Towards Proving Preservation of Behaviour of Refactoring of UML Models

Marc van Kempen[1]
Derrick Kourie[2]
Michel Chaudron[3]
Andrew Boake[4]

Eindhoven University of Technology marc@bowtie.nl
University of Pretoria dkourie@cs.up.ac.za
Eindhoven University of Technology m.r.v.chaudron@tue.nl
University of Pretoria andrew.boake@up.ac.za

Abstract. Refactoring of a design before updating and modifying software has become an accepted practice in order to prepare the design for the upcoming changes. This paper describes a refactoring of the design of a particular application to illustrate a suggested approach. In this approach, we represent the design using UML, more specifically the structure using class diagrams, and the behaviour of each class using statecharts.
Examining metrics of the specific design, we suggest a refactoring that changes a centralized control structure into one that employs more delegation, showing the refactored class and statechart diagrams. As preserving behaviour is one of the defining attributes of a refactoring, we use a csp-based formalism to describe the refactoring, and show that the refactoring indeed preserves behaviour.
Keywords: UML, Restructuring, Refactoring, Statechart, Behaviour, Process Algebra, CSP

1 Introduction

Software in a real-world environment keeps evolving, as detailed requirements become clearer, and new requirements emerge. The software evolves by being modified, having new functionality added or removed. This often results in the internal structure of the software becoming more complex and moving away from the original design. At a certain point it becomes desirable to reorganize (restructure) the code in order to make the software more maintainable, readable and extendable. Through experience, it has become clear that before adding new functionality, it is advisable to clean up the existing design in order to prepare it for the coming changes. This process has become known as restructuring [Arn86], or better known as refactoring [Opd92,Fow99] for object-oriented software. The defining feature of a refactoring is that the overall behaviour of the system must be preserved.

As UML has become a de facto standard as a choice for design language, we will focus on refactoring UML models. This area of research is relatively new and although a significant amount of refactorings have been defined that operate on a source code level, few have been described that operate on UML models. One example of UML refactoring research is the paper written by Sunyé et al., describing several (primitive) UML refactoring operations [SPTJ01].

To prove behaviour preservation several methods can be considered [MT04]. One way is using preconditions expressed in first-order predicate logic. This is a rather conservative approach however that rules out many legal refactorings. Mens et al. suggest a graph transformation formalism, used to check certain aspects of behaviour preservation [MDJ02]. Another approach is to use type checking [TKB03]. All typed software entities should still have the same type after the refactoring. These however all operate on source code for

the most part. Since we will focus on refactoring UML models we will need to employ a different method to prove behaviour preservation.

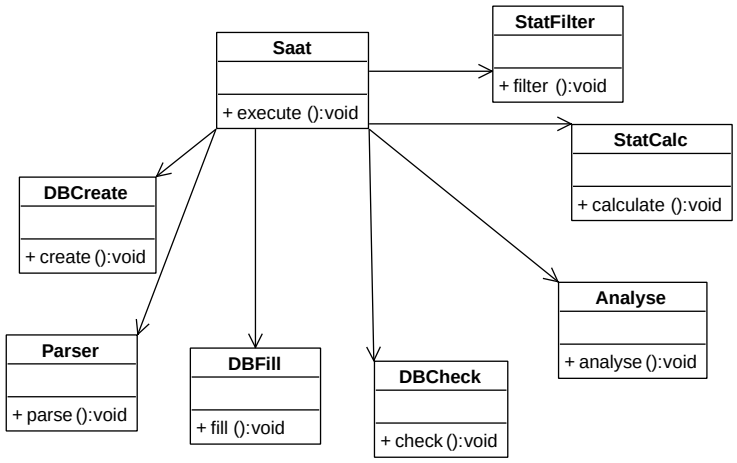
In this paper we describe the refactoring of a real-world application in the form of a case study illustrating our proposed approach. We describe a method, using statecharts and CSP, to prove behaviour preservation, a vital aspect of refactoring. The paper is organized as follows: The next section introduces the Saat case study and shows the UML design before and after the refactoring is applied. Section 3 describes the refactoring needed for the behaviour of the application. Section 4 describes the mapping of statecharts to CSP and the method used to prove that the refactoring preserves the behaviour of the system. Finally, we conclude the paper with a few observations.

2 Saat

The program that our case-study is based on is Saat (Software Architecture Analysis Tool) [MCL04]. Saat is a tool used to calculate metrics about UML models. These metrics can then be used in analysing the model for potential flaws or anti-patterns. As an exercise the Saat tool was used to calculate metrics about the Saat tool itself.

The UML Class-diagram for SAAT was initially drawn as can be seen in figure 1. The associations that are used indicate a call relationship, i.e. if module A has an association with module B, module A will call a method in module B.

Fig. 1. Saat Class diagram before refactoring



Saat essentially calls the methods in the associated modules sequentially. First the database is created (DBCreate.create()), the (XMI) input file is parsed (Parser.parse()) and the parsed data is inserted into the database (DBFill.fill()). After the data is inserted first it is checked (DBCheck.check()), after that the data is analysed (Analyse.analyse()), statistics are calculated (StatCalc.calculate()) and the resulting statistics are filtered according to user defined criteria.

After evaluating the architecture with SAAT the following metrics were found (only the most relevant ones are shown), see table 1. Typically high values for dynamic coupling and method call metrics indicate potential future problems regarding maintainability. Upon further investigation it was concluded that the Saat module was a so called “God-class”[SW00].

Table 1. Metrics for the SAAT architecture before refactoring

	Coupling	Dynamic Coupling	Method call
Saat	7	10	25
StatFilter	0	3	0
…	…	…	…

A “God-class”, also known as “The Blob” distinguishes itself by having a single class with many attributes and/or operations. It can be characterized as a “Controller class” with simple, data-object classes and it lacks object oriented design.

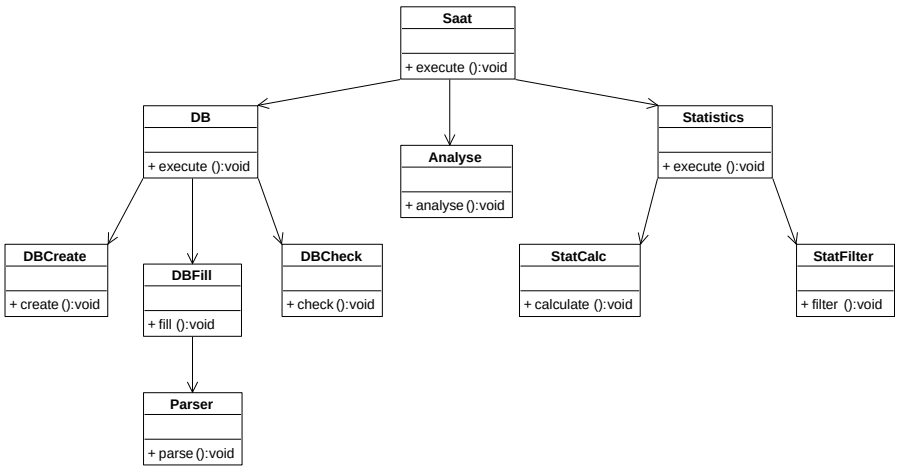
In this case we have a „Controller class“. The Saat class sequentially calls a number of methods (services) in other classes.

A “God-class typically has an impact on modifiability, maintainability and performance.

The design is refactored by introducing two new classes, DB and Statistics. The database operations (DBCreate.create(), Parser.parse(), DBFill.fill() and DBCheck.check()) will be delegated to the DB class. The statistics operations (StatCalc.calculate() and StatFilter.filter()) will be delegated to the Statistics class. The architect then modified the architecture as can be seen in figure 2.

The refactoring described here is similar to “Convert Procedural Design to Objects”, see [Fow99].

Fig. 2. Saat class diagram after refactoring



3 Statechart refactoring

In order to be able to reason about the behaviour it first needs to be defined. This will be done using statecharts in the following manner:

1. Every class in the UML model will have its own statechart, defining its behaviour.
2. Method calls on one object from another will be modeled by Call Events. Return from the method call is modeled by using a signal event in a systematic way.

3. Execution of the body of the method is implied by the semantics of the Call Event [Gro03].
   If so desired the behaviour of the method being called can also be defined using a statechart. We will not do so in this example.

For example, say class A wants to execute a method $m()$ in class B, the statechart for class A will send the event $m()$ after which it will wait for the return event m_r (the return event), the initial transition in the statechart for class B will wait for $m()$ to happen, upon reception of this event, the body of the method will execute, do whatever else is needed and then send m_r. Finally the statechart for class B will wait for the call event again.

The statechart will then be mapped to CSP notation and it will be shown that the processes before and after the refactoring (while hiding the method calls themselves) are identical in the method body executions and the order thereof by comparing their respective traces.

Using the heuristics described above we construct a statechart defining the behaviour for the Saat module before the refactoring, as can be seen in figure 3 on the facing page, and the statechart for one of the submodules (DBCreate) in figure 4 on the next page. All other submodules have a similar structure.

The notation for the statecharts is as follows:

- "Object.method()" is used to indicate a call event. The transition waits for the corresponding event. The semantics are that the transition is taken and the corresponding method is executed.
- The notation "^signal_name" is used to transmit an event by name. In the form "^Object.method()" it can be used to transmit a call event.
- The format for a transition label is as follows:
  event_name[guard expression]/action1; action2; ...;^event1,event2,...
  The transition waits for the event with the name "event_name". If the event occurs *and* the guard expression evaluates to true, the transition is taken, the actions are executed, and the events are transmitted.

After applying the refactoring, two classes have been introduced, and therefore we need two new statecharts: one for the class DB, see figure 6 and one for the class Statistics, see figure 7. The method DB.process() will be called from Saat, and the DB class will then call the methods from the classes whose associations have been moved to the DB class. The statechart for the Statistics class is implemented in a similar manner. As calling several methods has been delegated to DB and Statistics respectively, we also need to change the statechart corresponding to the Saat class. See figure 5 for the refactored statechart corresponding to the Saat class.

4 Mapping statecharts to CSP

We will now show how to map the statecharts to corresponding CSP constructs [Hoa85].

Module A has a (call method) association with Module B. The behaviour of Module A and Module B is defined by processes P and Q.

Assume that process P will call method m in Process Q. Name the request channel q, name the return channel q_r. Execution of the method $m()$ is indicated by the occurence of an identically named event.

Recall that in CSP, events are considered to be atomic. As a result, the foregoing assumes that any other event either occurs before or after the event $m()$. This is fine in the present context where concurrency is not being considered. However, if we needed to model the possible concurrent execution of two or more methods, then $m()$ could be replaced by two sequential events such as $StartExecution_{m()} \rightarrow FinishExecution_{m()}$. This would

Fig. 3. Saat statechart before refactoring and DBCreate statechart

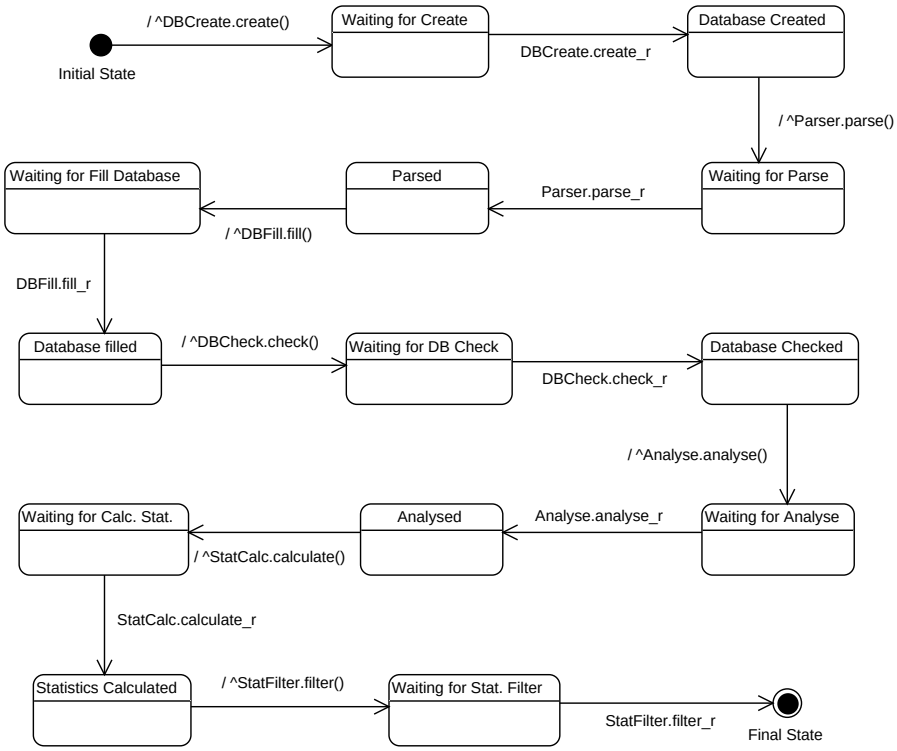


Fig. 4. DBCreate statechart

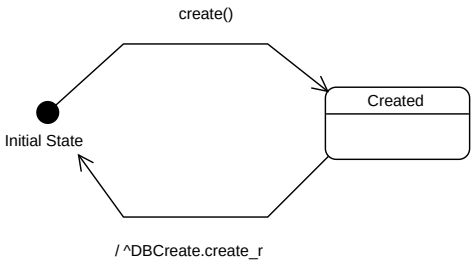


Fig. 5. Saat Statechart after refactoring

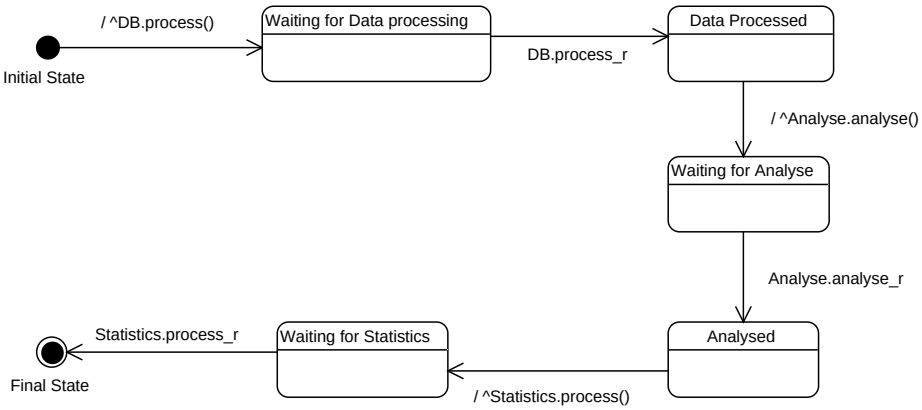


Fig. 6. DB Statechart, introduced after refactoring

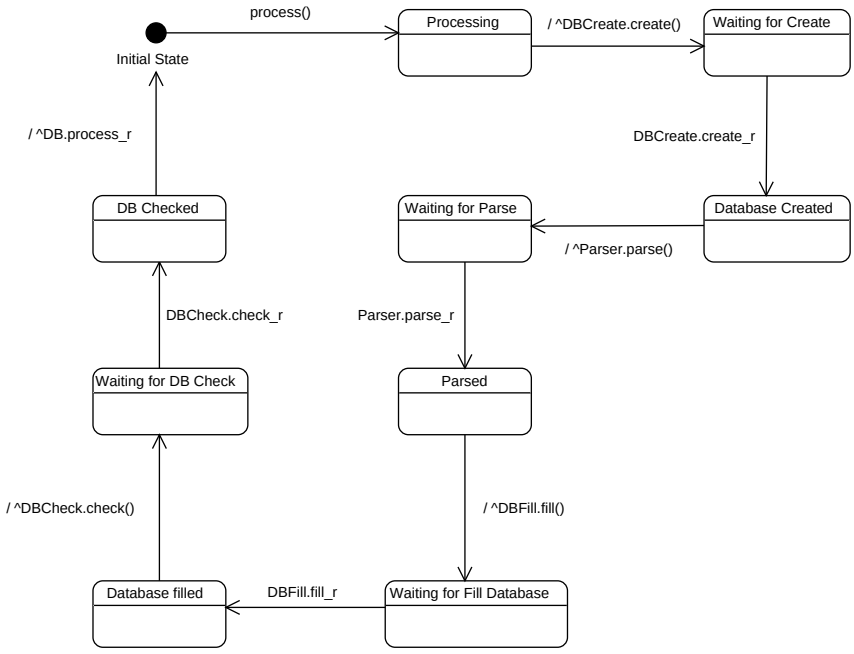
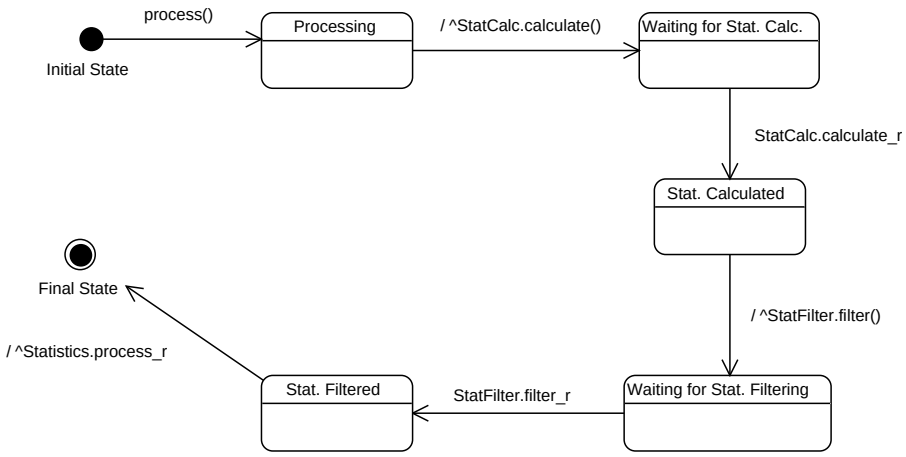


Fig. 7. Statistics Statechart, introduced after refactoring



allow for the start and finish events to be arbitrarily interleaved with other events, thus modeling concurency.

In the interest of clarity we will use the event $m()$ to indicate execution of the method $m()$.

$P = (q!m \rightarrow q\_r?x \rightarrow \ldots)$

Q waits for a request to execute method m, executes it and waits again.

$Q = (q?x \rightarrow m() \rightarrow q\_r!m \rightarrow Q)$

Further

$\alpha q(P) = \alpha q(Q) = \alpha q\_r(P) = \alpha q\_r(Q) = A = \{m\}$

For the events $q?x$ and $q\_r?x$, x must be such that $x \in A$.

Generally, therefore, the receiver can anticipate receiving any one of the events that are in A. However, in the present context where A is a singleton set, the receiver can rely on the message being the only member of the set.

4.1 Mapping Saat to CSP before refactoring

In the case of Saat the mapping to CSP is as follows:

We define channels for sending a call request message and for sending a return from call message. For the modules the following channels are defined. We keep the naming short in order to keep the definitions manageable.

Module name	Process name	Call request channel	Call return channel
Saat	Saat	n.a.	n.a.
DBCreate	DBCreate	cr	cr_r
Parser	Parser	p	p_r
DBFill	DBFill	f	f_r
DBCheck	DBCheck	ch	ch_r
Analyse	Analyse	a	a_r
StatGen	StatGen	sg	sg_r
StatFilter	StatFilter	sf	sf_r

This gives us the following process definitions. It should be clear that the CSP processes correspond to the statecharts shown earlier. We do not need a check for the correct message

at the receiving end, as per definition of the alphabet for the channels, only one message can be sent.

$$
\begin{aligned}
Saat &= (cr!create \rightarrow cr\_r?x \rightarrow p!parse \rightarrow p\_r?x \rightarrow f!fill \rightarrow f\_r?x \rightarrow \\
&\quad ch!check \rightarrow ch\_r?x \rightarrow a!analyse \rightarrow a\_r?x \rightarrow \\
&\quad sg!generate \rightarrow sg\_r?x \rightarrow sf!filter \rightarrow sf?x \rightarrow SKIP) \\
DBCreate &= (cr?x \rightarrow create() \rightarrow cr\_r!create \rightarrow DBCreate) \\
Parser &= (p?x \rightarrow parse() \rightarrow p\_r!parse \rightarrow Parser) \\
DBFill &= (f?x \rightarrow fill() \rightarrow f\_r!fill \rightarrow DBFill) \\
DBCheck &= (ch?x \rightarrow check() \rightarrow ch\_r!check \rightarrow DBCheck) \\
Analyse &= (a?x \rightarrow analyse() \rightarrow a\_r!analyse \rightarrow Analyse) \\
StatGen &= (sg?x \rightarrow generate() \rightarrow sg\_r!generate \rightarrow StatGen) \\
StatFilter &= (sf?x \rightarrow filter() \rightarrow sf\_r!filter \rightarrow StatFilter)
\end{aligned}
$$

The process S corresponding to the execution of the system is the concurrent execution of the processes:

$$
\begin{aligned}
S = Saat||DBCreate||Parser||DBFill||DBCheck|| \\
Analyse||StatGen||StatFilter
\end{aligned}
$$

4.2 Mapping Saat to CSP after refactoring

The refactoring introduces two new classes, DB and Statistics, for which the channels and processes are named as follows:

Module name	Process name	Call request channel	Call return channel
DB	DB	db	db_r
Statistics	Stat	st	st_r

The processes for the refactored model are defined as follows. Only the modified or added processes are shown. The others have not changed.

$$
\begin{aligned}
Saat' &= (db!process \rightarrow db\_r?x \rightarrow a!analyse \rightarrow a\_r?x \rightarrow \\
&\quad st!process \rightarrow st\_r?x \rightarrow SKIP) \\
DB &= (db?x \rightarrow cr!create \rightarrow cr\_r?x \rightarrow p!parse \rightarrow p\_r?x \rightarrow \\
&\quad f!fill \rightarrow f\_r?x \rightarrow ch!check) \rightarrow ch\_r?x \rightarrow db\_r!process \rightarrow DB) \\
Stat &= (st?x \rightarrow sg!generate \rightarrow sg\_r?x \rightarrow sf!filter \rightarrow sf\_r?x \rightarrow \\
&\quad st\_r!process \rightarrow Stat)
\end{aligned}
$$

The process S' corresponding to the execution of the refactored system is the concurrent execution of the former processes and the two new ones:

$$
\begin{aligned}
S' = Saat'||DB||Stat||DBCreate||Parser||DBFill||DBCheck|| \\
Analyse||StatGen||StatFilter
\end{aligned}
$$

Note that the introduction of the class DB, and $Statistics$ and the corresponding $DB.process()$, and $Stat.process()$ methods does not lead to the occurence of the corresponding method execution events in the CSP definition. This is because the execution of those methods is explicitely modeled by the events to execute the methods in the associated objects.

4.3 Behaviour

Showing that the behaviour of the model before and after the refactoring is equivalent will be accomplished by comparing the tracesets for the corresponding CSP processes S and S'.

Both processes produce a trace set with only one trace:
Trace set for S:

$$\begin{aligned} traces(S) = \{ < & cr!create, cr?x, create(), cr\_r!create, cr\_r?x, \\ & p!parse, p?x, parse(), p\_r!parse, p\_r?x, \\ & f!fill, f?x, fill(), f\_r!fill, f\_r?x, \\ & ch!check, ch?x, check(), ch\_r!check, ch\_r?x, \\ & a!analyse, a?x, analyse(), a\_r!analyse, a\_r?x, \\ & sg!generate, sg?x, generate(), sg\_r?x, sg\_r?x, \\ & sf!filter, sf?x, filter(), sf\_r!filter, sf?x > \} \end{aligned}$$

Trace set for S':

$$\begin{aligned} traces(S') = \{ < & db!process, db?x, \\ & cr!create, cr?x, create(), cr\_r!create, cr\_r?x, \\ & p!parse, p?x, parse(), p\_r!parse, p\_r?x, \\ & f!fill, f?x, fill(), f\_r!fill, f\_r?x, \\ & ch!check, ch?x, check(), ch\_r!check, ch\_r?x, \\ & db\_r!process, db\_r?x, \\ & a!analyse, a?x, analyse(), a\_r!analyse, a\_r?x, \\ & st!process, st?x, \\ & sg!generate, sg?x, generate(), sg\_r!generate, sg\_r?x, \\ & sf!filter, sf?x, filter(), sf\_r!filter, sf\_r?x, \\ & st\_r!process, st\_r?x > \} \end{aligned}$$

The determining factor for behaviour equivalence here is

1. Having the same method calls in both traces,
2. Having the same order of execution of the method bodies.

Equivalence can thus be examined by hiding all method call events and method call return events. We define a set C containing all method call events and method call return events, and we will employ the hide operator ('\') to hide these. This yields the following two trace sets:

$$\begin{aligned} traces(S)\backslash C = traces(S')\backslash C = \{ < & create(), parse(), fill(), check(), analyse(), \\ & generate(), filter() > \} \end{aligned}$$

As the traces are now identical we conclude that the behaviour of the two processes is equivalent and therefore that the refactoring preserved the behaviour of the system.

Note that this does not take into account behaviour preservation when real-time requirements are enforced, as timing aspects are not considered.

5 Conclusion

In this paper we have introduced and described a refactoring of a UML model. This specific refactoring adds more delegation and improves object oriented design and maintainability. We also showed a method using statecharts and CSP to prove behaviour equivalence. This method involved mapping statechart constructs to equivalent CSP.

Future work will need to focus on defining more, UML specific, refactorings. As these refactorings operate on more complex designs, comparing trace sets can become more difficult. A possible solution would be to define (primitive) statechart refactoring operations that are each behaviour preserving themselves. This will remove the need to compare tracesets and thus also address the problems associated with comparing tracesets of more complicated designs.

6 Acknowledgements

We wish to thank Prof. Judith Bishop of the University of Pretoria, for providing us with valuable feedback.

References

[Arn86] R.S. Arnold. An introduction to software restructuring. *Tutorial on Software Restructuring*, 1986.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Gro03] Object Management Group. Omg unified modeling language specification march 2003, version 1.5, March 2003.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[MCL04] J. Muskens, M.R.V. Chaudron, and C.F.J. Lange. Investigations in applying metrics to multi-view architecture models. In *EUROMICRO'04, Rennes, France*, September 2004.

[MDJ02] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proceedings of the First International Conference on Graph Transformation*, pages 286–301. Springer-Verlag, 2002.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[Opd92] W.F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[SPTJ01] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[SW00] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings 2nd International Workshop on Software and Performance, Sept. 2000*. Software Engineering Research and L&S Computer Technology, Inc., 2000.

[TKB03] Frank Tip, Adam Kieżun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, pages 13–26, Anaheim, CA, USA, November 6–8, 2003.