Emma: Developing an Industrial Reachability Analyser for SDL

Nisse Husberg[1] and Tapio Manner[2]

[1] Helsinki University of Technology, Theoretical Computer Science Laboratory, FIN-02015 HUT, FINLAND,
Nisse.Husberg@hut.fi,
http://www.tcs.hut.fi
[2] Nokia Telecommunications, P.O.Box 300, FIN-00045 Nokia Group, FINLAND,
Tapio.Manner@ntc.nokia.com

Abstract Testing products is very expensive in the telecommunication business and remaining errors can also be very difficult to correct in a working system. In this project formal methods are used for the verification of software written in TNSDL (a dialect of SDL-88), which is used as a programming language in telecommunication products. A front-end for the PROD reachability analyser translates the TNSDL code into a high level Petri net model which can be analysed by PROD. The results are translated back to TNSDL. The *complete* TNSDL can be analysed, except some very difficult constructs like pointers. Dynamic processes, all data types, signals with parameters and even timers can be handled. The granularity of the model is very fine, SDL statements are considered atomary but can be folded if they are independent.

1 Introduction

The application of formal methods to industrial problems is not easy. Usually it is a very time consuming task to create the formal model of the system. It has to be done manually and the creator should know the real system very well and he must also be an expert of the formal system. Such persons are, however, still very unusual. Thus the creation of a formal model by hand takes a lot of time and the link between the real system and the model might be distorted.

There are, however, some areas where rather good specification languages are used, for example telecommunications where SDL [9] is quite widespread. The *Emma* project started with the goal to design a system which automatically translates SDL programs into Petri net models which could be analysed with the PROD [13] reachability analyser and the results presented in SDL terms. Both *Emma* and *PROD* are developed at the Theoretical Computer Science Laboratory (former Digital Systems Laboratory).

At Nokia Telecommunications as much as 70 % of the time must be used for testing the systems. The remaining errors must be corrected in running systems which is difficult, slow and expensive. Thus any tool which can be used to make

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 642–661, 1999.

the verification of the systems easier would be most welcome. The TNSDL (TeleNokia SDL - a variant of SDL-88) is used in the implementation of the systems and there are approximately 2 million lines of code written in TNSDL already. Thus the *Emma* project had a good possibility to succeed in creating a useful industrial verification tool in this setting.

The model generator of *Emma* was not trivial to construct. There are data types in TNSDL like arrays and structures - things which often are left out of any academic work about formal modelling. Another problem was that there is dynamic creation of processes and dynamic targets of output statements in TNSDL. Not to speak of timers considering that the Petri net model used in PROD has no time concept. But fortunately all these problems were solved and the model generator can handle the *complete* TNSDL (with exception of rather impossible features like pointers and calls to procedures written in assembler or other programming languages).

The resulting TNSDL analyser was, however, just the first prototype. It was evaluated in a Master's thesis [6] by an engineer working at Nokia Telecommunications and the result prompted the start of a new project, *Maria*, where all the problems of the first prototype should be addressed. There is a working prototype of the *Maria* analyser but it has not yet been tested in an industrial environment. This paper describes the *Emma* analyser and the experiences gained in the *Emma* project and in the evaluation of the prototype. It was very valuable feedback for our laboratory although we have been building reachability analysers for more than 20 years.

Although *Emma* could handle the examples of TNSDL used in testing, the introduction of real TNSDL programs from the production showed that there still are quite a few problems. The problems were only partly related to TNSDL itself. Usually the most serious obstacles in applying *Emma* was the use of non-TNSDL features like calling routines written in other languages and the use of machine dependent features. But when we are taking the step from academic to industrial analysis we have to cope with this kind of problems for the foreseeable future. Although specification languages like SDL are introduced, these problems will persist on the implementation side as in TNSDL.

In this paper this problem is discussed and some solutions are proposed. In the evaluation of *Emma* about 40 out of the nearly 500 program blocks written in TNSDL for the DX 200 telecommunication system were used. This paper contains a section about the design of *Emma*, one section about the experience of using *Emma* and finally one section about the new analyser *Maria*, which will be a combination of *PROD* and *Emma*, but with many new features.

2 General Design of *Emma*

The creation of the formal model is a serious obstacle in industrial formal verification. The creator should be an expert both in the application area and in formal models. Academic personal may know a lot about formalisms and indus-

trial engineers a lot about the system to be verified, but this knowledge is seldom easy to bring together.

In the *Emma* project the main idea was to make the modelling easy for the industrial engineer, preferably automatic. Therefore *Emma* consists of a TNSDL parser, a net model generator, a reachability analyser and a result interpreter (Figure 1). In fact, the net model generator and the result interpreter are integrated into one C++ program, **emma**.

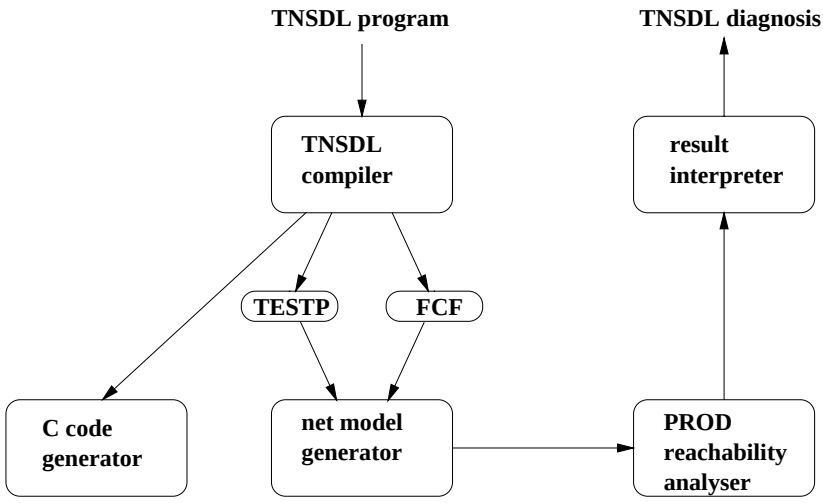

Fig.1. The *Emma* analyser.

The TNSDL parser used in *Emma* is part of the TNSDL compiler developed at Nokia Telecommunications. This compiler is used in the production and generates C code from the TNSDL source. It was only extended with a module writing the *flow control data* (FCF) to a (text) file. It can also write a text file containing the *symbol table* (TESTP) of the compiled program.

The model generator uses the *same information* as in the C code generator. Thus the model is very close to the implementation of the system thereby reducing possible modelling errors. The formal model which is verified must be a correct abstraction of the real system. As will be seen in this paper, this abstract model is difficult to create because the complexity must be kept quite low and the implementation (not only "clean" TNSDL code) causes problems.

The *PROD* reachability analyser [13] is a Petri net tool implemented in C by a student group in the early 90's, but substantially improved by Kimmo Varpaaniemi [12]. It uses an input language which is close to Predicate/transition nets (Pr/T nets) and can do on-the-fly verification of LTL properties and use reduction methods like stubborn sets and sleep sets . One problem with the model generation from TNSDL was that the data structures had to be coded into tuples of integers, which is the only data type that *PROD* knows.

The implementation of *Emma* is still on the prototype stage. Now the analyser consists of three different parts: the TNSDL compiler (internal tool at Nokia Telecommunications) which is written in C++ and runs only under Sun OS (and MS Windows NT), the **emma** program, written in C++ and using libraries from the TNSDL compiler running under Linux and the *PROD* reachability analyser written in C code and running under Linux (and MS Windows).

The experience with *Emma* showed that the idea was good, but also that the analyser should be rewritten completely and this is now done in the *Maria* project [10] which will produce a complete SDL and Petri net reachability analyser written in C++ under the Gnu public license, i.e. it will be *freely available*. The development of *Emma* is restricted to smaller bug fixes.

3 Automatic Generation of the Net Model

In principle it is not very difficult to translate SDL into Petri nets because both "languages" are designed for the description of concurrent systems. There are, however, certain problems with the data types and timers in SDL and the most important thing is to create a model which can be analysed. Reachability analysis easily generates huge graphs and even relatively simple systems can be impossible to analyse if a combinatorial explosion is allowed to take place.

When using Petri nets it is very important to decide how the *dynamic* features are modelled. The structure of the net is *static* and can not be changed by the behaviour. Fortunately, this is not necessary in high level nets, because it is easy to code the complete dynamics using the *marking* (tokens). It is even possible to use a high level net which consists of only one place and one transition, but then we are only trading the structural complexity with annotational complexity.

At first sight it seems appropriate to use one net transition to model one SDL statement. This is also possible for TNSDL except in the `WHILE` statement and in the call of a procedure. It is, however, not very efficient in reachability analysis because there will be too many intermediate states. If the system consists of several processes, as usually is the case, then there will be a lot of interleavings.

In *Emma* this approach is used, but in order to reduce the number of interleavings, a special *lock place* is added to the net. It is not the most optimal solution for analysis efficiency, but it allows the user to closely follow the behaviour of the system in terms of executed TNSDL statements.

The net transitions are connected using *control places*, e.g. **C1** and **C2** in Figure 3. Note that an *SDL transition* consists of a sequence of SDL statements, among others `OUTPUT` statements and statements referring to global variables. Thus these SDL transitions can not be treated as atomary behavioural unit as is done e.g. in ObjectGEODE [7]. *Emma* can treat sequences of SDL statements as behavioural units, but can break up SDL transitions if necessary.

3.1 Modelling Data

It is usually impossible to model the data in a real system completely. Thus it has to be limited heavily. In the analysis it makes little difference because there are a lot of symmetries which can be ignored in the verification.

The data in the SDL program is modelled in the marking of the net model. A *variable* is a *place* in the net and its value is a *token*. Because Petri nets are *resource-conscious*, the tokens which are read by a net transition are removed from the place and must be explicitly restored. A token written to a place does *not* destroy the old value, which must be removed before adding a new value.

Therefore it is necessary to use the marking of this place as a *frame*, i.e. the number and type of tokens are always the same, only the contents (the value) of the tokens is changed (Figure 2). Typically the same net transition which reads the token also returns it, possibly with a new value. This is so common that double arrowheads are used in the pictures (Figure 3).

This feature in Petri nets is quite useful when there are several instances of a variable. It is only necessary to tag the values in order to make them distinguishable. In this way *dynamic creation of processes* is quite easy to model.

One special "variable" is the *queue* used in SDL processes. It is implemented using frames although these frames are rather complex. One token at a queue place defines a slot in one queue. In *Emma* such a token is a quintuple `<.pid,spid,n,sig,par.>` consisting of the *process identifier* `pid`, the *sender process identifier* `spid`, the *queue entry number* `n`, the *signal number* `sig`, and the signal *parameter information* `par`. Because the process identifier is unique, it means that dynamic creation of processes can be handled with this model.

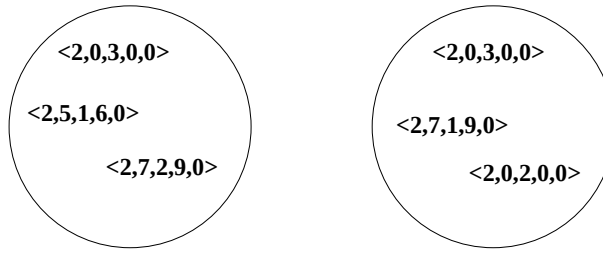

Fig.2. The queue modelled as a frame.

In the example in (Figure 2) the queue of process 2 has three entries, with the first one containing signal 6 sent from process 5, the second one has signal 9 sent from process 7, and the third one is empty. There are no parameters in the queue. When the first signal in the queue is read, the new queue (at the right) contents will be reduced to signal 9 sent from process 7 in the first entry. Note that there is no order among the tokens at a place in a Petri net. Therefore the entries must be numbered explicitly. The use of frames makes it much easier to

handle the net because there are problems with testing if there is a token in a place especially if the number of tokens is not bounded.

It is possible to use one single place for all queues in the net because the process identifiers are unique. This would be useful in handling the `OUTPUT` statements in SDL which have *dynamically defined targets*. It was reflected upon in *Emma*, but even if it does make the model simpler, it does not have any influence on the size of the reachability graph, which is more important.

The queue needs two auxiliary places *Cursor* and *Free*, which keep track of the first free slot and the signal to be input, respectively. Note that the signal is *not* removed from the queue by the net transition corresponding to the `INPUT` statement in SDL - only the *Cursor* is incremented. There are several reason for this, e.g. the existence of a `SAVELATEST` statement in TNSDL, which may force the signal to remain in the queue. Only the net transitions which move control to another SDL state (like `NEXTSTATE`) will clean the stack and set the *Cursor* to zero.

The `SAVE` statement in SDL should move a signal from the input queue to a save queue and put the save queue back to the input queue when the SDL state is changing. In *Emma* the **Save** net transition only increments the *Cursor*. This makes the change in the state of the net very small and with smart state space storage algorithms (storing only the *change* in the state) this should keep the memory needed for the state change very small. The queue model used in the SDL modelling of the PEP tool generates a lot of intermediate states [3] resulting in a big state graph.

In *Emma* the length of the queue is fixed because *frames* are used. This is in conflict with the definition of SDL, but it has no practical significance at all. In the analysis of the systems the length of the queue must be limited very heavily or the size of the reachability graph will really explode. In our tests with rather small examples, the maximal queue lengths which could be analysed were usually shorter than five. On the other hand, in the verification of systems the errors typically can be found already using very short queues. The increase of the queue length generates a lot of similar subgraphs, but very little new behaviour.

In the new analyser *Maria* a *queue data type* is introduced in order to increase efficiency and reduce the number of intermediate states in the reachability graph.

It was easy to implement variables with simple types, but the inclusion of more complex data types turned out to be rather troublesome and an excessive amount of work was needed to implement the translation of all possible expressions found in TNSDL. Everything had to be coded using tuples of integers.

A basic problem was that the structure of the net must be determined at translation time because it cannot be changed dynamically. Thus the SDL program must be analysed very carefully before the net structure is generated. However, the net class used in PROD was powerful enough to handle the creation of processes and *recursive procedure calls* can also be modelled. The solution is to tag all the tokens in a recursive process with the depth of the recursion.

Because industrial systems often have huge or infinite state spaces it is impossible to generate the complete reachability graph first and then perform the

analysis using this graph. Therefore the *on-the-fly method* must be used. It means that the graph is checked for the interesting property, e.g. a deadlock, at the same time as the graph is generated. If a deadlock is found, the process is stopped.

This makes it possible to work even with infinite state spaces, but the search strategy is extremely important. Simple search strategies like depth-first and breadth-first are not very useful in this case. Here *smart strategies*, the possibility to give the analyser some *hints* or even the use of *manual control* of the generation of the reachability graph are needed.

3.2 Dynamic Structures

The *static* structure of the net makes it impossible to change the output arcs of an **Output** net transition dynamically. When the SDL source has the target of the `OUTPUT` statement defined as a variable which can have several different values at run time it is necessary to generate *several* **Output** net transitions to all possible input queues. Only one of these can, however, be enabled for each instance and the size of the reachability graph is not affected.

The problems with transferring the parameters of a signal is related to the dynamical determination of the target of an `OUTPUT` statement in SDL.

A third problem is the dynamic determination of timer instance names. In principle a timer in SDL can have an infinite number of active instances distinguished by the parameters of the timer. Because the parameters are determined when the timer is activated, the name is determined dynamically. This causes problems as the `SET` statement in SDL will reset an active timer with the same parameters and set it again. The net transition must check the actual parameters with the parameters of all active timers, but it is impossible to have a variable unbounded number of active arcs in the net transition.

Thus it was necessary to have a fixed number of active timer instances for each timer in a process instance. The tuples for these timer instances are *frames* which are created at startup and remain constant in number and form during the whole analysis, except that their contents might vary.

3.3 Modelling Timers without a Time Concept

It seems impossible or at least useless to model timers in a formalism which has no concept of time at all, but in *Emma* it is done using a "window" technique. It does not even pretend to model the complete behaviour of timers, but at least a good approximation is achieved. The basic reason for this is that the communication in SDL is completely *asynchronous* and thus it is impossible to make any assumption of "when" a certain timer will expire in another process (in the same process it is possible).

In fact, the timer would be modelled in exactly the same way in a formalism having a time concept, i.e. the timer can in principle expire *at any time* seen from other processes. This will cause the reachability graph to be very big and some restriction which removes all the uninteresting paths must be used.

The solution to this problem was to implement a kind of "window", where the timer was allowed to expire. In practice, a special *lock place* (Figure 3) was introduced and the timer can only expire if this lock place has the right token. When the timer is set, the lock is closed because the **Set** net transition puts a CLOSED token into the lock place and it can be opened by any net transition which puts an OPEN token into this place. Any net transition can close the lock again. In this way a "window" is opened and the timer can expire only within the window, i.e. it will only be interleaved with the net transitions in the window - not with all net transitions in the model.

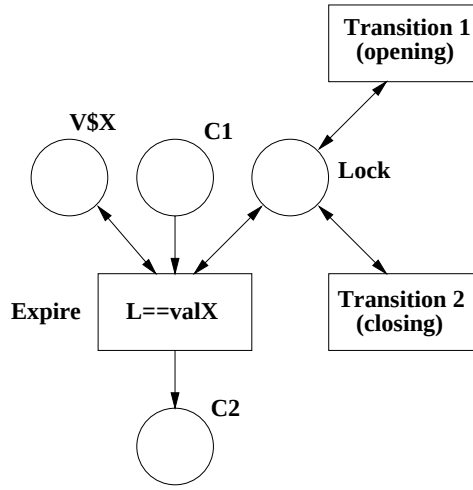

Fig.3. The timer model.

The transition **Expire** in Figure 3 is disabled until **Transition 1** puts a token $< pid, L >$ into it. After that the transition can fire if `L == valX`, but when **Transition 2** is fired it takes away the token in the lock place and the transition **Expire** is locked again. Note that there must always be a token (a frame) in **Lock** so that it does not interfere with the behaviour of **Transition 2**. In some paths the **Transition 2** might be fired before **Transition 1**.

The lock is closed by putting the token $< pid, L0 >$ into **Lock** (where **L0** is a value which cannot be a value of the variable **X**). In the example both transitions **Expire** and **Transition 2** will close the lock. Of course there can be much more complex situations.

The user can manually define where the lock should be opened and closed or he can let *Emma* use some algorithm for defining the window automatically. He can try different windows in critical segments of the code and see if erroneous behaviour occurs. This is probably a much faster way to analyse the system than generating a very big reachability graph (if it is possible at all).

When a timer expires, a signal is put into the queue, but this signal cannot be read until the next **Input** net transition. Therefore it is possible to keep the timer locked during the whole SDL transition, i.e. all SDL statements except `INPUT` and `OUTPUT` (affecting the same queue as the timer or having dynamical target). It is necessary to open the lock also before an **Output** net transition so that the order in the queue will not be altered - if the target of the **Output** is dynamical, it is not possible to know which queue it affects.

3.4 Limiting Concurrency

The reachability graph contains a lot of different interleavings of the same net transitions. If these net transitions are independent, it is unnecessary to consider but one of them. This means removing "unnecessary" concurrency. There are several methods for doing this ([11], [2], [14]), but here the *lock* mechanism can also be used as an *control distributor* or *sequencer*. Note that this method reduces the concurrency *at the level of the model*, while the above mentioned methods reduce the complexity at the level of the reachability graph.

Control distributors can be used with any analyser which can take the net model as an input - not only with a reachability analyser. This is important in the *Emma* project, where the use of multiple methods is stressed.

Especially in big systems with possibly infinite reachability graphs it can be necessary to limit the concurrency by using the control distributor very heavily. This can change the behaviour of the system considerably, but it might be the price to be paid for making analysis possible at all. The analyst can try different possibilities and maybe get some useful results even from very complex systems.

The biggest reduction in the reachability graph is achieved if control is given to a process only when it is necessary and given away again by the following net transition. Because the SDL processes can interfere with each other *only* by asynchronous communication, the only interesting windows are around **Output** and **Input** net transitions.

A more general form of a control distributor place is a *counter place*. It can be used to control the number of processes created etc. Each time a certain net transition fires, it decrements the counter and when it is zero, the net transition cannot fire any more. By making the system to put fresh tokens in the counter place in certain cases, the dynamics of the system can be controlled in a very sophisticated way.

It must be stressed that these *approximative* methods are fundamentally different from those used in [11], [2], [14], where the reduction is proved to retain the interesting properties. In an *industrial* analyser the use of *partial* analysis is often more interesting because the main objective is to *find errors quickly*. Note, however, that this approach is based on a formal model and that *exhaustive* analysis can be used as well.

Typically the work with *Emma* would consist of doing a lot of partial analysis checking for errors and at the end an exhaustive analysis would be done (perhaps taking several weeks). In an industrial environment an exhaustive analysis

is, however, seldom possible to do at all because of the huge (often infinite) reachability graph.

4 Experiences from Industrial Verification

When the first prototype of *Emma* was ready, a project was started to evaluate its usefulness in a real industrial environment. It was not possible to do an evaluation in the university because the conditions are quite different. A good academic tool is not necessarily very useful in an industrial setting.

Emma was tested at Nokia Telecommunications by Tapio Manner, who had some experience of *Emma* but did not take part in the development directly. We got very valuable feedback needed in the next stage of the development of the analyser.

The evaluation was not only a test of *Emma* using more complex TNSDL programs but a test of its usefulness as a tool in the software development of the entire DX 200 system. The importance is in the analysis of the concurrent behaviour of the entire system instead of separate program blocks.

There are some details in the DX 200 source code which are not analysable with the *Emma* analyser. Some of them were already documented in [5] and [4]. During the evaluation some new problems were found. These are divided into five categories: *other programming languages*, *pointers*, *linking*, *DX 200 specific features* and *bugs*. The bugs are only a temporary quality problem in the prototype, but the other four categories represent DX 200 implementation details which were not addressed in the design of the *Emma* analyser.

4.1 Other Programming Languages

The telecommunication systems developed at Nokia Telecommunications are implemented using several programming languages. A good deal of the asynchronous communicating processes belong to program blocks written in PL/M, C or assembler. The operating system interface and many other basic services appear as *synchronous services* in libraries, not necessarily written in TNSDL.

The *Emma* analyser can read only TNSDL descriptions, the programs and synchronous services written in other languages are outside the scope of the analyser. To *Emma* these parts of the system are unknown – possible effects on the system state can not be detected.

However, *Emma* is not completely powerless in this case. It can model the effect of an external synchronous service by a random value. This gives a partial model of the synchronous service. If a better model is needed, it must be modelled manually as a TNSDL process or directly as a Petri net. This only takes us back to square one, i.e. to the situation before *Emma* was created.

Some C source code may also be included within the TNSDL source code file. The `MACRO` feature in TNSDL supports the use of in-line C language macros implementing either synchronous services or data type operators. The macros

are mainly used for reasons of efficiency. The macros cause the same kind of problems as the external synchronous services implemented as procedures.

Analysing the concurrent behaviour of the complete system with only partial access to the system description is impossible. This is why the inaccessibility to the parts written in other languages is probably one of the greatest shortcomings in *Emma* from the industrial point of view. Some ideas how to get the behaviour of these black boxes more visible to *Emma* is presented in Section 5.

4.2 Pointers

Pointers can be used to optimise the use of stacks with respect to size and time by avoiding the copying of large structures to the stack in procedure calls. Pointers can also be used to make changes in structures transparent to interfaces: adding a new field to a structure does not require rewriting related procedure prototypes. A pointer is a very convenient in implementing dynamical data structures. These are important in reserving a minimal amount of memory while retaining flexibility to meet peak size needs.

Parameter passing, especially to external synchronous services, is where pointers probably are used most frequently. Many synchronous services can be rewritten not to use pointers in their interface, but it is questionable whether this is reasonable in general. Section 5 discusses some widely used cases which are transformable to program code without pointers.

4.3 Linking

In an industrial system it is not only the program code that defines the behaviour. For example, the linking of modules in TNSDL is a rather complex operation where the *visibility* of procedures can be restricted.

The linkable modules of processes in TNSDL are composed into a linkable module of the entire program block. The object files of TNSDL modules can be linked into any of these linkable modules. Every link of the TNSDL module places one copy of the object code into the target module. In addition to possibly many physical copies of TNSDL module with binding, the visibility of procedures of the TNSDL module can be restricted. As this is defined outside the TNSDL source code, *Emma* expects the TNSDL modules to be linked into a single shared copy of module in the program block. If this is not the case the model represents something else than was defined in TNSDL source code.

4.4 DX 200 Specific Features

To increase the fault tolerance there are spare units in DX 200 storing backup values. This system is not fully transparent to the programs. There are a few statements having attributes tuning the fault tolerance features. For example the `OUTPUT` statement can contain implementation-dependent attribute specifications, which are needed in certain situations causing the operating system to

deliver the message to the destination process in a specific way such as "delete the message if the receiver is in a spare unit" or "put the message into the receiver's message queue even if this would cause queue-length quota to be exceeded".

What happens to the system when the working unit and the spare unit for some logical computer unit change their roles on the fly is something that the analysis with *Emma* can not handle.

4.5 Bugs

During the tests several bugs were found and many of them were successfully corrected. However, there are still some problems which will be corrected only in the new version, i.e. in the *Maria* analyser. One of these is the handling of *union types*. This is a problem in TNSDL because the programmers often use a union type as a formal parameter in a procedure to save space. The actual parameter type is known only when the procedure is called. The new analyser *Maria* has a complete type system and the union types will be handled by it.

4.6 Testing with Real Programs

The purpose of the tests was to show how *Emma* copes with real TNSDL programs, and learn what kind of modifications must be made in order to get them analysed with *Emma*.

The extensive test suite that was developed can later be used in testing future versions of *Emma*. This work produced a lot of workarounds, i.e. methods of adapting the source code to analysis with *Emma*. For industrial application there is a need to design a new front-end which processes the TNSDL source code used in the production. Now editing one of the smallest program blocks according to the suggested workarounds takes days, which is intolerable.

In Section 5 it is described which constructs in the TNSDL source code must be handled to make the analysis possible. The following list shows how common each problem is by giving the proportion of program blocks involved. All of these problems can be solved by the methods in Section 5. Note that some of problems can not be solved without changing the behaviour of the TNSDL system to some extent.

5 Preparing TNSDL Source Code for the Analysis

Many tools are presented using some examples which work very nicely, but here we use "dirty" production programs which we try to verify using *Emma*. It was difficult to find a program which could be analysed directly. Usually some TNSDL source code modification was necessary, in some cases a rewriting of the code was necessary. This is, however, still very far from creating a formal model manually. Table 5 shows the frequency of the most important problems encountered during the evaluation.

Table1. Problem Frequency

The Problem Source	Ratio in %
pointer	98
external synchronous services	100
union	71
memset	55
memcpy	33
block without implementation	100
output attributes	14

Pointers are probably causing most of the problems because *Emma* does not handle pointers. Clearly the necessity for manual modifications impedes common use of the *Emma* system because pointers are used in every TNSDL program studied in this work. It is, however, possible to do some modifications automatically, but manual modification will always be at least a partial solution to deal with pointers. Most of the guidelines given in this section are actually procedures precise enough for automatic implementation.

The second group of changes is related to *synchronous services*. Most of the changes are due to the use of pointers in parameters, but it is convenient to handle this separate from detailed pointer discussion.

The internal behaviour of a synchronous service is unknown to *Emma*, only the interface is known. In most cases, the only option available is to simulate the interface and ignore the behaviour. In some cases, we can build a TNSDL implementation for an external synchronous service. There are also external synchronous services that can be ignored without problems.

5.1 Modelling Synchronous Services

Some of the synchronous services have no effect on the analysis at all like the those offered by the program block CLUGEN which typically are procedures that display data at terminal monitors. All use of these external synchronous services can be left out of the analysis. To speed up the removal of irrelevant synchronous services a tool program **emmastrip** was made when preparing the program blocks for the tests. The tool is described in more detail in Section 5.5.

When the behaviour of a synchronous service is absolutely essential to the analysis one can write partial substitutions called *stubs* in TNSDL, or even completely replace some synchronous services with TNSDL programs. In TNSDL, the functions from the ANSI C library defined in `string.h` can be used. There exist synchronous service definitions for them in the TNSDL library. Fortunately one can replace the C statements with equivalent TNSDL statements.

memset conversion example The following example demonstrates the results of performing such a translation. Consider the given data type definitions:

```
TYPE struct1_t
    REPRESENTATION STRUCT
        big     word;
        pos     bool;
    ENDSTRUCT;
ENDTYPE struct1_t;

TYPE struct2_t
    REPRESENTATION STRUCT
        digit   byte;
        ptr     bytepointer;
        string  array(2) of character;
        stru    struct1_t;
    ENDSTRUCT;
ENDTYPE struct2_t;
...
DCL
   x struct2_t;
...
```

If a piece of source code has as its original form

```
TASK memset(x, 0, SIZEOF(struct2_t));
```

then applying the translation rules results in

```
TASK x.digit := 0;
TASK x.ptr := NIL;
TASK x.string(0) := character(0);
TASK x.string(1) := character(0);
TASK x.stru.big = 0;
TASK x.stru.pos = F;
```

memcpy(target, source, size); is normally used to copy data between variables of the same data type. Copying between differing data types is normally an error, and is not considered further here. With this simplification `memcpy` turns into an assignment of the value of one variable to another variable. With simple data types a simple assignment statement will do, but with structures the assignment must be performed field by field as in initialisations in the `memset` case. TNSDL allows direct assignment of arrays, which makes their treatment easy [1].

The predefined variable `UNINTERPRETED_SIGNAL_DATA` is a special case where the array must be copied entry by entry up to `UNINTERPRETED_SIGNAL_LENGTH`. A simple array assignment will not do because the size of `UNINTERPRETED_SIGNAL_DATA` is not known at compilation time.

The untagged union type is a problem in *Emma*, but it will be solved in the *Maria* analyser, which has a union type.

[1] Surprisingly TNSDL assignment with arrays is more efficient in machine code than `memcpy`

Some software designers always build a structure for a union. The structure has two fields: the union field and a tag field indicating the type used in the union field at any specified moment. In this case, the analysis engineer has the possibility to write a `DECISION` statement which handles all possible alternatives correctly. The tag field does not allow automatic transformation because it is not specified exactly how to refer to each of the union members.

The principle is the same for handling many other functions like `memmove` which copies a number of bytes from one memory address to another, `strcpy` which is a special case of `memcpy`, `strncpy` which is a variant of `strcpy` and `strncmp` which compares two strings byte by byte.

Efficiency Considerations As always in reachability analysis it is very important to take the effect on the size of the reachability graph into consideration also when some functions are replaced by TNSDL code. When TNSDL arrays are initialised as in the `memset` case it is possible to choose between writing a `TASK` statement for each array index, or using a loop.

1) Straight code:

```
TASK ai(0) := 0;
TASK ai(1) := 0;
TASK ai(2) := 0;
TASK ai(3) := 0;
TASK ai(4) := 0;
etc. up to 7
```

2) Loop:

```
DCL i byte;
...
WHILE (i < SIZEOF(ai));
    TASK ai(i) := 0;
ENDWHILE;
```

These alternatives were added to a system consisting of one program block and one master process. The master process contained originally only the START SDL transition which ended in a `STOP` statement and the net model consisted of 27 places and 3 net transitions giving a reachability graph with 2 nodes.

Adding the straight code increased the net model with 9 places and 8 transitions giving a reachability graph with 8 more nodes. Adding a loop increased the net model with 5 places and 5 transitions, but resulting in a reachability graph with 34 additional nodes. The result suggests that loops should be avoided.

Concatenating assignments of local variables within a TNSDL transition into a single net transition would reduce the size of the reachability graph. A simple rule is that all local variable assignments where the left hand side (that is the target of assignment) is not used in the right hand size of an assignment or as by-reference argument of a procedure call or in any other expression can be concatenated without distorting analysis.

This is really a question for the model generator in *Emma*. It is possible to join multiple assignments in a TNSDL transition into a single net transition as long as the set of input places is distinct from the set of output places as suggested by Markus Malmqvist [5].

5.2 Data Type Related Changes

Because of the problems with union data types all data types of this kind must be changed. There are basically three alternatives:

- Change a UNION type into a structure. If the size of the data type is not essential the changed source code behaves identically except for the growth in memory consumption.
- Replace a UNION type with a structure where only one of the union's alternatives exist. Then remove all source code related to the alternatives.
- Remove the UNION type completely and all source code related to it.

In the next generation analyser, Maria, the problems related to the union data type are solved elegantly by connecting the data type to a value for every piece of data in a union.

5.3 Handling Pointers

Due to extensive use of pointers a large portion of the TNSDL source code would require modifications to make the source code understandable to *Emma*. In many situations, the only possible way to modify programs to pointer-less is to completely remove statements containing pointer interpretation. In addition to the fact that such removals are time consuming to perform, they might change the concurrent behaviour. This is the main reason why the analysis engineer must consider very carefully whether to remove or replace a problematic statement.

In this section, a few possible ways to handle TNSDL pointers are defined. They are designed to allow *Emma* analysis with very limited preparations. Easy access to the analysis is essential if one wants to include it into a set of normal software development activities, much in the same way as *lint* variants are used in professional C source code development.

Neglecting pointers can cause trouble in cases where pointers are used in expressions of `DECISION` statements for example resulting in loss of some executions paths. These paths may contain `OUTPUT` or `NEXTSTATE` statements. To have the source code make sense syntactically we must consider the entire expression, not just pointers. We could simply use some default, e.g. FALSE.

To avoid infinite loops it is suggested that the (sub)expressions containing pointer handling in a `WHILE` loop condition are interpreted as FALSE by default. The only exception to this is the NOT expression: with NOT operators the outermost NOT must evaluate to FALSE. The outermost here does not mean code such as `NOT(T OR NOT(expr))` but code such as `NOT(NOT(expr))`.

The complete removal of pointers is a very harsh treatment of the original TNSDL source code and should be avoided.

If all accesses to an object to which a pointer refers ($ and ->) were interpreted somehow, complete removal would be necessary only for references accessing an address of an object (@). It is suggested that the default values for $ and -> dereferences evaluates to the initialisation value of the corresponding data type as in the `memset` case. The default interpretation does not give much more precision to analysis compared to complete removal. What it gives is a closer relationship between the original source code and a more complete path in the interpreted reachability graph because fewer statements are removed.

A better way is to let the user specify a value for a pointer expression or give a replacement for a statement containing a pointer. These could be specified in the option file of *Emma*.

5.4 DX 200 Specific Features

This section describes how the DX 200 specific features mainly related to fault tolerance issues can be eliminated from the source code.

Usually addressing a message is done by using the process identifier available in predefined TNSDL variables `SELF`, `PARENT`, `OFFSPRING` and `SENDER`. The most common exception is the start-up messages using the signal route definition in the routing file. In some rare backup related situations, neither of these methods will do but we must determine the process identifier dynamically, for example if it is necessary to change the computer information of the sender from the spare unit to the working unit. Such treatment of backup strategies is quite out of the scope of *Emma* and such modifications may be commented out without problems in the analysis.

There are also cases where a process identifier is not only changed, but built from scratch. Should this happen there is no way *Emma* could construct the address of a receiver.

The `OUTPUT` statement can contain implementation-dependent attribute specifications. The following is an example of this use of attributes and how it should be commented out:

```
OUTPUT print_char_s TO special_pid /*, SET GROUP=funny_c,
                                       PRIORITY=max_priority-1*/;
```

5.5 Tool Set

Most of the changes to the TNSDL source code files must be done manually before feeding them into the *Emma* analyser. To make preparation easier some tools were made to find possible sources of problems and to automate some of the changes as defined in this section. All of them alter either the TNSDL source code or the FCF and TESTP files generated by the TNSDL compiler.

emmastrip removes the calls of synchronous services not relevant to the analysis of concurrency. This is a very useful tool because it succeeded in removing a great number of problematic statements in every program block in the sample programs.

chkptr investigates TNSDL source code modules and reports any use of pointers. This is useful in pinpointing statements requiring special attention. The `chkptr` is a simple Bourne shell script using basic Unix tools to do the search efficiently. It simply searches for occurrences of @, $, and ->.

6 The New Modular Analyser *Maria*

Emma was intended to be a modular analyser but it was impossible to achieve this goal using *PROD*, which is implemented in a very "non-modular" way and

difficult to develop further. Together with the problems concerning complex data types this caused us to start designing a completely new analyser, *Maria* (Modular Reachability Analyser, Figure 4). This project [10], financed by TEKES, started in 1998 and will result in a free analyser (under Gnu Public License) with front-ends for both standard SDL (possibly SDL-2000) and TNSDL, in addition to typed (sorted) Pr/T nets.

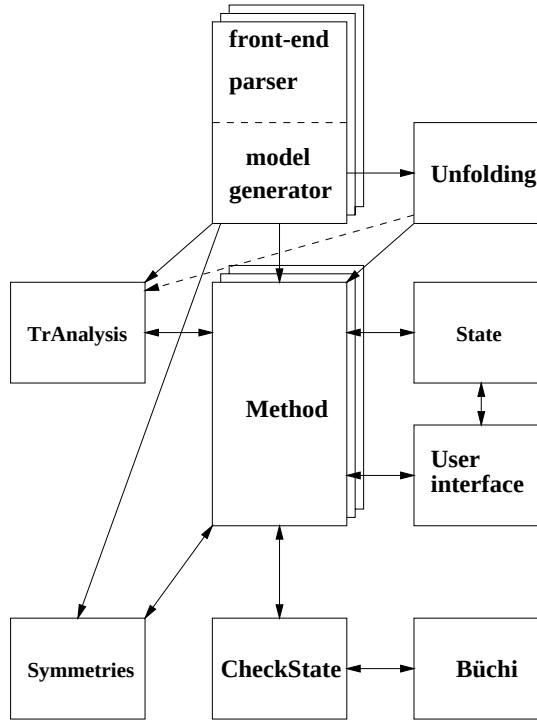

Fig.4. The *Maria* analyser.

The new analyser should be easy to connect to other tools like Design/CPN [1] and [8] which have very good graphical front-ends. It is also important that new analysis methods can easily be added to the analyser because there is no method which would be the best for all kinds of systems. Thus we need a number of different specialised analysis methods.

The experience from applying *Emma* to industrial programs showed that substantial changes and simplifications have to be made to make any analysis at all possible. As mentioned it is a real problem to handle huge and infinite reachability graphs. Therefore the new analyser will have methods for on-the-fly, but also *partial* reachability analysis. In *Emma* there are some options which can give a partial analysis, but these possibilities must be expanded considerably.

These approximative methods use the *same* model which is used for complete (and usually very time consuming) reachability analysis. They are used mainly for fast debugging or when complete analysis is impossible. It is considered important also that the user can guide the analysis to those portions of the code which he considers most sensitive to errors.

Storing the reachability graph is not very efficient in *Emma*. A graph with only 35 000 nodes may need as much as 150 Mbytes. This must be made much better in the new analyser. The *State* module will be implemented using efficient techniques like incremental state storage.

The most important thing is *not* how many states the analyser can generate, because most of those states are correct and thus of no interest. What we really need are new methods which can leave the correct states out of the generated graph and concentrates on those parts which may contain problems.

The*user interface* of the analyser must be improved before it can be used in an industrial environment. Partly this can be obtained by making interfaces to other tools as mentioned above, but also the reachability analysis must be easier to follow and control. This would also increase the efficiency of the analysis, especially when a lot of interactive work is required. As suggested above, there is also a need for tool support in *preparing* the programs for analysis. Another important question, which will not be addressed here, is how to make a good interface to the model checker. It must be understandable also for an engineer without special training in time logic.

7 Conclusions

The *Emma* project was unexpectedly successful because the *complete* TNSDL could be handled *automatically*, including complex data types, dynamic processes and timers. The analysis can be performed with a very fine granularity down to the SDL statement level, but can also be made coarser e.g. for sequences of independent statements.

Although successful on the pure TNSDL level, the evaluation at Nokia Telecommunications showed that there still is a lot to do before the analysis can be performed automatically in a real industrial environment. This is mainly a question of preparing the TNSDL programs for the analysis. It should be noted, however, that this preparatory work, even as it stands now, is just a fraction of the work which has to be done in manual design of a formal model.

A future project for implementing a *new more powerful analyser*, *Maria* is already under way at the Theoretical Computer Science Laboratory. It will implement many of the ideas behind *Emma* which could not be realised in the settings available earlier. This is a very research intensive project but the driving force will be the needs of the practical software engineering involved in the design and maintenance of distributed concurrent systems.

The construction of *Emma* and the industrial evaluation of it has shown that it is possible to build an industrial analyser for a *complete* and *real* programming

language using high level Petri nets. The first prototype has brought good experience and the industrial feedback shows the direction of further development.

The aim with the *Maria* project is to make this new analyser useful both for academic prototyping and industrial analysis.

8 Acknowledgements

Special thanks to Esa Kettunen who initiated the *Emma* project while working at Nokia Telecommunications, which completely financed the project. This work has partly been supported also by the ETX program of TEKES - the Technology Development Centre in Finland.

References

[1] Design/CPN Home Page. http://www.daimi.aau.dk/CPnets/.
[2] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Phd thesis, University of Liege, November 1994.
[3] Berndt Grahlmann. *Parallel Programs as Nets.* Phd thesis, Universität Hildesheim, September 1998.
[4] Tero Jyrinki. Dynamical Analysis of SDL Programs with Predicate/Transition Nets. Technical report, Helsinki University of Technology, Digital Systems Laboratory, April 1997.
[5] Markus Malmqvist. Methodology of Dynamical Analysis of SDL Programs Using Predicate/Transition Nets. Technical report, Helsinki University of Technology, Digital Systems Laboratory, April 1997.
[6] Tapio Manner. Extending Verification of Industrial TNSDL Programs with Formal Methods Using Emma. Master's thesis, Helsinki University of Technology, Theoretical Computer Science Laboratory, November 1998.
[7] ObjectGEODE Home Page. http://www.verilogusa.com/solution/geode.htm.
[8] SDT Home Page. http://www.telelogic.se/.
[9] CCITT Specification and Description Language (SDL). Technical Report Z.100 (1993), ITU-T, June 1994.
[10] Theoretical Computer Science Laboratory, Helsinki University of Technology. *Maria - a Modular Reachability Analyzer*, October 1 1998. Unpublished research plan 4.0.
[11] Antti Valmari. State Space Generation: Efficiency and Practicality. PhD thesis 55, Tampere University of Technology, Tampere, Finland, 1988.
[12] Kimmo Varpaaniemi. On th Stubborn Set Method in Reduced State Space Generation. Research Report 51, Digital Systems Laboratory, Helsinki University of Technology, FIN-02015 HUT, Finland, May 1998.
[13] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. PROD Reference Manual. Technical Report 13, Digital Systems Laboratory, Helsinki University of Technology, FIN-02150 ESBO, Finland, August 1995.
[14] François Vernadat, Pierre Azéma, and François Michel. Covering Step Graph. In J. Billington and W. Reisig, editors, *Application and Theory of Petri Nets 1996*, volume 1091 of *LNCS*, pages 516–535. Springer-Verlag, 1996.