Model checking business processes

The search for the most compatible model checker

Sam van Dijk

University of Groningen
Faculty of Natural Sciences
Industrial Engineering & Management, Bachelor Thesis

Supervisor: prof. dr. ir. M. Aiello
Daily supervisors: H. Groefsema, MSc. and Dr. N.R.T.P. van Beest
24-01-2014

Student number 1877151

Abstract

Companies nowadays have to adapt to a changing market. As a result, business processes are changed in order to meet customer demands. However, these still need to comply with the requirements. A compliance check to verify that the requirements are followed is often done afterwards, when a change has already occurred. In this way, errors that are made can only be rolled back, but not prevented. At the Johann Bernoulli Institute for Mathematics and Computer Science, Heerko Groefsema and Nick van Beest have formulated a methodology in which a changed process can be checked for compliance prior to the execution of the change.

The methodology consists of two parts. First, the business process is converted into an abstract model representing the business process, and its requirements into a specification. Thereafter, a model checker (a computer program) checks the model against its specifications. In this research a number of model checkers have been compared in order to come up with a model checker that satisfies all or at least most of the requirements the authors have.

In order to do this, a ranking system has been established, in which model checkers receive a score based on how well they match the requirements. The model checker that earned the highest score, and therefore satisfied more requirements then the other model checkers, was MCheck. With the use of the mapping described in this report, the models produced by Groefsema and Van Beest can be tested with MCheck.

Contents

Introduction

Companies nowadays have to r eact to an ever changing market. Where Ford could still say: ‘Any customer can have a car painted any colour that he wants so long as it is black’ (Ford, 1922); flexibility and variability are the words in the 21st century (Groefsema, Bulanov & Aiello, 2012). However, implementing flexibility and variability is not an easy task. The business processes need to change, while they still have to comply with a set of requirements. Therefore, companies, in every industry, make models of their business processes. With the help of these models, it becomes easier to check whether a changed process still complies with the requirements. Most of the time, a compliance check to verify that the requirements and rules are followed is done afterwards, when a change has already occurred. In this way, errors that are made are only ascertained, but not prevented.

Ascertaining errors is of course important, but preventing errors can save a lot of money. This is one of the basic ideas of lean manufacturing and is also discussed in software engineering with Boehm’s first law (Boehm, 1975). Lean manufacturing is all about reducing waste and Boehm’s law explains that errors are more and more costly the later they are removed. Therefore both focuses on doing work properly the first time, instead of working faster, but with the need to perform repairs afterwards. The idea of lean manufacturing is becoming more and more known in today’s business, but is sometimes overlooked when talking about changing business process. As described above, the compliance check is done afterwards, what will make it possible to repair the errors that are already made. When one could perform the compliance check already prior to the change, a lot of waste can be prevented.

At the Johann Bernoulli Institute for Mathematics and Computer Science, Heerko Groefsema and Nick van Beest are working on exactly this problem. They have formulated a methodology in which a changed process can be checked for compliance prior to the execution of the change. With this methodology a business will be able to prove a process will still comply with the requirements after a certain change. With the help of this methodology companies will be able to check, in advance, if they can change their business process in such way that they can meet a new demand from the changing market. When they are able, nothing really different happens when compared to the current situation. However, if the check shows they cannot implement the change as they desired, they think of another solution beforehand. This will save them both money and time, which they then can use to come up with a new and better solution.

Although this methodology looks promising, it is, not an easy task to check a BPM, because there is a mismatch between the structure of a business process model and specification languages in the form of temporal logic. The aforementioned methodology solves this problem by translating a BPM into a Kripke structure which can be tested with temporal logic. In theory, this logic can be checked on the Kripke structure directly, but the use of a model checker is preferred, because these are very efficient. However, the problem is that most of the model checkers do not accept Kripke structures as input directly, but they alter the input internally. The output of the methodology of Groefsema and Van Beest is an efficient representation of reality, when this model is altered inside the model checker, there is a high probability that the model checker will produce unnecessary overhead. Therefore it is needed to come up with a model checker and a translation from Kripke to the input language of the model checker, which produces a model as close as possible to the original Kripke structure, while it does not introduce additional overhead or state explosion and which supports a useful temporal logic (Groefsema and Van Beest, personal communication, 11-10-2013). When a suitable model checker has been found, the next step will be to make a mapping between the Kripke structure with the temporal logic formulas and the input the model checker requires.

This project is a search for the best model checker when checking business process compliance and using the methodology of Groefsema and Van Beest. With the help of this new model checker, the methodology can become better and one step closer to being used by companies. Therefore, one could say that all the companies that could benefit from this methodology, are also stakeholders in this project. However, although there are a lot of companies that could benefit from this methodology, they are not closely involved in this project and cannot influence the outcome. Therefore, for this project, only the supervisors Groefsema and Van Beest are considered as stakeholders. The implementation of the methodology is of course the bigger picture, but this project is just a small piece of the puzzle.

In order to come to the solution, and give a slight help in the realization of the methodology, this project aims to find the best model checker and mapping and therefore the research question and sub questions stated below will be addressed in this bachelor thesis.

Research question

Which model checker can be used best with a Kripke structure and temporal logic as input and how can a mapping be made of this model and the required input of the model checker with the least overhead?

Sub questions

- (For each model checker) Which input does the model checker require?
- (For each model checker) To which degree is the structure of the output of the methodology compatible with the input of the model checker?
    - Within this question there will also be looked into the possibility of a direct conversion between the Kripke structure and the internal transition system generated by the model checker. This would make it possible to insert the Kripke structure and the temporal logic formulas directly into the model checker without the need of going through the model checker’s input language.
- (Only for the selected model checker) Which objects of the output need to be mapped with which objects of the input?

Methodology

For the ‘Definitions’ section the information has come mainly from textbooks and lectures. The given example was created especially for this docment, and tries to clarify the subjects of temporal logic and Kripke structures.

The requirements are determined in collaboration with the supervisors and future users Groefsema and Van Beest. The use of a scoring system makes it possible to count all the points every model checker has earned and to make a comparison based on these scores. The scores are based on the requirements, whereby the more important requirements resulted in more points.

Supporting the right property language was one of the most important requirements. Therefore the property language was one of the requirements to form the short list (see below). Still, there are also differences between CTL, LTL, both CLT and LTL and CTL* (here in order of ascending preference). To give an impression of this preference, CTL* has been given quite a bit higher grade than CTL or LTL alone and a little bit higher than CTL and LTL together. The maximum score (10) was also higher than in some other sections (4 or 5), because of the importance of the requirement.

The other requirements are graded in a similar way. The first requirements in table 1 (page 10) received the highest maximum score, because they were the most important ones. The other requirements were especially nice to have, but not mandatory. This is reflected in the fact that the maximum score was only half of the maximum score of the other requirements, which cases their score to have a smaller effect on the total score. The overhead/speed section is a little bit different then the rest, because it has three different possible scores instead of a maximum score. This is done because real data was absent and a real order could not be made. Therefore, more global information concerning the overhead or speed of the model checkers is used on basis of which the model checkers could not be ranked exactly, but could be divided in three groups: among the worst, moderate and among the best.

The model checkers in the section ‘Searching for the right model checker’ are therefore discussed on the basis of the requirements. At the end of the paragraph on a specific model checker, an overview on the model checker is given. At the end of the chapter all these overviews are combined and are the basis for the choice for the right model checker.

At first, a shortlist has been made based on a few requirements. Only model checkers that are freely available for Windows and use at least CTL or LTL made it to the shortlist. Still not all of the model checkers that meet these criteria are discussed below. There are many model checkers that do not support CTL*, but only CTL or LTL. The model checkers that are most used, are typically also the ones with the most documentation. Therefore only the CTL and/or LTL model checkers with a fair amount of documentation are included in this paper.

For the model checkers that do support CTL*, the criteria where a little less strict because there are far less that do. Therefore all CTL* supporting model checkers are included in this paper. When a CTL* supporting model checker has been omitted from this list, or when a new model checker becomes available, they can still be ranked with the use of the ranking system given below. In this way, the new model checker can be compared with the already discussed model checkers and can be concluded if this new model checker would be better.

Definitions

What is a model checker?

A model checker is a computer program which checks exhaustively and automatically, whether a given model complies with the requirements that have been set for this model. A model checker uses a representation of reality, in the form of a model, which can be seen as a directed graph (*Model checkers*, 2013). There are many different model checkers, which (among other differences) use a wide range of property, programming and modeling languages.

Models used

What is a transition system?

A transition system TS is a tuple $(S, Act, \rightarrow, I, AP)$ where (Baier & Katoen, 2008):
• S is a set of states (need not be finite)
• *Act* is a set of actions
• $\rightarrow \subseteq S \times Act \times S$ is a transition relation which defines which state y can be reached from state x with the use of which action α, in the form ($x \overset{\alpha}{\rightarrow} y$)
• $I \subseteq S$ is a set of initial states, which are the states in which the transition can begin
• *AP* is a set of atomic propositions, the simple know facts about the states of the system

A transition system is basically a directed graph where nodes represent states, and edges model transitions, i.e., state changes. A state gives some information about the system at a certain moment in its behavior. For example, a state of a traffic light will indicate which color is currently shown. The transitions specify, as one can foresee, how the system can evolve from one state to another. When we look at the example of a traffic light again, a transition can indicate a switch from one color to the next (Baier & Katoen, 2008, p. 19-20).

The behavior of a transition system can be described as follows. The start is made in an initial state $s_0 \in I$ and then the system evolves according to the transition relation. So for example a state s can evolve in a state s' through an action α. Then s' can itself evolve further trough a next transition. This procedure will stop once a state is encountered that has no outgoing transitions. The system will also not start when the set I is empty, i.e. there is no initial state. Further, a transition system is nondeterministic, so in the case of multiple possible transitions or initial states, the outcome of the selection process is not known a priori. (Baier & Katoen, 2008, p. 20-21)

What is a Kripke structure?

First the definition of a Kripke structure is presented (Necula, 2004):
• AP: set of atomic propositions, the simple known facts about the states of the system
• K: Kripke structure over AP = (S, S_0, T, L)
• S: set of states (need not be finite)
• S_0: set of initial states, which are the states in which the transition can begin
• T: transition relation ($T \subseteq S \times S$) defines which states y can be reached from state x in the form (x, y)
• L: labeling function ($L : S \rightarrow 2^{AP}$) defines which atomic proposition y can state x have in the form $(x, \{y\})$ (in which y can also be replaced by more variables)

A Kripke structure (KS) a transition system with a few specific characteristics. A Kripke structure is a labeled transition system and is state-based (meaning that states are labeled in contrast to an event-based system in which the transitions are labeled) (Schoren, n.d.). Furthermore, where a transition system will stop once a state is encountered with no outgoing transitions, in a KS there is always the possibility to construct an infinite path. This is because the transition relation is left-total, which means that every input must have at least one output. Although a KS may continue infinitely in theory, in practice it will stop once it reaches a deadlock state. A deadlock state only has one single outgoing edge back to itself (Gupta et al., 2013).

Temporal logic

What is Linear Temporal Logic (LTL)?

'LTL is a temporal logic which uses connectives that allow us to refer to the future' (Huth & Ryan, 2004, p. 175). LTL is called linear, because the qualitative notion of time is path-based and viewed to be linear: at each moment of time there is only one possible successor state and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the interpretation of LTL formulae is defined in terms of paths, i.e., sequences of states. Paths themselves, though, are obtained from a transition system that might be branching: a state may have several, distinct direct successor states, and thus several computations may start in a state. (Baier & Katoen, 2008, p. 313)

LTL formulae over the set of atomic propositions are formed according to the following grammar (Baier & Katoen, 2008):

$\boldsymbol{\varphi_1} \wedge \boldsymbol{\varphi_2}$ = φ_1 has to hold and φ_2 has to hold

$\boldsymbol{\varphi_1} \vee \boldsymbol{\varphi_2}$= φ_1 has to hold, φ_2 has to hold, or both have to hold

$\neg\, \boldsymbol{\varphi}$ = φ has to be false (negation)

$\Diamond\ \boldsymbol{\varphi}$ = φ has to hold in the future/finally, can also be written as **F** φ (Future/Finally)

$\Box\ \boldsymbol{\varphi}$ = φ has to hold now and on the entire following path, can also be written as **G** (Globally)

$\circ\ \boldsymbol{\varphi}$ = φ has to hold on the next state, can also be written as **X** (neXt)

$\boldsymbol{\varphi_1 \mathrm{U}\ \varphi_2}$= φ_1 has to hold at least Until φ_2, which has to hold at the current or a future state

$\boldsymbol{\varphi_1 \mathrm{W}\ \varphi_2}$= φ_1 has to hold at least Until φ_2, but φ_2 may never occur. (Weak until)

$\boldsymbol{\varphi_1 \mathrm{R}\ \varphi_2}$= φ_2 has to hold until and including to the point where φ_1 holds, but φ_1 may never occur (in which case φ_2 will (have to) be true for the entire path) (Release)

$\boldsymbol{\varphi_1} \rightarrow \boldsymbol{\varphi_2}$= If φ_1 than φ_2, which is defined by: $\neg\,\varphi_1 \vee \varphi_2$(implication)

$\boldsymbol{\varphi_1} \leftrightarrow \boldsymbol{\varphi_2}$= If φ_1 than φ_2 and vice versa, which is defined by $(\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$ (equivalence)

$\boldsymbol{\varphi_1} \oplus \boldsymbol{\varphi_2}$= either φ_1 or φ_2 has to hold, but not both, which is defined by $(\varphi_1 \wedge \neg\,\varphi2) \vee (\varphi_2 \wedge \neg\,\varphi_1)$ (exclusive or)

$\boldsymbol{\mu} \vDash \boldsymbol{\varphi}$ = a *satisfaction relation*, indicating the evaluations μ for which a formula φ is true

These formulae can be satisfiable (if there exists a linear time structure M for which $M \vDash \varphi$ holds), valid (if for all linear time structures M, $M \vDash \varphi$ holds) or false (Necula, 2004).

What is Computation Tree Logic (CTL)?

'CTL is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be 'actual' path that is realized' (Huth & Ryan, 2004, p. 208). An important difference from LTL that comes forth from this is that the temporal operators in branching temporal logic allow the expression of properties of some or all computations that start in a state. To that end, it supports an existential path quantifier (denoted ∃ or E) and a universal path quantifier (denoted ∀ or A) (Baier & Katoen, 2008, p. 314). Furthermore does CTL distinguish between state formulae (Φ) and path formulae (φ). State formulae express a property of a state, while path formulae express a property of a path, i.e., an infinite sequence of states.

CTL grammar is closely related to the aforementioned LTL grammar, but no two temporal operators (◊/F, □/G, ○/X, U, W, R) may occur without being separated by path quantifiers (∀ and ∃) (Necula, 2004). Therefore XX φ is an LTL formula, but is not allowed in CTL.

What is CTL*?
Since there exist LTL formulae for which no equivalent CTL formula exists and CTL formulae for which no LTL equivalent exists, CTL* has been introduced. 'CTL* is a logic which combines the expressive powers of CTL and LTL' (Mark & Ryan, 2004, p. 218). With CTL* it is possible to formulate formulae that are not possible in either CTL or LTL. CTL* is an extension of CTL as it allows path quantifiers ∃ and ∀ to be arbitrarily nested with linear temporal operators such as X and U. In contrast, in CTL each linear temporal operator must be immediately preceded by a path quantifier. As in CTL, the syntax of CTL* distinguishes between state and path formulae. The syntax of CTL∗ state formulae is roughly as in CTL, while the CTL* path formulae are defined as LTL formulae, the only difference being that arbitrary CTL* state formulae can be used as atoms. For example, ∀XX φ is a legal CTL* formula, but does not belong to CTL. The same applies to the CTL∗ formulae ∃GF φ and ∀GF φ (Baier & Katoen, 2008, p. 422).

Example
When looking at figure 1, we see an example KS. To represent this structure in mathematical form, we refer to the second paragraph where is stated that a KS consist of the following elements: AP, S, S_0, T and L. When we state this for the example, the outcome is as follows:

AP	$= \{p, q, r, s\}$
S	$= \{1, 2, 3, 4\}$
S_0	$= \{1\}$
T	$= \{ (1,2), (1,3), (2,1), (2,3), (2,4), (3,4), (4,4) \}$
L	$= \{ (1,\{p\}), (2,\{p,q\}), (3,\{p,r\}), (4,\{s\}) \}$

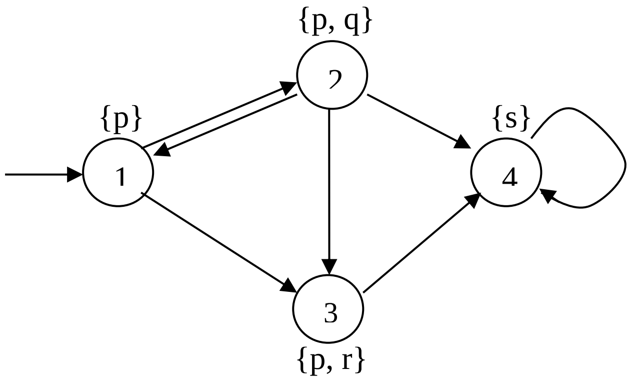

Figure 1. Example of a Kripke structure.

Now it is possible to test this KS with some logic formulae.
At state 1, the following formulae hold (among others):

$\forall \boldsymbol{X}\ \boldsymbol{p}$ – since in every state related to 1 (2 and 3) there is a p
$\exists \boldsymbol{FG}\ \boldsymbol{s}$ – since state 4 complies to Gs, and state 4 is state that can be reached from state 1
$\exists(\boldsymbol{p}\ \boldsymbol{U}\ \boldsymbol{Gs})$ – since there exist a path in which p is always true, until it becomes s for the rest of the time.
$\forall(\boldsymbol{p}\ \boldsymbol{W}\ \boldsymbol{Gs})$ – in this case it states that all paths comply to $p\ W\ Gs$, which is true because of the Weak until (p will always be true until s would become globally true, which may never occur) and would be false when Until was used, because there is a path (1,2,1,2,1,2, etc.) which never becomes Gs.
$\forall \boldsymbol{G}\ (\boldsymbol{r}\ \rightarrow\ \forall \boldsymbol{X}\ \boldsymbol{s})$ – at any state reachable from 1 (the $\forall G$ part), where r is true ($r\ \rightarrow$) the next state is always ($\forall X\ s$).
$\exists(\boldsymbol{q}\ \boldsymbol{R}\ \boldsymbol{p})$ – since there exist a path that where p holds, until and including to the point where q holds (1,2)
$\forall \boldsymbol{F}\ (\neg \boldsymbol{q} \wedge \neg \boldsymbol{r})$ – every path from 1 has a future point where neither q nor r is true (either state 1 or 4)

The following formulae however, are examples of formulae that do not hold at state 1:
$\forall \boldsymbol{Fs}$ – although there are paths that from state 1 that comply to Fs, not all of the paths from state 1 become s eventually (for example 1,2,1,2,1,2, etc.).
$\forall \boldsymbol{G}\ (\exists \boldsymbol{X}\ \boldsymbol{r}\ \rightarrow\ \forall \boldsymbol{Fs})$ – this means: at any state reachable from 1 (the $\forall G$ part), where there exist a path in which the next will be r ($\exists X\ r$) (true for state 1 and 2), there will always be an s in the future ($\forall Fs$). This formula does not hold, because as mentioned before, $\forall Fs$ does not hold at state 1.

Requirements

In table 1 (below), the requirements concerning the choice for the right model checker are listed. These are also the elements which will be investigated when answering sub question one (Which input does the model checker require?). Instead of making a sub question for each of these requirements, one sub question has been chosen in which all these elements will be discussed.

Furthermore, a model checker will be given points for each requirement. In the following chapter the model checkers will be compared, based on their score. The score is the total of all the points the model checker acquires. When a requirement is formulated in a negative way (e.g. the supported platform should not be only Linux), the requirement results in a negative score. The model checkers that do not comply with this requirement will therefore not come up at the top of the list with the scores.

Table 1. Requirements for the model checker.

Requirement		**Points**	**Comments**
Property language	CTL*	10	Only one of these possibilities has to be available for the selected model checker, but the top one is the most preferred.
	CTL and LTL	8	
	CTL	4	
	LTL	4	
Platform	Windows	10	
	(only) Linux	-100	
Software license	Free	10	
	Not free	-100	If the model checker is found to be highly suitable, the score may become 10.
Transition system	Labeled and state based	10	
	Supporting fairness	5	
Overhead/speed		0 – 2 – 4	These points reflect the position of the model checker when compared with the other possibilities. The scores reflect respectively: among the worst – average – among the best.
Ease of use	Programming language Java	5	The use of java as programming language can have some advantages, because Groefsema and Van Beest are using java in their current project.
	A direct conversion between the Kripke structure and the internal transition system generated by the model checker.	5	This would make it possible to insert the Kripke structure and the temporal logic formulas directly into the model checker without the need of going through the model checker's input language

Searching for the right model checker

While searching for model checking tools, a first division had been made based on the supported property language, the supported platform and the need for a license, because these are important requirements and relatively easy to check. In this chapter only the tools that made it past this first division are discussed. Otherwise the list will become too extensive, which will affect the comprehensibility.

NuSMV
An earlier research looked into the model checker NuSMV (Tello, 2013). This model checker is used in the project of Groefsema and Van Beest at the moment and will serve in the research as a standard, with which the rest of the model checkers are compared. NuSMV supports both CTL and LTL, is available for Windows and free to use (Frappier, Fraikin, Chossart, Chane-Yack-Fa, & Ouenzar, 2010). Therefore it receives 8 points for property language, and 10 for both platform and software license. Furthermore, it makes use of a state based labeled transition system and supports fairness (Cimatti, Clarke, Giunchiglia & Roveri, 2000). This results in 10 and 5 points for the transition system. A drawback of NuSMV is that the description level is quite low-level. Because of this it is necessary that all assignment, parameters or array indexes are constant and therefore each case may have to be written separately (Frappier et al. 2010). Therefore NuSMV is considered among the worst when considering Overhead/speed and does NuSMV receive 0 points. Because NuSMV does not fulfill any of the requirement in the ease of use section, it receives 0 points in this section.

Table 2. Score of the NuSMV model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
NuSMV	8	10	10	10	5	0	0	0	43

LTSmin
LTSMin is a model checker developed by the university of Twente, uses a labeled state based model, supports CTL* (Blom et al., 2010) and is free to use (10 points for property language, software license and transition system). LTSmin is also among the faster model checkers, because a good deal of work has been done to optimize the model checker and is awared 4 points for speed (*Model Checking and LTS Minimization*, n.d.). Therefore, at first, LTSmin seems a good candidate. However, LTSmin is written in C and therefore more difficult for newcomers, especially if they are used to a higher level programming language (Oostinga, 2012). Oostinga has tried to make a java bridge to overcome this problem, but state labels and deadlocks are not yet implemented, while these are necessary for the current project. Besides the problem concerning the programming language, LTSmin is not directly available on Windows. LTSmin can only be used on Windows through Cygwin, which is a collection of tools which provide a Linux look and feel environment for Windows (*Model Checking and LTS Minimization*, n.d.). Because of this LTSmin receives only 5 points for platform. Another drawback is that LTSmin does not support fairness, this is concluded because it is nowhere mentioned that LTSmin does support fairness. So it can be concluded that LTSmin is an option for this project because of the features is has concerning the checking of the models, but is not immediately preferred because of the difficulties it brings regarding the platform and the programming language.

Table 3. Score of the LTSmin model checker

Model checker	Property language	Platform	Software license	Transition system		Overhead/ Speed	Ease of use		Score
LTSmin	10	5	10	10	0	4	0	0	39

CADENCE SMV
CACENCE SMV is a model checking tool developed by the Caddence Berkely Labs and it uses a state based labeled transition system (as it is an extension of SMV (MacMillen, n.d.) and SMV uses a state based labeled transition system (MacMillen, 1998)). Furthermore it is available for Windows and free to use. Therefore it receives 10 points for platform, software license and transition system. Although it doesn't support CTL* it does support both CTL and LTL (8 points for property language). CADENCE SMV also supports fairness and receives another 5 points for transition system (MacMillen, 1998). At first CADENCE SMV seems to be a good candidate for the rest of this project (despite the lack of supporting CTL*), but as can be seen by looking at the names, both NuSMV and CADENCE SMV are an extension from SMV and therefore there is a strong resemblance between the two model checkers. Although CADENCE SMV is a little bit faster and uses less memory when compared with NuSMV (Owen, 2007), it might be too much work to replace the current model checker with a new one for these small advantages. Still, because CADENCE is somewhat faster than NuSMV it is grouped in the moderate class and does CADENCE receive 2 points for overhead/speed.

Table 4. Score of the CADENCE SMV model checker

Model checker	Property language	Platform	Software license	Transition system		Overhead/ Speed	Ease of use		Score
CADENCE	8	10	10	10	5	2	0	0	45

UPPAAL
UPPAAL is a toolbox jointly developed by Uppsala University and Aalborg University (*Uppaal Home*, n.d.). UPPAAL is built especially for verification of real-time systems and therefore it uses a subset of TCTL (Timed Computational Tree Logic) (Behrmann, David & Larsen, 2006). Furthermore, UPPAAL uses state based labeled transition systems, is freely available for Windows and has a nice GUI (10 points for platform, software license and transition system). Despite these good features, UPPAAL has also a few big drawbacks. Since the focus of UPPAAL is to test real-time systems, the language is not expressive enough for other (non-real-time) systems. Therefore some parts of functional and imperative languages cannot be expressed correctly in UPPAAL, like the 'Until' and 'Release' operators (Wang, 2002). Because of this UPPAAL only receives 2 points for the property language. Another problem with UPPAAL is that is does not support fairness (Bortnik, 2005). Other advantages are that UPPAAL uses java, which results in 5 points for ease of use, and is probably among the faster model checkers because the authors have done quite some work on optimization, which earns UPPAAL 4 points for overhead/speed (Behrmann, David & Larsen, 2006).

Table 5. Score of the UPPAAL model checker

Model checker	Property language	Platform	Software license	Transition system		Overhead/ Speed	Ease of use		Score
UPPAAL	2	10	10	10	0	4	5	0	41

Xspin
Xspin is the graphical interface of Spin and makes the difficult to use model checker Spin, more convenient to use. Xspin does run Spin in the background and has therefore the same characteristics as Spin (Wang, 2002). Spin is free to use and is available for Windows and supports fairness (Holzmann, 1997), which results in 10 points for both platform and license and 5 points for transition system. Although Spin does use labeled transitions systems internally, the input the user has to provide looks more like programming code than the build-up of a transition system as given in the example in the 'Definitions' chapter (Wang, 2002). This is because the internal transition system is computed on-the-fly, while checking a property. Because of this different kind of transition system, Xspin receives only 2 points for the transition system. Working on-the-fly avoids the need of a global state transition graph, but it also implies that transitions are (re-)computed for each property that has to be verified (Frappier et al., 2010). This can cause Spin to generate an unnecessary amount of overhead. Therefore Spin receives 0 points for overhead/speed. Last but not least, the only logic Spin uses is LTL. As mentioned above LTL does not use existential and universal path quantifiers, which makes reachability properties difficult and sometimes even impossible (Frappier et al., 2010). Because of the lack of existential and universal path quantifiers, an LTL formula holds if and only if it holds at every possible path. Some researchers say that it is possible, after some changes, to verify CTL* formulae with SPIN (Visser & Barringer, 1999). However, this is discusses in an article from 1999 and is still not used in current versions of SPIN. Because of this combined with the difficulties of including CTL* verification brings with it; in this paper SPIN is discussed as only supporting LTL and therefore receives only 4 points for property language. Since Spin only supports LTL and works on-the-fly (with no pre-made global state transition graph), Spin is probably not the best choice for this project.

Table 6. Score of the Xspin model checker

Model checker	Property language	Platform	Software license	Transition system		Overhead/ Speed	Ease of use		Score
Xspin	4	10	10	2	5	0	0	0	31

CADP - XTL
CADP is a toolbox for the design of asynchronous concurrent systems, but also offers the model checker XTL (*CADP Homdepage*, n.d.). CADP has a free academic license, is available for Windows and the latest version also supports fairness (Garavel, Lang, Mateescu & Serwe 2011). This results in 10 points for both platform and software license and 5 points for transition system (second part). XTL makes use of a labeled transition system, in the CADP toolbox encoded in the BCG (Binary Coded Graph) format (Mateescu, & Garavel, 1998). Although it is built up a little different than a Kripke structure, it shows quite some resemblance. The biggest drawback is that the internal transition system is event based, in contrast to a state based Kripke structure. Because of this CADP only receives 5 points for transition system (first part). Another disadvantage of this model checker is that is does not support CTL* or both CTL and LTL, but only CTL (among other temporal logics that are not used in this research) and thus receives only 4 points for property language (Frappier et al., 2010). In the overhead/speed section does CADP receive 4 points, because quite some work has been done on optimizing CADP and therefore it is considered among the fastest model checkers (Garavel et al., 2011). Because CADP – XTL only uses CTL and is event based, it will probably not be the best choice.

Table 7. Score of the CADP - XTL model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
CADP	4	10	10	5	5	4	0	0	38

ProB
ProB is a model checking tool designed for the B Method (Leuschel & Butler 2003). It is free of license available for Windows and supports both CTL and LTL, which results in 10 points for both platform and license and 8 for property language. Despite these nice features, there are big drawbacks. The biggest problem with ProB is that it does not work with transition systems, states and events, but with actions specified as an operation defined as a precondition and a postcondition (Frappier et al., 2010). Because there is also no mention of supporting fairness, ProB receives 0 points for the transition system. Besides this problem is ProB not a very fast model checker and will therefore have relatively much overhead when compared with the other model checkers (Leuschel & Butler 2003). This results in 0 points in the overhead/speed section.

Table 8. Score of the ProB model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
ProB	8	10	10	0	0	0	0	0	28

ARC
The ARC model checker supports the property language CTL* and therefore it receives 10 points for property language (*ARC*, n.d.). Furthermore, the model checker is freely available and therefore receives another 10 points. From there, it goes downhill when talking about the usefulness of ARC for this project. First, ARC is available for Windows, but only with the use of Cygwin (*ARC - AltaRica Checker*, n.d.), so only 5 points are rewarded for the platform. The biggest problem with ARC however is that is that ARC does not make use of transition systems. ARC makes use of UML collaboration (also called communication diagrams, and therefore both states and events are expressed in a very different then in a transition system (Henderson, 2003). There is also no mention of fairness in the ARC literature, and therefore it is likely that ARC does not support fairness. Because of both last observations, ARC receives 0 points for transition system. These drawbacks are so big that this also means that ARC is no longer a serious candidate and therefore the research on both overhead and ease of use are skipped.

Table 9. Score of the ARC model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
ARC	10	5	10	0	0	N.A.	N.A.	N.A	25

MLSolver
MLSolver is among the possibilities, because it supports CTL* (Friedmann & Lange, 2010). Because of this, MLSolver scores 10 point for property language. Another 10 points are rewarded for the software license, because the model checker is freely available. A

disadvantage is that MLSolver needs OCaml (*Readme*, n.d.), which can only run on Windows with the use of Cygwin (*Install OCaml*, n.d.). Therefore, only 5 points are rewarded for the platform. A positive aspect is that MLSolver uses labeled transitions systems, but is unfortunately event based (instead of state based) (Friedmann & Lange, 2010). Therefore, MLSolver receives only 5 out of 10 points for the transition system plus another 5, because the model checker supports fairness (Friedmann & Lange, 2010). Although a comparison between MLSolver and the one of the other model checkers in this research has not been found, it is plausibly that MLSolver would end up in the top of the list when speaking about speed, because there has been put quite some effort in making the model checker faster (Friedmann & Lange, 2010). Therefore, 4 points are rewarded for speed.

Table 10. Score of the MLSolver model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
MLSolver	10	5	10	5	5	4	0	0	39

VeriCS
VeriCS is a model checker that, like UPPAAL, is specialized in real-time systems (Kacprzak et al., 2008). VeriCS might be a bit faster than UPPAAL (a comparison has not been made), but the VeriCS GUI is less intuitive to use than the UPPAAL GUI. However these are only small differences. The main advantage of VeriCS over UPPAAL is that VeriCS supports CTL* (*Temporal logics*, n.d.). At least, the latest available version does. The problem is that this latest version is only available for Linux. The latest version for Windows is VeriCS 2008, but in a paper from 2010 only variations of timed CTL are mentioned and not (timed) CTL* (Knapik et al., 2010). Therefore, the version of VeriCS that is available for this project shows too many similarities with UPPAAL and will not be discussed in more detail or in the conclusion of this chapter.

MCheck
In the literature suggested by the producer of MCheck is written: 'The program supports three flavors of temporal logic formulas: LTL, Linear Time Temporal Logic; CTL, Computation Tree Logic; and CTL* …' (Sember, 2005). However, in the next sentence is stated that only LTL and CTL formulae can be verified with the use of MCheck. Because it is necessary for this project that the formulae can be verified, MCheck will only receive 8 points for the property language. Furthermore, MCheck is freely available for Windows and therefore receives 10 points for both platform and license. A minor drawback is that MCheck does not support fairness (Sember, 2005). On the other hand a big advantage is that the model checker makes use of Kripke structures and therefore has the same structure as the models of Groefsema and Van Beest. Therefore MCheck scores 10 points for its transition system. Furthermore, the model checker is written in Java (4 points) and this combined with the use of Kripke structures results in another 5 points in the 'ease of use' section because the model of Groefsema and Van Beest can be fed to the model checker without much change. Unfortunately there is no comparison between the speed of MCheck compared to another model checker and the literature on the speed of MCheck also is inconclusive. Because there is also no mention of issues with the speed of the model checker, MCheck is considered average and receives 2 points for speed. Finally, in the future work section both CTL* and fairness are listed as improvements (Sember, 2005). Although this is interesting, a newer version of MCheck which already encompassed these features was not found.

Table 11. Score of the MCheck model checker

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
MCheck	8	10	10	10	0	2	5	5	50

Which model checker suits this project best?
All the information given above is summarized in the following table:

Table 12. Model checkers compared.

Model checker	**Property language**	**Platform**	**Software license**	**Transition system**		**Overhead/ Speed**	**Ease of use**		**Score**
NuSMV	8	10	10	10	5	0	0	0	43
LTSmin	10	5	10	10	0	4	0	0	39
CADENCE	8	10	10	10	5	2	0	0	45
UPPAAL	2	10	10	10	0	4	5	0	41
Xspin	4	10	10	2	5	0	0	0	31
CADP	4	10	10	5	5	4	0	0	38
ProB	8	10	10	0	0	0	0	0	28
ARC	10	5	10	0	0	N.A.	N.A.	N.A.	25
MLSolver	10	5	10	5	5	4	0	0	39
MCheck	8	10	10	10	0	2	5	5	50

MCheck is the model checker with the highest score and this is mainly the result of the fact that MCheck uses both Java and Kripke structures, which makes it easier to use in this particular project. Furthermore, it might be possible to communicate with the producer of MCheck to see if the remarks on future work can be realized. If both satisfaction testing for CTL* formulas and fairness constraints could be added, MCheck would be the ideal model checker. At the moment it is ‘only’ the best of the discussed model checkers.

Mapping

Now that a certain model checker is chosen, one last step is necessary before the model checker can be actually used in the methodology. A translation has to be made from the Kripke structures produced by the methodology to the input that MCheck requires. One of the main reasons to choose MCheck was that the model MCheck used internally was also a Kripke structure and looked a lot like the models produced by Groefsema and Van Beest. Because of this, the translation only needed to change a few details and change the sequence of the variables. In figure 1 (next page) the piece of pseudo code is given, that will perform the translation.

In MCheck statements about the model need to be surrounded by accolades followed by the temporal logic formulae. Therefore, the translation code starts with writing an opening accolade to the string (line 4).

Then it is important to show the build-up of the MCheck input, because this is the build-up the string has to have. Table 13 shows this build-up. The first column shows a >-mark when the particular state is an initial state. In the table only one possible next state and only one atomic proposition are included, but these can be replaced by any quantity (with a minimum of 1). And after each state an enter is included.

Table 13. The build-up of the MCheck input

If initial state then: “>”	State	Possible next state	Atomic proposition
Lines 8-10 in code	Line 11	Line 12	Line 13

Now the statements about the model are completed, so a closing accolade is written to the string (line 17). Then a few last lines are added to include the temporal logic formulae in the string (lines 19-22). This is done by writing the formula to the string and writing each formula on a separate line of text.

```
public class MCheckConverter {
public static String converter() {

String inputMCheck = ("{");

     for each state {

          if (State is in Kripke.initial){
               inputMCheck.addToString("> ");
               }
          inputMCheck.addToString(State.id + " ");
          inputMCheck.addToString(State.getNextStates() + " ");
          inputMCheck.addToString(State.getAtomicPropositions() + " ");
          inputMCheck.addToString("\r\n");
      }

      inputMCheck.addToString("}");

     for each formula {
          inputMCheck.addToString(formula);
          inputMCheck.addToString("\r\n");
     }

     return inputMCheck;
```

Figure 1. MCheck converter code

Evaluation

The main goal of this project was to come up with a new model checker that is better suited for the methodology then the one currently used and provide a mapping to make it possible to use that model checker in the methodology. Therefore, at first a framework has been made within which model checkers can be compared with each other. In this research only a selection of model checkers is compared with each other, based on some selection criteria. The model checkers that had the highest change score and therefore the highest change of being able to add value to the methodology was chosen. By formulating the criteria, a generic way of rewarding model checkers has been established, based on measurable criteria. Consequently any model checker that is not incorporated in this paper can still be tested, using these criteria. Therefore, the results and the applicability of the selection procedure is not limited to the results here, but can also be applied beyond the scope if this paper.

Conclusion

This project is a small step in a larger project which tries to help companies to anticipate better on the changing market. Groefsema and Van Beest have formulated a methodology which makes it possible to check compliance of changed business processes prior to the execution of the change. In this methodology they made use of a model checker, but they were not sure it was the most suitable one for their project.

Therefore, the goal for this project was to find a model checker that would fit better in the methodology of Groefsema and Van Beest than NuSMV, the one currently used. In this paper, a framework has been presented within which model checkers can be compared with each other. In this way, it is possible to come to a conclusion which model checker is best suited for this project. MCheck is recommended to be used in the methodology from now on, because it was the model checker that had the highest score.

The model checker that is recommended in this paper suits the tool chain particularly well because it shows such great resemblance with the models that are produced with the methodology. However, it also has a number of shortcomings. A disadvantage is that it supports neither CTL* nor fairness, but this is not too much of a problem. Groefsema and Van Beest told that these problems can be circumvented. Another disadvantage is that MCheck is relatively old and not actively maintained or supported. The manual has been published in 2005, and therefore the model checker is probably finished in the same year. Thereafter, there is no documentation of upkeep, improvements or other changes. Since these shortcomings do not outweigh the advantages of the model checker, it is still an improvement compared to the current situation. This said, it is possible that in time a better alternative will become available that can replace MCheck.

To use MCheck in the methodology, a translation is needed. A mapping is given in this paper, through which the models produced by Groefsema and Van Beest can be turned into the format MCheck accepts. In this way the model checker can actually be used to improve the methodology.

With the help of this project, the methodology has come one step further in its development and therefore one step closer to be actually used in business. Then, companies will be able to anticipate better on the changing market, which results in a higher degree of customer satisfaction and can ensure compliance to any future regulation. That is the ultimate goal, but this project has helped a little to bring that closer to realization.

Literature

ARC (n.d.), http://altarica.labri.fr/wp/?page_id=32

ARC - AltaRica Checker (n.d.), https://altarica.labri.fr/forge/projects/4/wiki

Baier, C. & Katoen, J-P. (2008), *Principles of Model Checking*, Cambridge [Massachusetts]: The MIT Press.

Behrmann, G., David, A., & Larsen, K. G. (2006). *A Tutorial on Uppaal 4.0*. Retrieved 20-12-2013 from http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf

Blom, Stefan, Jaco van de Pol, and Michael Weber (2010). *LTSMIN: Distributed and Symbolic Reachability*, consulted on 25-11-2013 via http://eprints.eemcs.utwente.nl/18152/01/cav2010_BPW2.pdf

Boehm, B. W., Mcclean, R. K., & Urfrig, D. E. (1975). Some experience with automated aids to the design of large-scale reliable software. *Software Engineering, IEEE Transactions on*, (1), Pages 125-133.

Bortnik, E., N. Trčka, A.J. Wijs, B. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, J.E. Rooda (November–December 2005). Analyzing a χ model of a turntable system using Spin, CADP and Uppaal, *The Journal of Logic and Algebraic Programming*, Volume 65, Issue 2, Pages 51-104.

CADP Homepage (n.d.). http://www.inrialpes.fr/vasy/cadp/

Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (2000). NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4), Pages 410-425.

Ford, Henry, (1922). *My Life and Work*, available through http://www.gutenberg.org/ebooks/7213

Frappier, M., Fraikin, B., Chossart, R., Chane-Yack-Fa, R., & Ouenzar, M. (2010). Comparison of model checking tools for information systems, in *Formal Methods and Software Engineering*, Pages 581-596, Springer Berlin Heidelberg.

Friedmann, O., & Lange, M. (2010). A solver for modal fixpoint logics. Electronic Notes in *Theoretical Computer Science*, 262, Pages 99-111.

Garavel, H., Lang, F., Mateescu, R., & Serwe, W. (2011). CADP 2010: a toolbox for the construction and analysis of distributed processes. In T*ools and Algorithms for the Construction and Analysis of System*, Pages 372-387, Springer Berlin Heidelberg.

Gupta, Ashutosh et al. (last modified on 23 October 2013), *Kripke structure (model checking)* consulted on 11-11-2013 via http://en.wikipedia.org/wiki/Kripke_structure_(model_checking)

Henderson, P. (2003). Modelling architectures for dynamic systems, in *Programming methodology*, Pages 161-174. Springer New York.

Holzmann, G. J. (May 1997). The Model Checker SPIN, IEEE Transactions on Software Engineering, Vol. 23, No. 5, Pages 279-295.

Huth, Micheal & Ryan, Mark, (2004). *Logic in Computer Sciende: Modelling and Reasoning about Systems (2nd Edition)*, Cambridge: University Press.

Install OCaml (n.d.), http://ocaml.org/docs/install.html

Kacprzak, M., Nabiałek, W., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M., Wozna, B. & Zbrzezny, A. (2008). Verics 2007-a model checker for knowledge and real-time. *Fundamenta Informaticae,* 85(1), Pages 313-328.

Knapik, M., Niewiadomski, A., Penczek, W., Półrola, A., Szreter, M., & Zbrzezny, A. (2010). Parametric model checking with VerICS, in: *Transactions on Petri nets and other models of concurrency* IV, Pages 98-120. Springer Berlin Heidelberg.

Leuschel, Michael, & Butler, Michael (2003). ProB: A model checker for B. *FME 2003: Formal Methods*. Springer Berlin Heidelberg, Pages 855-874.

Mateescu, Radu, & Garavel, Hubert (1998). XTL: A meta-language and tool for temporal logic model-checking. *STTT*, 98, Pages 33-42.

McMillan, K. L., (1998). *Getting started with SMV,* consulted on 25-11-2013 via http://www.mimuw.edu.pl/~sl/teaching/04_05/PMW/SMV-doc/tutorial/.

McMillan, K.L., (n.d.). *The Cadence SMV Model Checker,* consulted on 25-11-2013 via http://www.kenmcmil.com/smv.html.

Model Checking and LTS Minimization (n.d.) consulted 22-11-2013 via http://fmt.cs.utwente.nl/tools/ltsmin/

Necula, G. C. (2004), *Temporal Logics:CS 294-3: Techniques for Automated Deduction,* consulted on 12-11-2013 via http://www.cs.berkeley.edu/~necula/autded/lecture23-temporal.pdf.

Oostinga, Ruben (2012). *A java bridge for LTSmin,* consulted on 26-11-2013 via http://essay.utwente.nl/61714/

Owen, David R. (2007). *Combining Complementary Formal Verification Strategies to Improve Performance and Accuracy*, Morgantown, West Verginia.

Readme (n.d.), https://github.com/tcsprojects/mlsolver

Schoren, R. (no date), *Correspondence between Kripke Structures and Labeled Transition Systems for Model Minimization*, consulted on 14-11-2013 via http://www.win.tue.nl/~timw/downloads/schoren_seminar.pdf

Sember, J. (2005). *MCheck: A Model Checker for LTL and CTL Formulas*, via http://www.cs.ubc.ca/~jpsember/mcheck/mcheck.pdf

Tello, Andrés (2013). *Business Process Model Checking: using PDVI techniques & the NuSMV model cheker tool,* internship report at Rijksuniversiteit Groningen.

Temporal logics (n.d.), http://verics.ipipan.waw.pl/node/8#top

Uppaal Home (n.d.), consulted on 20-12-2013 via http://www.uppaal.org/

Visser, W., & Barringer, H. (1999). CTL* model checking for SPIN. *Software Tools for Technology Transfer, LNCS*.

Wang, B. (2002). *A GUI for Model Checkers*; Msc Thesis Delft University of Technology, Department of EEMCS.

Wikipedia (last modified on 24 May 2013), *Model checking*, consulted on 22-11-2013 via http://en.wikipedia.org/wiki/Model_checking

Wikipedia (last modified on 17 November 2013). *List of model checking tools*, consulted on 22-11-2013 via http://en.wikipedia.org/wiki/List_of_model_checking_tools