK_FILE, "w")) == NULL)
        err(link_ss, 32, 0);
}

/*****************************************************************************
 *
 * Function     : close_files_links
 *
 * Description  :
 *          This function closes the necessary files in this module
 *
 * Arguments    :
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

static  void
close_files_links(files)
files_type  *files;
{
    fclose(files->state2fp);
    fclose(files->trans1fp);
    fclose(files->linkfp);
}

/*****************************************************************************
 *
 * Function     : link_subseq
 *
 * Description  :
 *          This is the main function of this module.
 *
 * Arguments    :
 *          efg_type    *efg    : pointer to EFG
 *          files_type  *files  : pointer to all files
 *
 * Returns      : void
 *
 *****************************************************************************/

extern  void
link_subseq(efg, files)
efg_type    *efg;
files_type  *files;
{
    open_files_links(files);

    close_files_links(files);
```