Analysis of a Session-Layer Protocol in mCRL2*
Verification of a Real-Life Industrial Implementation

Marko van Eekelen[1] Stefan ten Hoedt[2]
René Schreurs[2] Yaroslav S. Usenko[3,4]

[1] Institute for Computing and Information Sciences, Radboud Universiteit Nijmegen,
PO Box 9102, 6500 HC Nijmegen, The Netherlands

[2] Aia Software B.V.
PO Box 38025, 6503 AA Nijmegen, The Netherlands

[3] Laboratory for Quality Software (LaQuSo), Technische Universiteit Eindhoven,
PO Box 513, 5600 MB Eindhoven, The Netherlands

[4] Centrum voor Wiskunde Informatica,
PO Box 94079, 1090 GB Amsterdam, The Netherlands.

Abstract. This paper reports the analysis of an industrial implementation of the session-layer of a load-balancing software system. This software comprises 7.5 thousand lines of C code. It is used for distribution of the print jobs among several document processors (workers). A large part of this commercially used software system has been modeled closely and analyzed using process-algebraic techniques. Several critical issues were discovered. Since the model was close to the code, all problems that were found in the model, could be traced back to the actual code resulting in concrete suggestions for improvement of the code. All in all, the analysis significantly improved the quality of this real-life system.

1 Introduction

In this paper we consider the following real-life industrial case study. The ITP Document Platform (developed and marketed by Aia Software BV) enables organizations to produce critical business documents in a scalable and personalized environment. This application has a load-balancer, a process kernel that makes diverse document processors and clients communicate with each other, distribute and execute tasks. This system is used satisfactorily for several years (in 2007 in over 25 countries by more than 800 customers). However, it comes every now and then in an undesirable state. The goal of the project was to investigate to what extent the inter-process communication and synchronization of this load-balancer could be modeled and analyzed. The desired results had to be detailed enough to give an advise on how to avoid these undesirable situations, and to suggest concrete code changes.

* This research was supported by SenterNovem Innovation Voucher Inv053967. The fourth author has also been supported by NWO Hefboom project 641.000.407.

The project has been performed in the following phases: In a discussion with two employees of Aia Software (Stefan ten Hoedt and René Schreurs) we obtained the overall idea of the structure and the behavior of the software in general and the parts to be modeled in particular. The relevant parts were modeled in mCRL2 [1]. The session layer of the load-balancer protocol was modeled quite close to the C code. Both the higher-level application layer and the underlying TCP-socket layer were modeled in an abstract manner. The code and the model were reviewed by the LaQuSo-modeler and the Aia-developer in order to achieve the maximal matching. This led to a number of changes in the model, as well as to a number of questions about the code and a number of concrete desired properties that could be analyzed. The model was analyzed with the help of the model-checking techniques of the mCRL2 Toolset w.r.t deadlock-freedom and a number of other starvation and consistency properties that were formulated together with the client. This revealed 6 problems in the C code. These problems were accepted by Aia Software and incorporated to the production release of the software system.

The type of analysis presented in this paper is as such not new. It was performed before using different kinds of model checkers (e.g. imperative [2] and declarative [1]: see also the related work paragraph below). Noteworthy characteristics of our work are that the model is very close to the code, the code is relatively large (7500 lines), the code has been running within a commercial product for years and it has been improved several times while problems still kept occurring, errors have been found that led to code improvements and finally, problems regarding the code have not occurred since the code was corrected. This project was done with a model checker based on Process Algebra [3]. It is the first time that a project with such characteristics was achieved with a model checker based on Process Algebra.

Related Work Many projects study the verification of the *design* of a software system. Karl Palmskog in his Master Thesis [4] studied using the SPIN model checker the design of a Session Management Protocol developed at Ericsson Research. He has discovered a design flaw. This study was done on the level of the design without looking carefully at the implemented code. Also on the design level, in [5] He and Janicki present a verification of a Wireless Transaction Protocol design in SPIN. Another verification project concerning model checking of the design of a software system in mCRL2 is the parking garage project done by Mathijssen and Pretorius [6]. In [7] Brock and Jackson prove correctness of an industrial implementation of a 'fault tolerant computer' by creating a small abstract model in CSP.

A real-life *code* example was recently studied by Hessel and Pettersson [8] with nice results. In contrast to our project, they do not model the code but use a black-box testing approach.

In [9] an application of the Verisoft model checking approach to a software system from Lucent is presented. The model checking was applied as a part of the testing procedure during the software development. The paper reports about

a large number of revealed errors, most of which indicated incorrect variable initializations.

A framework for C code analysis with CADP [10] is presented in [11], where the methods of process graph extraction and generation of an LTS for a C program are described. In [12] a model checker MOPS was used to model-check safety properties of single-threaded C programs. This paper reports on automatic analysis of million lines of code.

The Java Pathfinder tool is described in [13] as a tool that is used to find deadlocks and other behavioral properties in java programs. The tool has been used to analyse software systems at NASA. It is also used as the back-end model checker of the Bandera project [14]. The Bandera project uses abstraction techniques based on abstraction-based program specialization: a combination of abstract interpretation and partial evaluation.

Research at Microsoft Corporation led by Thomas Ball has shown significant results for a restricted subset of programs: device drivers. Using an automatic analysis engine - called SLAM - that combines model checking with symbolic execution for the language C, they have successfully found many errors in many real-life industrial device drivers [15]. They do not support analysis of multi-threaded systems.

Probably, the most related work is performed by Holzmann and Smith in [16]. Using SPIN they followed the development of a piece of telephone call processing software of about 1600 lines of C code. They verified successfully so-called feature requirements. They found many errors in different stages of the development.

Organization of the paper The paper is organized as follows. Section 2 presents the case study and the problems that were to be investigated. Section 3 presents the mCRL2 language and the toolset and the way they were used in the modeling of the case study. Section 4 presents the details on the analysis with the mCRL2 toolset and the issues that were detected. Section 5 contains conclusions and possibilities for future work. In the Appendix a part of the C code and the corresponding part of the mCRL2 model are presented. The whole mCRL2 model can be found in the Appendix of [17].

2 Intelligent Text Processing (ITP) and its Load-Balancer

The Intelligent Text Processing system is used to prepare large quantities of documents to be printed. Sometimes it is done in an interactive way, where additional information is being asked from the client during the processing. In the early versions of the ITP software the *clients* could directly communicate to the *document processors*, but with the increased complexity of the processing jobs a coordinating mechanism was needed. The task of the *load-balancer* is to distribute the jobs of the clients to the available document processors, without actually changing the application layer of the client-server communication protocol too much (see Figure 1).

Due to the evolutionary way the ITP software was developed in the late nineties, the load-balancer has been implemented in C on the Windows platform

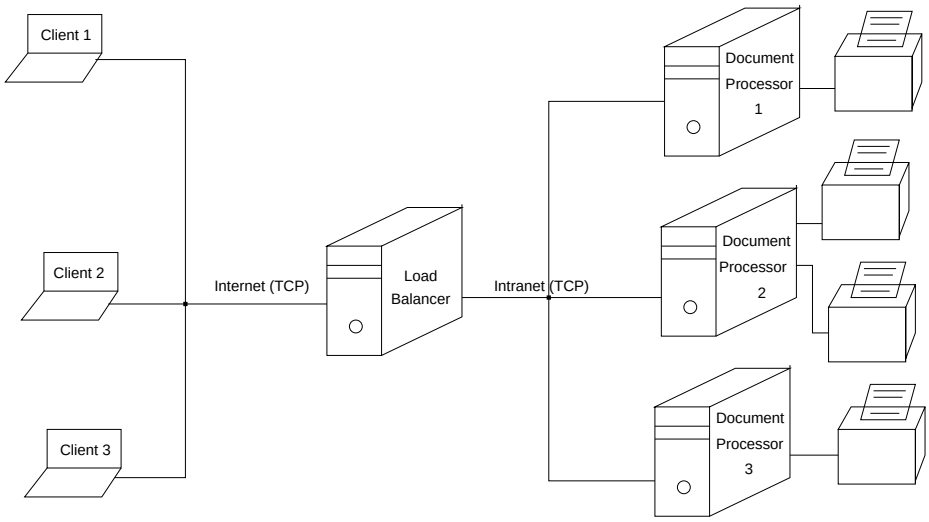

Fig. 1. ITP and a Load-Balancer in it.

making use of the Windows Socket Library. The possibility of using a standard solution for load-balancing, like the Linux virtual server, has not been used for a number of reasons.

A typical use-case scenario of the load-balancer deployment is presented as a Message Sequence Chart on Figure 2. There, a client of the load-balancer communicates with the client object and a document processor communicates with the document processor object. The client sends a request to print and the document processor sends a request for work. After that the document processor object asks the client object for work and gets the answer. At this point the client and the document processor objects are linked together by a partnership link. Further, the document processor asks for additional data and goes to a sleeping state. The client object gets the data from the client and wakes up the document processor object. The document processor object transfers the data to the document processor.

2.1 Issues and Artifacts

The load-balancer software was developed in the late nineties and has been tested both at AIA and at clients' environments since that time. The system has been in use in production for quite some time now. During testing and maintenance a number of issues with the software have been fixed, but some items remained unsolved till the beginning of our project.

Most of these 'difficult' issues could be classified as follows:

- the load-balancer would get to a state where it did not respond at all to the requests of neither clients nor document processors;
- the load-balancer would ignore the document processors that were free and willing to accept jobs;

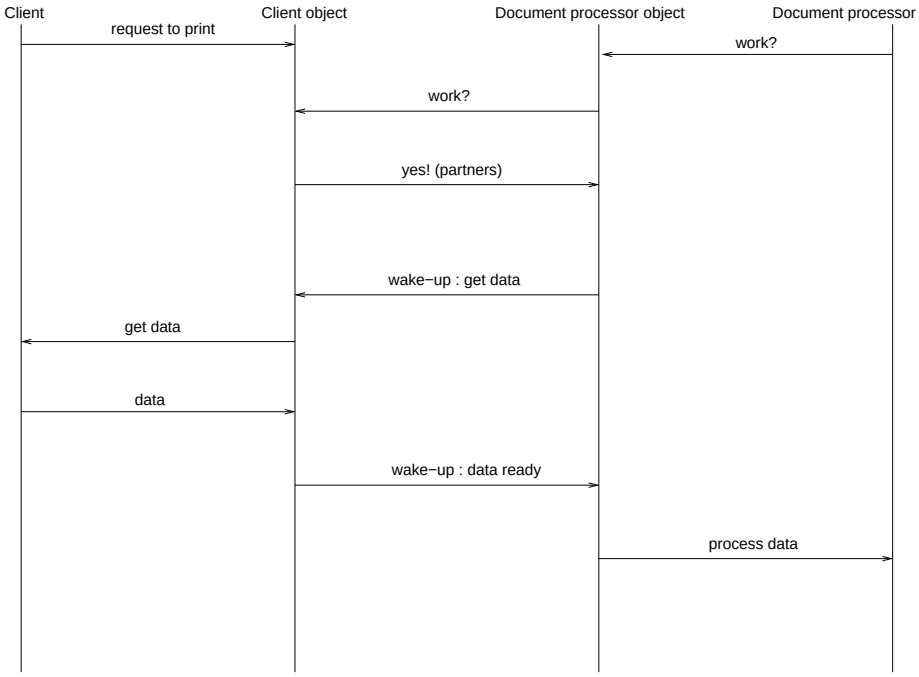

Fig. 2. A typical use-case scenario of the load-balancer.

– a client would not get any response from the load-balancer about the status of its jobs.

These issues occurred in rare situations, mostly on particular hardware configurations. Reproducing such errors was very difficult or impossible. Restarting the system solved the issue but it could occur again somewhere in the future.

The company provided the source code in C for windows (7681 lines) and the application layer protocol documentation. Further information was communicated during meetings, via phone calls and e-mail. Analysis of the artifacts revealed that the system was a multi-threaded Windows application using mutual exclusion primitives (mutexes, semaphores) and multiple event synchronization (WaitForMultipleObjects). For the asynchronous I/O and the network communication the Windows Socket Administration and call-back functions were used. The reverse engineering of the design revealed the structure of the load-balancer (see Figure 3). Here each client and document processor object has a request queue and a partnership link to a possible partner. Each such object implements a finite state machine that first waits for one of the two events, either a network socket event or a wake-up event from a partner. After that, a certain action is performed and the object proceeds to a new state.

Based on the source code and the revealed architecture of the load-balancer the following properties were considered to be important for the further analysis.

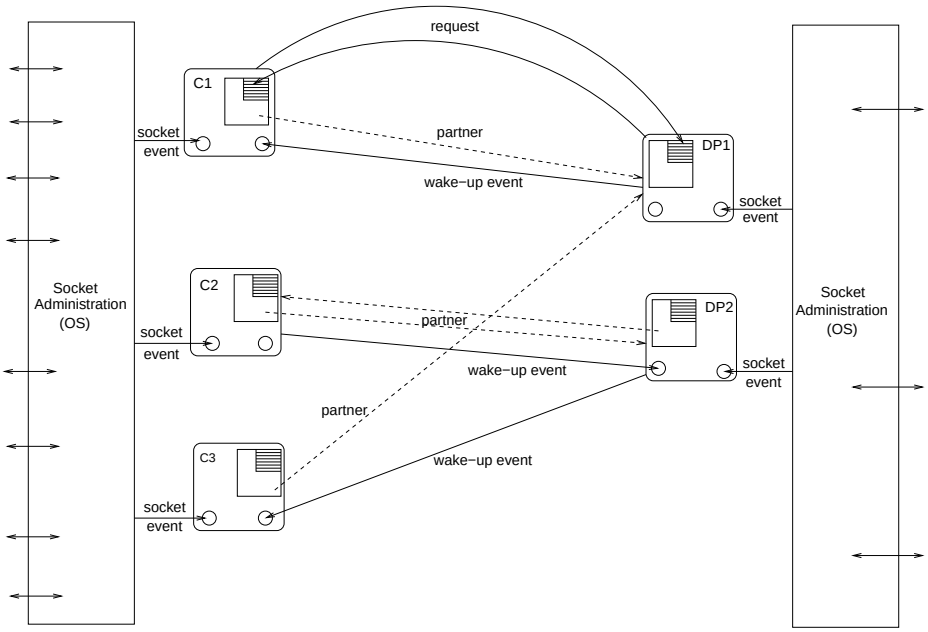

Fig. 3. Architecture of the load-balancer.

- The software should be free from *deadlocks*.
- Certain log messages are considered to be of critical importance. These should never occur as they indicate that there is something fundamentally wrong with the system.
- The partnership links should be consistent, e.g., if the partner of A is $B > 0$ (0 means no partner), then the partner of B is A or 0.
- Waiting for a partner should only be done if the partner link is not 0. This boils down to the fact that a document processor may not be in a sleeping state if it has no partner (except when a request is pending to it).
- The number of times a thread acquires a lock should be limited. In case a lock is acquired a multiple number of times it has to be released the same number of times. If a thread acquires a lock in a loop, a certain bound induced by the operating system can be reached, resulting in an undesired behavior. Moreover, a high number of nested lock acquisitions may indicate a logical error in the program.
- The number of requests that are pending in the system should be limited.

3 Modeling in mCRL2

To check the desired properties part of the system had to be formally modelled in a language that supports model-checking. For the reasons of available expertise we decided to use mCRL2 and its toolset.

3.1 Description of the mCRL2 language

mCRL2 [1] is a process algebraic language that includes data and time. It is an extension of the language μCRL [18] with multi-actions, built-in data types and local communication functions instead of a single global one. mCRL2 is basically intended to study description and analysis techniques for (large) distributed systems. The abbreviation mCRL2 stands for milli Common Representation Language 2.

An mCRL2 specification consists of two parts. The first part specifies the data types, the second part defines the processes. Data are represented as terms of some sort, for example `2`, `cos(pi)`, and `concat(L1,L2)` could be terms of sort natural number, real number and list, respectively.

The process equations are defined in the following way. Starting from a set Act of actions that can be parameterized with data, processes are defined by means of guarded recursive equations and the following operations.

First, there is a constant δ ($\delta \notin \mathsf{Act}$) that cannot perform any action and is called deadlock or inaction.

Next, there are the sequential composition operation $\cdot$ and the alternative composition operation $+$. The process $x \cdot y$ first behaves as x and if x successfully terminates continues to behave as y. The process $x + y$ can either do an action of x and continue to behave as x or do an action of y and continue to behave as y.

Interleaving parallelism is modeled by the operation $\|$. The process $x \,\|\, y$ is the result of interleaving actions of x and y, except that actions from x and y also synchronize to multiactions. So $\mathsf{a} \,\|\, \mathsf{b} = \mathsf{a} \cdot \mathsf{b} + \mathsf{b} \cdot \mathsf{a} + \mathsf{a} \mid \mathsf{b}$. The communication operation Γ allows multiactions to communicate: parameterized actions $\mathsf{a}(d)$ and $\mathsf{b}(d')$ in $\Gamma_{\{\mathsf{a}|\mathsf{b}\to\mathsf{c}\}}(\mathsf{a}(d) \mid \mathsf{b}(d'))$ communicate to $\mathsf{c}(d)$, provided $d = d'$.

To enforce that actions in processes x and y synchronize, we can prevent actions from happening on their own, using the encapsulation operator ∂_H. The process $\partial_H(x)$ can perform all actions of x except that actions in the set H are blocked. So, in $\partial_{\{a,b\}}(\Gamma_{\{\mathsf{a}|\mathsf{b}\to\mathsf{c}\}}(x \,\|\, y))$ the actions a and b are forced to synchronize to c. Another way to restrict process behaviour is the allow operation. By specifying a list of multiactions one can prohibit all other multiactions by renaming them to δ. So $\nabla_{\{\mathsf{a}|\mathsf{b}\}}(\mathsf{a} \,\|\, \mathsf{b}) = \mathsf{a} \mid \mathsf{b}$.

We assume the existence of a special action τ ($\tau \notin \mathsf{Act}$) that is internal and cannot be directly observed. The hiding operator τ_I renames the actions in the set I to τ. By hiding all internal communications of a process only the external actions remain.

The following two operators combine data with processes. The sum operator $\sum_{d:D} p(d)$ describes the process that can execute the process $p(d)$ for some value d selected from the sort D. The conditional operator $\_ \to \_ \diamond \_$ describes the *if-then-else*. The process $b \to x \diamond y$ (where b is a boolean) has the behavior of x if b is true and the behavior of y if b is false. The expression $b \to x$ is a syntactic sugar representing the *if-then* construction. It is an abbreviation to $b \to x \diamond \delta$.

3.2 The mCRL2 Toolset

The mCRL2 Toolset (`http://www.mcrl2.org`) has been developed at Technical University of Eindhoven to support formal reasoning about systems specified in mCRL2. It is based on term rewriting techniques and on formal transformation of process-algebraic and data terms. At the moment it allows to generate state spaces, search for deadlocks and particular actions, perform symbolic optimizations for mCRL2 specifications and simulate them.

The tool set is constructed around a restricted form of mCRL2, namely the Linear Process Specification (LPS) format. An LPS contains a single process definition of the *linear form*:

$$\texttt{proc}\ \ \mathsf{P}(x{:}D) = \sum_{i \in I} \sum_{y_i : E_i} c_i(x, y_i) \rightarrow \alpha_i(x, y_i) \cdot \mathsf{P}(g_i(x, y_i))$$

$$\texttt{init}\ \ \mathsf{P}(d_0);$$

where data expressions of the form $d(x_1, \ldots, x_n)$ contain at most free variables from $\{x_1, \ldots, x_n\}$, I is a finite index set, and for $i \in I$ the following are:

- $c_i(x, y_i)$ are boolean expressions representing the conditions,
- $\alpha_i(x, y_i)$ is a multiaction $a_i^1(f_i^1(x, y_i)) \mid \cdots \mid a_i^{n_i}(f_i^{n_i}(x, y_i))$, where $f_i^k(x, y_i)$ (for $1 \leq k \leq n_i$) are the parameters of action name a_i^k,
- $g_i(x, y_i)$ is an expression of sort D representing the next state of the process definition P;
- d_0 is a closed data expression;
- $\sum_{i \in I} p_i$ is a shorthand for $p_1 + \cdots + p_n$, where $I = \{1, \ldots, n\}$.

The form of the summand as described above is sometimes presented as the *condition-action-effect* rule. In a particular state d and for some data value e the multiaction $\alpha_i(d, e)$ can be done if condition $c_i(d, e)$ holds. The effect of the action on the state is given by the fact that the next state is $g_i(x, y_i)$.

The tool `mcrl22lps` checks whether a certain specification is a well formed mCRL2 and attempts to transform it into a linearized (i.e. LPS) form (See [19] for the detail of the linearization). All other tools use this linearized format as their starting point (see Figure 4).

These tools come in four kinds:

1. a tool (`xsim`) to step through the process specified in the LPS;
2. a tool (`lps2lts`) to generate the labeled transition system (LTS) underlying a given LPS;
3. several tools to optimize the LPSs:
   (a) `lpsrewr`, normalizes an LPS by rewriting the data terms in it;
   (b) `lpsconstelm`, removes data parameters that are constant throughout any run of the LPS;
   (c) `lpsparelm`, reduces the state space of the transition system by removing the data parameters and sum variables that do not influence the behavior of the system,
   (d) `lpsstructelm`, expands variables of compound data types;
4. a tool (`lpspp`) to print the linearized specification.

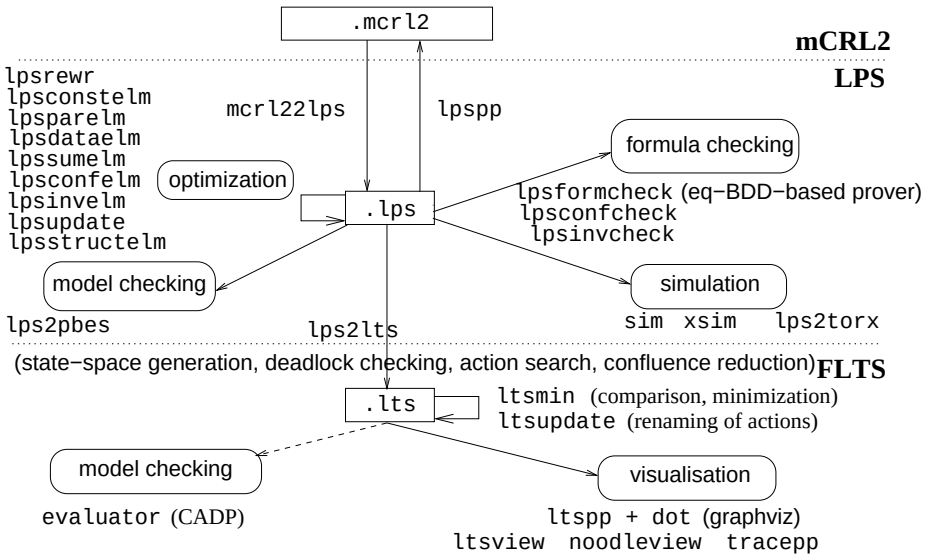

Fig. 4. The mCRL2 Toolset (www.mcrl2.org)

3.3 The Load-Balancer in mCRL2

For the modeling we concentrated on the session layer of the protocol. This layer is responsible for controlling the connections with the clients and the document processors, e.g., establishing, breaking the connection, handling non-expected connection breaks and network errors. Sending and receiving of data goes through this layer as well.

The lower-level interface (back-end) of the session layer protocol goes to the Windows Socket Administration (WSA) library. This library is a part of the operating system and is responsible for sending and accepting network socket events from the application. In our mCRL2 model WSA is modelled as a part of the environment.

The high-level interface (front-end) of the session layer performs calls to the application layer of the protocol. This happens when a certain part of data is received from a client or a document processor in a state when data is expected, or a connection is broken and this fact has to be noticed by the application layer (sometimes the session layer can close the session itself and no action from the application layer is required). The code of the application layer happens to be a rather large piece of homogeneous code, a large case distinction so to say. We modelled it by making an over-approximation of all possible behaviors and choosing them in a non-deterministic way. By doing this we ended up with less than ten alternatives for the application layer.

The model of the session layer follows the C code in a way to make it as precise as possible. The model resembles the request handling and the network events handling in most details, following the state-transition paradigm implemented in

the code. Appendix B presents a part of the mCRL2 models that corresponds to the request handling session layer part of the C-implementation in Appendix A. The model and the code in these appendices follow each other rather closely. The sizes of the two specifications are more or less the same.

The shared variables and arrays that are used for inter-thread communications are modelled by separate processes. Parts of the operating system are modelled by processes as well. Below an mCRL2 process for the mutual exclusion primitive of Windows (MSDN Mutex objects) is presented. A thread can acquire a mutex a multiple number of times and has to release it the same number of times.

$$
\begin{aligned}
&\mathsf{Lock}(owner{:}Nat, count{:}Nat) = \\
&\quad \sum_{tid:Pos} (owner == 0 \vee owner == tid) \rightarrow \mathsf{lock}(tid) \cdot \mathsf{Lock}(tid, count + 1) \\
&+ (owner > 0) \rightarrow \mathsf{unlock}(Nat2Pos(owner)) \cdot \\
&\qquad\qquad \mathsf{Lock}(if(count == 1, 0, owner), Int2Nat(count - 1)) \\
&+ (count > nMaxLock) \rightarrow \_\mathsf{error}(MaxLock) \cdot \delta;
\end{aligned}
$$

The process Lock has two natural numbers as parameters. The first one represents the id of the thread that owns the mutex, or is equal to 0 if the mutex is free. The second parameter is used to count how many times the mutex has been acquired.

The actions lock and unlock are parameterized by positive numbers representing the id of the locking/unlocking thread. Such a thread would perform a corresponding _lock or $_{\mathsf{u}}$nlock action parameterized with its id. The two corresponding actions (with and without the underscore) are then forced to synchronize by the process defining the entire system.

The first summand of the process Lock says that it can be acquired (by performing a _lock action) by a thread with its id represented by the variable *tid*. This is allowed for a thread with any id in case the mutex is free (condition $owner == 0$) or for the owner thread ($owner == tid$). After this acquisition the lock is owned by the thread identified by *tid* and the acquisition number counter is incremented.

The second summand says that a non-free mutex can be unlocked by the owner. Here we use *Nat2Pos* to cast the value of the natural variable *owner* to the positive number. This function maps 0 to 1 and any number bigger than 1 to itself. Given the condition $owner > 0$, this cast is always an identity mapping. The function *Int2Nat* is used to cast the integral value of $count - 1$ to the natural number. It maps the negative integers to 0 and does not change the non-negative integers. It can be shown that $owner > 0 \implies count > 0$ is and *invariant* of the Lock process. Therefore, this cast is also an identity mapping.

The third summand lets the process perform an _error action if the value of *count* reaches a certain limit *nMaxLock*. In this way, by checking for absence of _error actions, one can prove that the mutex is acquired in a nested way less than *nMaxLock* number of times.

3.4 Modeling the Properties

It turned out that all the desired properties (except for the deadlock absence) could be modeled as safety properties and checked by adding $_{\mathsf{e}}\mathsf{rror}$ actions to the model and check for them. For example, the partner consistency property from Section 2.1 is modelled as a summand in the SharedConnection process:

$$\sum_{cid:Nat} \sum_{n:Nat} (n \neq 0 \ \wedge \ getpartner(connections.n) \neq 0 \wedge getpartner(connections.n) \neq cid) \rightarrow \mathsf{setConnectionPartner}(cid, n) \cdot \_\mathsf{error}(WrongPartners) \cdot \delta$$

Here $getpartner(connections.n)$ gives the current partner link value for the connection n. Once an attempt to change the partner of connection cid to the value n is performed by one of the threads (by performing the corresponding _setConnectionPartner action with the actual parameters), the condition is checked and if it is true, the error action is enabled. The condition says that neither n nor the partner of connection n is 0 (meaning 'no partner') and the partner of n is not cid. The latter condition means the actual partnership link inconsistency between n and cid.

4 Analysis and Issues

The model has been analyzed for the absence of deadlocks and for validity of certain properties. These properties were incorporated in the model itself so that an _error action would occur if the property is violated. In this way the verification is performed by the explicit generation of the entire state-space and by looking for the _error actions and the deadlocks. Once one of this is found in a particular state, a minimal trace to this state gives a counterexample.

Performing the analysis takes only a few steps that can be activated from the command line. To give the reader an idea how this is done in practice, we give the actual commands with their actual parameters and options. As the first step, the linearization of the model takes place: with the command `mcrl22lps ITPpatched.mcrl2 ITPpatched.lps` that produces the linearized version of the model. Next, we apply the optimization steps on the LPS: `lpsrewr ITPpatched.lps | lpsconstelm > ITPpatched_opt.lps`. The actual generation of the transition system and checking for the properties is done with the command `lps2lts -vrDt -a _error -R jittyc ITPpatched_opt.lps` where the `-D` option enables deadlock checking and `-a _error` enables checking for _error actions. The `-t` option enables generation of trace files. In case a deadlock or an _error action is found, a trace file is generated with the shortest trace to that deadlock state or a state where the _error action is possible. The trace files can be printed out with `tracepp` or simulated in the `xsim` simulator.

4.1 Experiments and Results

The analysis has been performed by an exhaustive generation of the underlying state space using the mCRL2 Toolset. The experiments were carried out on a

computer with 2.6GHz 64 bit AMD CPUs and 128Gb RAM running Linux. The execution times and the resulting numbers of states and transitions are presented in Table 1. The mCRL2 state space generator uses the depth-first search algorithm (by default), and the levels are the levels of depth reached by performing the search. The cases with the total number of clients+document processors larger than 4 could not be fully analyzed.

clients	DPs	time	levels	states	transitions
1	1	7m 38s	237	368k	796k
1	2	1h 42m	365	9.8m	21m
2	1	4h 52m	442	28m	61m
1	3	36h	480	209m	455.6m
2	2	7d6h	550	1.5b	31.9b
3	1	9d3h	637	1.8b	38.9b

Table 1. Execution time (days, hours, minutes and seconds), number of levels, number of states and number of transitions (thousands, millions and billions) for different numbers of clients and document processors (DPs).

4.2 Detected Issues

An early analysis of the model revealed multiple modeling problems. After resolving these initial modeling problems, the model was compared with the original C code by both the modeler and the author of the C code working together. This revealed some essential difference between the code and the model. Once these differences between the code and the model were resolved, the mCRL2 tools were applied and the following issues were detected.

- **Issue 1.** In one case partner links were inconsistent. This was due to the fact that in one place in the C code the 'forward' partner link was set to 0 and the 'backward' one was forgotten. This piece of code was found 'unclear' during the model-code comparison activity, and later was confirmed to be erroneous by the mCRL2 toolset finding a shortest trace to the property violation.
- In two cases a document processor could end-up in a sleeping state without having a partner.
  - **Issue 2.** In one case this happened because the client's partner link was set to 0 before actually waking up the document processor (happened due to an earlier bug 'fix'). This problem was found by the model-code comparison and later confirmed by the mCRL2 Toolset.
  - **Issue 3.** In another case it was simply forgotten to wake-up the document processor. This problem can be clearly explained by a use-case scenario in Figure 5. This use-case scenario is similar to the one presented in Figure 2, with the difference that after sending a request for data to the client this client disconnects, instead of providing the actual data. This problem was found using the tools.
- It also happened that critical logs could occur in the program:

- **Issue 4.** A client could send a request to disconnect to itself in a wrong state, because changing of a state was forgotten;
- **Issue 5.** Request to wake up could lead to an inappropriate state change when a document processor was in the middle of a disconnection (found to be non-critical).

– **Issue 6.** The number of requests sent to a client could exceed the preset limit and could have possibly been unbounded. This happened when a document processor sent a request to disconnect to its partner client and did not break the partnership afterwards.

These issues were analyzed and accepted by Aia and led to modifications of the original C code. The corresponding modifications, fixing the problems mentioned above, were also brought into the model. The subsequent analysis of the model revealed no more property violations.

Most of the issues were detected in the case of 1 client and 1 document processor, while the rest in 1-2 or 2-1 situations. Analysis of the situations with more clients and document processors did not lead to detection of new issues.

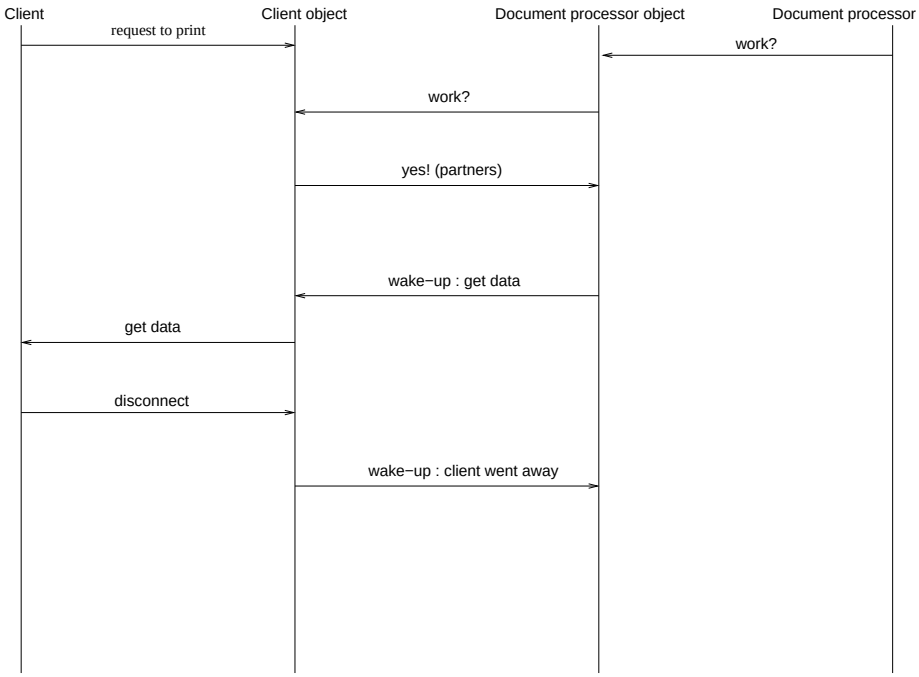

Fig. 5. A faulty scenario.

5 Conclusions and Future Work

We modelled the session layer of the ITP load-balancer in mCRL2 such that the model is close to the actual C code. A number of properties were verified using the mCRL2 tool set. This led to the discovery of 6 issues that were easily traced back to the actual C code. The code was repaired and also the corrections were brought into the model. The resulting model was verified with respect to the desired properties by checking the entire state space for several configurations.

mCRL2 could be used successfully in this industrial setting of a load-balancer for document production. A part of the operating system services (sockets, locks, events, etc.) could also be modeled. Unfortunately the verification could only be done on a restricted setting, so an improvement of the tool set is required for bigger cases. Also an automatic conformance checking of the model w.r.t. the code could be of interest.

Lessons Learned: The case study gave the researchers more confidence that real-life examples can actually be dealt with using a close-to-code model. It increases the motivation to further improve the power of the analysis tool and to start investigating code generation from the model (the proximity to the code may simplify code generation).

Aia released the new version with the improved code about half a year ago. While previously it happened now and then that their systems infrastructure came to a standstill and had to be restarted again, this situation never occurred anymore with the new release. The infrastructure (which has the load balancer as the most critical part) kept running all the time.

They have now a working reference model in mCRL2 of a crucial part of their load-balancer software. In principle, they are able to incorporate code changes into the model and check whether the properties still hold for the new version. In practice, they probably need assistance of the researchers in the beginning. Aia has acquired an increased interest in using formal models for analyzing software quality aspects, in particular for the most critical parts of their system.

Future Work: In the future an improvement of the tool set could lead to model checking of bigger cases. Analyzing more properties of the session layer (e.g. verifying client notification of document processor failures) could lead to certification of the software. If we want to improve the relation between the model and the code, we can consider code generation directly from the model.

References

1. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: The formal specification language mCRL2. In: Proc. Methods for Modelling Software Systems. Number 06351 in Dagstuhl Seminar Proceedings (2007)
2. Holzmann, G.J.: Software model checking with spin. Advances in Computers **65** (2005) 78–109
3. Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra. Elsevier (2001)

4. Palmskog, K.: Verification of the session management protocol. Master's thesis, School of Computer Science and Communication, Royal Institute of Technology, Stockholm (2006)
5. He, Y.T., Janicki, R.: Verifying protocols by model checking: a case study of the wireless application protocol and the model checker spin. In: CASCON '04: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research, IBM Press (2004) 174–188
6. Mathijssen, A., Pretorius, A.J.: Verified design of an automated parking garage. In Brim, L., Leucker, M., eds.: FMICS '06: 11th International Workshop on Formal Methods for Industrial Critical Systems. Volume 4346 of LNCS., Springer Verlag (2007) 165–180
7. Brock, N.A., Jackson, D.M.: Formal verification of a fault tolerant computer. In: Proc. 11th Digital Avionics Systems Conference (IEEE/AIAA). (1992) 132–137
8. Hessel, A., Pettersson, P.: Model-based testing of a wap gateway: An industrial case-study. In Brim, L., Leucker, M., eds.: FMICS '06: 11th International Workshop on Formal Methods for Industrial Critical Systems. Volume 4346 of LNCS., Springer Verlag (2007) 116–131
9. Chandra, S., Godefroid, P., Palm, C.: Software model checking in practice: an industrial case study. In: ICSE, ACM (2002) 431–441
10. Fernandez, J.C., Garavel, H., A. Kerbrat, R.M., Mounier, L., Sighireanu, M.: CADP: A protocol validation and verification toolbox. In: Proceedings of the 8th Conference on Computer-Aided Verification, New Brunswick, New Jersey, USA (August 1996) 437–440
11. del Mar Gallardo, M., Merino, P., Sanán, D.: Towards model checking c code with open/cæsar. In Barjis, J., Ultes-Nitsche, U., Augusto, J.C., eds.: MSVVEIS, INSTICC Press (2006) 198–201
12. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of c code. In: NDSS, The Internet Society (2004)
13. Visser, W., Mehlitz, P.C.: Model checking programs with java pathfinder. In Godefroid, P., ed.: SPIN. Volume 3639 of Lecture Notes in Computer Science., Springer (2005) 27
14. Iosif, R., Dwyer, M.B., Hatcliff, J.: Translating java for multiple model checkers: The bandera back-end. Formal Methods in System Design **26**(2) (2005) 137–180
15. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In Berbers, Y., Zwaenepoel, W., eds.: EuroSys, ACM (2006) 73–85
16. Holzmann, G.J., Smith, M.H.: Automating software feature verification. Bell Labs Technical Journal **5**(2) (- 2000) 72–87
17. van Eekelen, M., ten Hoedt, S., Schreurs, R., Usenko, Y.S.: Modeling and verifying a real-life industrial session-layer protocol in mCRL2. Technical Report ICIS-R07014, Radboud University Nijmegen (jun 2007)
18. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. In Ponse, A., Verhoef, C., van Vlijmen, S.F.M., eds.: Algebra of Communicating Processes 1994. Workshop in Computing, Springer (1995) 26–62
19. Usenko, Y.S.: Linearization in μCRL. PhD thesis, Eindhoven University of Technology (December 2002)

A Part of C code of the Request Handling

```
while (Interface->Request != (REQUEST *) NULL){
  REQUEST *Req = Interface->Request;
  DWORD ID = Req->Connection - Req->Connection->Interface->Connections;
  switch (Req->Request){
  case requestDisconnect:
    /* Partner requests a disconnect */
    if (Req->Connection->State != STATE_PENDING &&
        Req->Connection->State != STATE_SLEEP){
      if (Req->Connection->State == STATE_EVENT){
        CancelEvent (Req->Connection);
      } else if (Req->Connection->State != STATE_DISCONNECT&&
                 Req->Connection->State != STATE_BREAK     ){
        LogMessage (ClassError,
            L"Disconnect: Forcing illegal state switch %s->%s on socket %d",
                    ShowConnState(Req->Connection->State),
                    ShowConnState(STATE_DISCONNECT),
                    ID);
      } else {
        /* Our own connection was already shutting down. Just confirm it. */
      }
    }
    if (Req->Connection->State != STATE_BREAK){
      Req->Connection->State = STATE_DISCONNECT;
    }
    break;
  case requestSend:
  case requestReceive:
    if (Req->Connection->State != STATE_PENDING &&
        Req->Connection->State != STATE_SLEEP){
      CONNECTION *Partner;
      if (Req->Connection->State == STATE_BREAK     ||
          Req->Connection->State == STATE_DISCONNECT){
        /* Lost connection to client */
        LogMessage (ClassError,
                L"Remote host closed connection unexpectedly on socket %d.",
                    ID);
        /* Detach our connection */
      } else {
        LogMessage (ClassError,
          L"Send/Receive: Forcing illegal state switch %s->%s on socket %d",
                    ShowConnState(Req->Connection->State),
                    ShowConnState(STATE_TRANSACTION),
                    ID);
      }
      /* Remove our link to the partner */
      WaitHandle (PartnerLock);
      Partner = Req->Connection->Partner;
      Req->Connection->Partner = (CONNECTION *) NULL;

      /* Wake the partner */
      if (Partner != (CONNECTION *) NULL){
        if (Partner->Partner == Req->Connection){
          Partner->Partner = (CONNECTION *) NULL;
        }
        WakeConnection (Partner);
      }
      ReleaseMutex (PartnerLock);
      /* And close our socket */
      if (Req->Connection->State != STATE_BREAK){
        Req->Connection->State = STATE_DISCONNECT;
      }
      break;
    }
    /* Start the requested operation */
    Req->Connection->State = STATE_TRANSACTION;
```

```
66      Req->Connection->Protocol = Req->NewState;
        Req->Connection->Read = (Req->Request == requestReceive);
        Req->Connection->Write = (Req->Request == requestSend);
        Req->Connection->Size = Req->Size;
        Req->Connection->Buffer = Req->Data;
71      break;
      case requestWakeUp:
        /* Our partner finished its operations and tries to wake us up. */
        if (Req->Connection->State == STATE_TRANSACTION){
          /*
76         * We are already awake and handling transactions.
           * Don't change anything.
           */
        } else if (Req->Connection->State != STATE_PENDING    &&
                   Req->Connection->State != STATE_SLEEP){
81        /* Detach our connection */
          LogMessage (ClassError,
                   L"Wake up: Forcing illegal state switch %s->%s on socket %d",
                      ShowConnState(Req->Connection->State),
                      ShowConnState(STATE_TRANSACTION),
86                    ID);
        } else {
          Req->Connection->State = STATE_TRANSACTION;
          Req->Connection->Read = FALSE;
          Req->Connection->Write = FALSE;
91      }
        break;
      default:
        LogMessage (ClassError, L"INTERNAL ERROR: State %d.", Req->Request);
        break;
96    }
      Interface->Request = Req->Next;
      Free (Req);

      /* Reset event flag so we won't delay processing the requests */
101   SetEvent (Interface->Pending);
    }
```

B Corresponding Part of the mCRL2 Model

```
1   TCP_ProcessRequests(tid:Pos,pending:Bool,nConns:Nat)=
      sum reqs:List(REQUEST).
        _getRequests(tid,reqs).
        (reqs==[])->_unlockPartner(tid).
                    (pending->_setPendingEvent(tid).
6                             TCP_WaitEvent(tid,nConns)
                            <>TCP_WaitEvent(tid,nConns)
                    )
                   <>_popRequest(tid).
                     TCP_ProcessRequest(tid,head(reqs),nConns);
11
    TCP_ProcessRequest(tid:Pos,req:REQUEST,nConns:Nat)=
      % first we need to get the state of the connection in the request:
      sum state:STATE._getConnectionState(tid,getcid(req),state).(

16      ( (getname(req)==requestDisconnect &&
           (state==STATE_BREAK ||
            state==STATE_DISCONNECT)
        )||
        (getname(req)==requestWakeUp &&
21       (state==STATE_TRANSACTION ||
          state==STATE_DISCONNECT ||
          state==STATE_BREAK)
        )
      )-> TCP_ProcessRequests(tid,true,nConns)<> % do nothing in these cases
26
        (getname(req)==requestDisconnect)->(
```

```
          (state==STATE_PENDING ||
           state==STATE_SLEEP)
           -> _setConnectionState(tid,getcid(req),STATE_DISCONNECT).
              TCP_ProcessRequests(tid,true,nConns)<>

          % otherwise log and force.
          (state==SOCK_FREE ||
           state==SOCK_ACCEPT ||
           state==SOCK_READING ||
           state==SOCK_WRITING ||
           state==SOCK_SHUTDOWN ||
           state==STATE_TRANSACTION)
           -> _log(tid,LogDisconnectForsingIllegalStateSwitch(getcid(req),
                                                   STATE_DISCONNECT)).
              error(CriticalLog).
              _setConnectionState(tid,getcid(req),STATE_DISCONNECT).
              TCP_ProcessRequests(tid,true,nConns)
       )<>

       (getname(req)==requestSend ||
        getname(req)==requestReceive)->(
          (state==STATE_PENDING ||
           state==STATE_SLEEP)
           ->  _setConnectionStateProtocolReadWrite(
                   tid,
                   getcid(req),
                   STATE_TRANSACTION,
                   getnewprotocol(req),
                   getname(req)==requestReceive,
                   getname(req)==requestSend).
              TCP_ProcessRequests(tid,true,nConns)+
          (state==STATE_BREAK ||
           state==STATE_DISCONNECT)
           -> _log(tid,LogRemoteHostClosedUnexpectedly(getcid(req))).
              TCP_ProcessRequest_Close(tid,getcid(req),nConns)+

          (state==STATE_EVENT ||
           state==SOCK_FREE ||
           state==SOCK_ACCEPT ||
           state==SOCK_READING ||
           state==SOCK_WRITING ||
           state==SOCK_SHUTDOWN ||
           state==STATE_TRANSACTION)
           -> _log(tid,LogSendReceiveForsingIllegalStateSwitch(getcid(req),
                                                   STATE_TRANSACTION)).
              error(CriticalLog).
              TCP_ProcessRequest_Close(tid,getcid(req),nConns)
       )<>

       (getname(req)==requestWakeUp)->(
          (state==STATE_PENDING ||
           state==STATE_SLEEP)
           -> _setConnectionStateReadWrite(tid,getcid(req),
                                           STATE_TRANSACTION,false,false).
              TCP_ProcessRequests(tid,true,nConns)<>
          (state==STATE_BREAK ||
           state==STATE_EVENT ||
           state==SOCK_FREE ||
           state==SOCK_ACCEPT ||
           state==SOCK_READING ||
           state==SOCK_WRITING ||
           state==SOCK_SHUTDOWN)
           ->_log(tid,LogWakeUpForsingIllegalStateSwitch(getcid(req),state)).
              error(CriticalLog).
              TCP_ProcessRequests(tid,true,nConns)
       )
   );
```