Specification Guidelines to Avoid the State Space Explosion Problem

Jan Friso Groote, Tim W.D.M. Kouters, and Ammar Osaiweran

Eindhoven University of Technology
Department of Computer Science
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{J.F.Groote,A.A.H.Osaiweran}@tue.nl,
T.W.D.M.Kouters@student.tue.nl

Abstract. During the last two decades we modelled the behaviour of a large number of systems. We noted that different styles of modelling had quite an effect on the size of the state spaces of the modelled system. The differences were so substantial that some specification styles led to far too many states to verify the correctness of the model, whereas with other styles the number of states was so small that verification was a straightforward activity. In this paper we summarise our experience by providing seven specification guidelines, of which five are worked out in more detail.

Keywords: Design for verifications, specification guidelines, state space explosion, model checking.

1 Introduction

These days, we and others have ample experience in system design through discrete behavioural specification of computer systems. The primary lesson is that, although, behavioural specification is extremely helpful, it is not enough. We need to verify that the designed behaviour is correct, in the sense that it either satisfies certain behavioural requirements or that it matches a compact external description. It turns out that discrete behaviour is so complex, that a flawless design without verification is virtually impossible.

When verifying system behaviour, the state space explosion problem kicks in. This means that the behaviour of any real system quickly has so many states that despite the use of clever verification algorithms and powerful computers, verification is often problematic. Three decades of improvements of verification technology did not provide the means to overcome the state space explosion problem.

We believe that the state space explosion problem must, therefore, also be dealt with in another way, namely by designing models such that their behaviour can be verified. We call this *design for verifiability* or *modelling for verifiability*. Compared to the development of state space reduction techniques, design for verifiability is a barely addressed issue. The best we could find is [11], but it primarily addresses improvements in verification technology, too. Specification styles from the perspective of expressiveness have been addressed [14], but verifiability is also not really an issue here.

In this article we provide five specification guidelines that we learned by specifying complex realistic systems (e.g. traffic control systems, medical equipment, domestic appliances, communication protocols). For each guideline we give two examples. The first one does not take the guideline into account and the second does. We show by a transition system or a table that the state space that is using the guideline is much smaller. The 'bad' and the 'good' specifications are in general not behaviourally equivalent (for instance in the sense of branching bisimulation) but as we will see, they both capture the application's intent. All specifications are written in mCRL2, which is a process specification formalism based on process algebra [8,15]. A detailed version of this paper that contains a concise introduction of mCRL2 and two more guidelines can be found in [6].

In hindsight, we can say that it is quite self evident why the guidelines have a beneficial effect on the size of the state spaces. Some of the guidelines are already quite commonly used, such as reordering information in buffers, if the ordering is not important. The use of synchronous communication, although less commonly used, also falls in this category. Other guidelines such as information polling are not really surprising, but specifiers appear to have a natural tendency to use information pushing instead. The use of external specifications may be foreign to most specifiers.

Although we provide a number of guidelines that we believe are really important for the behavioural modellist, we do not claim completeness. Without doubt we have overlooked a number of specification strategies that are helpful in keeping state spaces small. Furthermore, a systematic or formal approach to relate the specification pairs for each guideline are beyond the interest of this paper. Hopefully this document will be an inspiration to investigate state space reduction from this perspective, which ultimately can be accumulated in effective teaching material, helping both students and working practitioners to avoid the pitfalls of state space explosion.

2 Overview of Design Guidelines

In this section we give a short description of the five guidelines that we present in this paper.

I **Information Polling**. This guideline advises to let processes ask for information, whenever it is required. The alternative is to share information with other components, whenever the information becomes available. Although, this latter strategy clearly increases the number of states of a system, it appears to prevail over information polling in most specifications that we have seen.

II **Global Synchronous Communication**. If more parties communicate with each other, it can be that a component 1 communicates with a component 2, and subsequently, component 2 informs a component 3. This requires two consecutive communications and therefore two state transitions. By using multi-actions it is possible to let component 1 communicate with component 2 that synchronously communicates with a component 3. This only requires one transition. By synchronising communication over different components, the number of states of the overall system can be substantially reduced.

III **Avoid Parallelism Among Components**. If components operate in parallel, the state space grows exponentially in the number of components. By sequentialising the behaviour of these components, the size of the total state space is only the sum of the sizes of the state spaces of the individual components. In this latter case state spaces are small and easy to analyse, whereas in the former case analysis might be quite hard. Sequentialising the behaviour can for instance be done by introducing an arbiter, or by letting a process higher up in the process hierarchy to allow only one sub-process to operate at any time.

IV **Restrict the use of Data**. The use of data in a specification is a main cause for state-space explosion. Therefore, it is advisable to avoid using data whenever possible. If data is essential, try to categorise it, and only store the categories. For example, instead of storing a height in millimetres, store *too_low*, *right_height* and *too_high*. Finally, take care that data is only stored in one way. E.g., storing the names of the files that are open in an unordered buffer is a waste. The buffer can be ordered without losing information.

V **Specify the External Behaviour of Sets of Sub-components**. If the behaviour of sets of components are composed, the external behaviour tends to be overly complex. In particular the state space is often larger than needed. A technique to keep this behaviour small is to separately specify the expected external behaviour first. Subsequently, the behaviours of the components are designed such that they meet this external behaviour.

3 Guideline I: Information Polling

One of the primary sources of many states is the occurrence of data in a system. A good strategy is to only read data when it is needed and to decide upon this data, after which the data is directly forgotten. In this strategy data is polled when required, instead of pushed to those that might potentially need it. An obvious disadvantage of polling is that much more communication is needed. This might be problematic for a real system, but for verification purposes it is attractive, as the number of states in a system becomes smaller when using polling.

Currently, it appears that most behavioural specifications use information pushing, rather than information polling. E.g., whenever some event happens, this information is immediately shared with neighbouring processes.

In order to illustrate the advantage of information polling, we provide two specifications. The first one is 'bad' in the sense that there are more states than in the second specification. We are now interested in a system that can be triggered by two sensors $trig_1$ and $trig_2$. After both sensors fire a trigger, a traffic light must switch from red to green, from green to yellow, and subsequently back

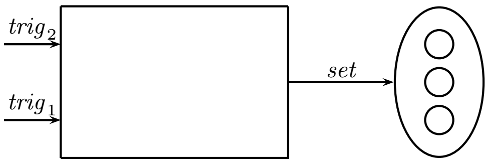

Fig. 1. A simple traffic light with two sensors

to red again. For setting the colour of the traffic light, the action *set* is used. One can imagine that the sensors are proximity sensors that measure whether cars are waiting for the traffic light. Note that it can be that a car activates the sensors, while the traffic light shows another colour than red. In figure 1 this system is drawn.

First, we define a data type *Colour* which contains the three aspects of a traffic light.

sort $\mathit{Colour} = \textbf{struct}\ \mathit{green} \mid \mathit{yellow} \mid \mathit{red};$

The pushing controller is very straightforward. The occurrence of $trig_1$ and $trig_2$ indicate that the respective sensors have been triggered. In the pushing strategy, the controller must be able to always deal with incoming signals, and store their occurrence for later use. In the specification below, the pushing process has two booleans b_1 and b_2 for this purpose. Initially, these booleans are false, and the traffic light is assumed to be red. The booleans become *true* if a trigger is received, and are set to *false*, when the traffic light starts with a *green*, *yellow* and *red* cycle. Note that we underline all external actions in the specification (but not in the text or in the diagrams) and we use the same style throughout the paper. External actions are those actions communicating with entities outside the described system, whereas internal actions happen internally in components of the system or are communications among those components.

$$
\begin{array}{lll}
\textbf{proc} & \multicolumn{2}{l}{\mathit{Push}(b_1, b_2{:}\mathbb{B}, c{:}\mathit{Colour})} \\
& = & \underline{trig}_1\cdot \mathit{Push}(\mathit{true}, b_2, c) \\
& + & \underline{trig}_2\cdot \mathit{Push}(b_1, \mathit{true}, c) \\
& + & (b_1 \wedge b_2 \wedge c \approx \mathit{red}) \rightarrow \underline{set}(\mathit{green})\cdot \mathit{Push}(\mathit{false}, \mathit{false}, \mathit{green}) \\
& + & (c \approx \mathit{green}) \rightarrow \underline{set}(\mathit{yellow})\cdot \mathit{Push}(b_1, b_2, \mathit{yellow}) \\
& + & (c \approx \mathit{yellow}) \rightarrow \underline{set}(\mathit{red})\cdot \mathit{Push}(b_1, b_2, \mathit{red}); \\
\textbf{init} & \multicolumn{2}{l}{\mathit{Push}(\mathit{false}, \mathit{false}, \mathit{red});}
\end{array}
$$

The polling controller differs from the pushing controller in the sense that the actions $trig_1$ and $trig_2$ now have a parameter. It checks whether the sensors have been triggered using the actions $trig_1(b)$ and $trig_2(b)$. The boolean b indicates whether the sensor has been triggered (*true*: triggered, *false*: not triggered). In *Poll*, sensor $trig_1$ is repeatedly polled, and when it indicates by a *true* that it has been triggered, the process goes to $Poll_1$. In $Poll_1$ sensor $trig_2$ is polled, and when both sensors have been triggered $Poll_2$ is invoked. In $Poll_2$ the traffic light goes through a colour cycle and back to $Poll$.

$$\begin{array}{ll}
\textbf{proc} & Poll = \underline{trig}_1(false)\cdot Poll + \underline{trig}_1(true)\cdot Poll_1;\\
 & Poll_1 = \underline{trig}_2(false)\cdot Poll_1 + \underline{trig}_2(true)\cdot Poll_2;\\
 & Poll_2 = \underline{set}(green)\cdot\underline{set}(yellow)\cdot\underline{set}(red)\cdot Poll;\\
\textbf{init} & Poll;
\end{array}$$

The transition systems of both systems are drawn in figure 2. At the left the diagram for the pushing system is drawn, and at the right the behaviour of the polling traffic light controller is depicted. The diagram at the left has 12 states while the diagram at the right has 5, showing that even for this very simple system polling leads to a smaller state space.

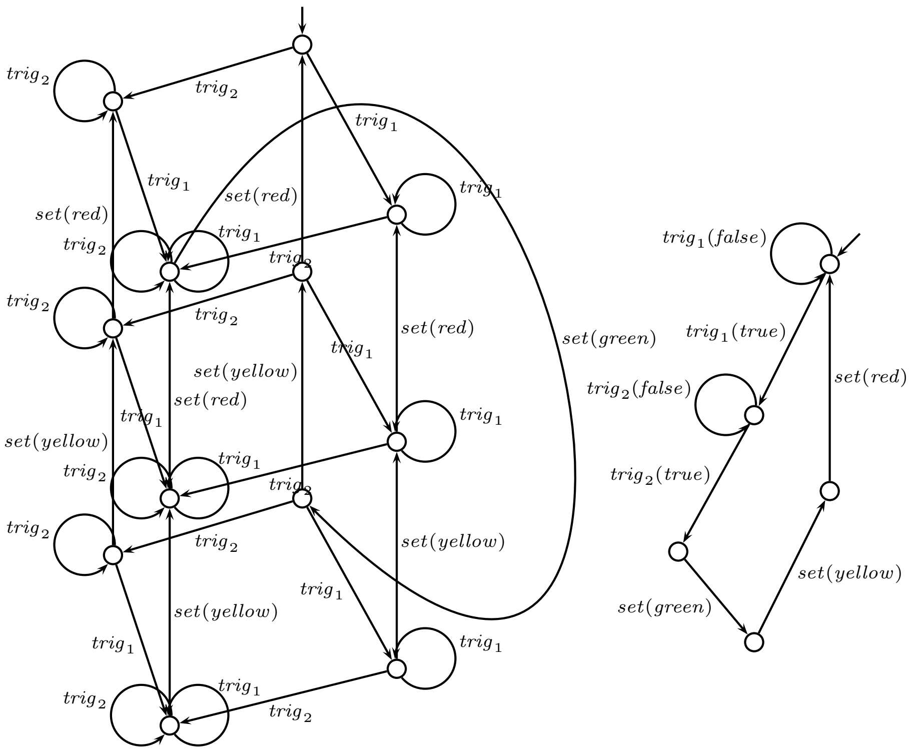

Fig. 2. Transition systems of push/poll processes

4 Guideline II: Use Global Synchronous Communication

Communication along different components can sometimes be modelled by synchronising the communication over all these components. For instance, instead of modelling that a message is forwarded in a stepwise manner through a number of components, all components engage in one big action that says that the message travels through all components at once. In the first case there is a new state for every time the message is forwarded. In the second case the total communication only requires one extra state. The use of global synchronous communication can be justified if passing this message is much faster than the other activities of the components, or if passing such a message is insignificant relative to the other activities.

Several formalisms use global synchronous interactions as a way to keep the state space of a system small. The co-ordination language REO uses the concept very explicitly [2]. A derived form can be found in Uppaal, which uses committed locations [10].

To illustrate the effectiveness of global synchronous communication, we provide the system in figure 3. A trigger signal enters at a, and is non-deterministically forwarded via b_c or c_c to one of the two components at the right. One might for instance think that there is a complex algorithm that determines whether the information is forwarded via b_c or c_c, but we do not want to model the details of this algorithm. After being passed via b_c or c_c, the message is forwarded to the outside world via d or e. To illustrate the effect on state spaces, it is not necessary that we pass an actual message, and therefore it is left out.

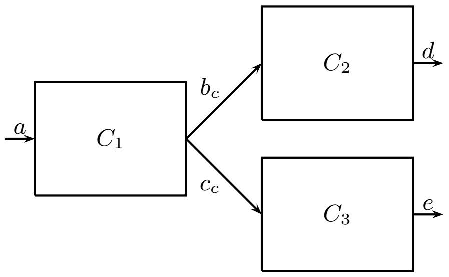

Fig. 3. Synchronous/asynchronous message passing

The asynchronous variant is described below. Process C_1 performs a, and subsequently performs b_s or c_s, i.e. sending via b or c. The process C_2 reads via b by b_r, and then performs a d. The behaviour of C_3 is similar. The whole system consists of the processes C_1, C_2 and C_3 where b_r and b_s synchronise via the Γ_s operator to become b_c, and c_r and c_s become c_c. The ∇_v operator allows multi-actions in v to happen, and blocks all others. The behaviour of this system contains 8 states and is depicted in figure 4 at the left.

proc $C_1 = \underline{a}\cdot(b_s + c_s)\cdot C_1;$
$C_2 = b_r\cdot\underline{d}\cdot C_2;$
$C_3 = c_r\cdot\underline{e}\cdot C_3;$

init $\nabla_{\{\underline{a},b_c,c_c,\underline{d},\underline{e}\}}(\Gamma_{\{b_r|b_s\to b_c,c_r|c_s\to c_c\}}(C_1||C_2||C_3));$

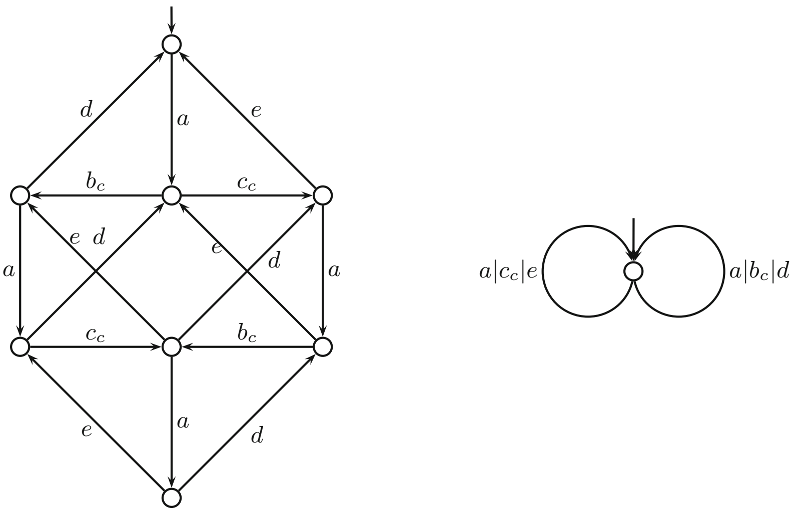

Fig. 4. Transition systems of a synchronous and an asynchronous process

The synchronous behaviour of this system can be characterised by the following mCRL2 specification. Process C_1 can perform a multi-action $a|b_s$ (i.e. action a and b_s happen exactly at the same time) or a multi-action $a|c_s$. This represents the instantaneous receiving and forwarding of a message. Similarly, C_2 and C_3 read and forward the message instantaneously. The effect is that the state space only consists of one state as depicted in figure 4 at the right.

proc $C_1 = \underline{a}|b_s\cdot C_1 + \underline{a}|c_s\cdot C_1;$
$C_2 = b_r|\underline{d}\cdot C_2;$
$C_3 = c_r|\underline{e}\cdot C_3;$
init $\nabla_{\{\underline{a}|c_c|\underline{e},\underline{a}|b_c|\underline{d}\}}(\Gamma_{\{b_r|b_s\rightarrow b_c, c_r|c_s\rightarrow c_c\}}(C_1||C_2||C_3));$

The operator $\nabla_{\{a|c_c|e,a|b_c|d\}}$ allows the two multi-actions $a|c_c|e$ and $a|b_c|d$, enforcing in this way that in both cases these three actions must happen simultaneously.

5 Guideline III: Avoid Parallelism among Components

When models have many concurrent components that can independently perform an action, then the state space of the given model can be reduced by limiting the number of components that can simultaneously perform activity. Ideally, only one component can perform activity at any time. This can for instance be achieved by one central component that allows the other components to do an action in a round robin fashion.

It very much depends on the nature of the system whether this kind of modelling is allowed. If the primary purpose of a system is the calculation of values, sequentialising appears to be defendable. If on the other hand the sub-components are controlling all kinds of devices, then the parallel behaviour of the sub-components might be the primary purpose of the system and sequentialisation can not be used.

In some specification languages explicit avoidance of parallel behaviour between components has been used. For instance Esterel [3] uses micro steps which can be calculated per component. In Promela there is an explicit atomicity command, grouping behaviour in one component that is executed without interleaving of actions of other components [9].

As an example we consider M traffic lights guarding the same number of entrances of a parking lot. See figure 5 for a diagrammatic representation where $M{=}3$. A sensor detects that a car arrives at an entrance. If there is space in the garage, the traffic light shows green for some time interval. There is a detector at the exit, which indicates that a car is leaving. The number of cars in the garage cannot exceed N.

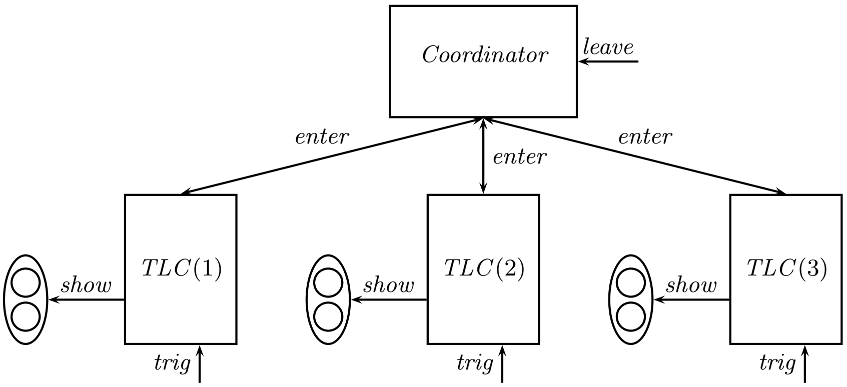

Fig. 5. A parking lot with three entrances

The first model is very simple, but has a large state space. Each traffic light controller (TLC) waits for a trigger of its sensor, indicating that a car is waiting. Using the $enter_s$ action it asks the $Coordinator$ for admission to the garage. If a car can enter, this action is allowed by the co-ordinator and a traffic light cycle starts. Otherwise the $enter_s$ action is blocked. The $Coordinator$ has an internal counter, counting the number of cars. When a $leave$ action takes place, the counter is decreased. When a car is allowed to enter (via $enter_r$), the counter is increased.

$$
\begin{aligned}
\textbf{proc}\ & Coordinator(count{:}\mathbb{N})\\
&\quad = (count{>}0) \rightarrow \underline{leave} \cdot Coordinator(count{-}1)\\
&\quad + (count{<}N) \rightarrow enter_r \cdot Coordinator(count{+}1);\\[1em]
& TLC(id{:}\mathbb{N}^+)\\
&\quad = \underline{trig}(id) \cdot enter_s \cdot \underline{show}(id, green) \cdot \underline{show}(id, red) \cdot TLC(id);
\end{aligned}
$$

$$
\textbf{init}\ \ \nabla_{\{\underline{trig},\underline{show},enter_c,\underline{leave}\}}(\Gamma_{\{enter_s | enter_r \rightarrow enter_c\}}(Coordinator(0) \|\ TLC(1) \| TLC(2) \| TLC(3)));
$$

The state space of this control system grows exponentially with the number of traffic light controllers. In columns 2 and 4 of table 1 the sizes of the state spaces for different M are shown. It is also clear that the number of parking places N only contributes linearly to the state space.

Following the guideline, we try to limit the amount of parallel behaviour in the traffic light controllers. So, we put the initiative in the hands of the co-ordinator in the second model. It assigns the task of monitoring a sensor to one of the traffic light controllers at a time. The traffic controller will poll the sensor, and only if it has been triggered, switch the traffic light to green. After it has done its task, the traffic light controller will return control to the co-ordinator. Of course if the parking lot is full, the traffic light controllers are not activated. Note that in this second example, only one traffic light can show green at any time, which might not be desirable.

$$\begin{array}{ll}
\textbf{proc} & \mathit{Coordinator}(\mathit{count}{:}\mathbb{N}, \mathit{active\_id}{:}\mathbb{N}^+) \\
& \quad = (\mathit{count}{>}0)\rightarrow \underline{\mathit{leave}}\cdot \mathit{Coordinator}(\mathit{count}{-}1, \mathit{active\_id}) \\
& \quad + (\mathit{count}{<}N)\rightarrow \mathit{enter}_s(\mathit{active\_id})\cdot \sum_{b:\mathbb{B}} \mathit{enter}_r(b)\cdot \\
& \qquad\qquad \mathit{Coordinator}(\mathit{count}{+}\mathrm{if}(b,1,0), \mathrm{if}(\mathit{active\_id}{\approx}M, 1, \mathit{active\_id}{+}1));
\end{array}$$

$$\begin{array}{l}
\mathit{TLC}(\mathit{id}{:}\mathbb{N}^+) \\
\quad = \mathit{enter}_r(\mathit{id})\cdot \\
\qquad (\ \underline{\mathit{trig}}(\mathit{id}, \mathit{true})\cdot \underline{\mathit{show}}(\mathit{id}, \mathit{green})\cdot \underline{\mathit{show}}(\mathit{id}, \mathit{red})\cdot \mathit{enter}_s(\mathit{true}) + \\
\qquad\quad \underline{\mathit{trig}}(\mathit{id}, \mathit{false})\cdot \mathit{enter}_s(\mathit{false}) \\
\qquad)\cdot \\
\qquad \mathit{TLC}(\mathit{id});
\end{array}$$

$$\textbf{init}\quad \nabla_{\{\underline{\mathit{trig}}, \underline{\mathit{show}}, \mathit{enter}_c, \underline{\mathit{leave}}\}}(\Gamma_{\{\mathit{enter}_s|\mathit{enter}_r\rightarrow \mathit{enter}_c\}} \\ \qquad (\mathit{Coordinator}(0,1)||\mathit{TLC}(1)||\mathit{TLC}(2)||\mathit{TLC}(3)));$$

As can be seen in table 1 the state space of the second model only grows linearly with the number of traffic lights.

Table 1. State space sizes of parking lot controllers (N: no. of traffic lights, M: no. of parking places)

M	parallel ($N = 10$)	restricted ($N = 10$)	parallel ($N = 100$)	restricted ($N = 100$)
1	44	61	404	601
2	176	122	1,616	1,202
3	704	183	6,464	1,803
4	2,816	244	25,856	2,404
5	11,264	305	103,424	3,005
6	45,056	366	413,696	3,606
10	$11.5\,10^6$	610	$106\,10^6$	6,010

6 Guideline IV: Restrict the Use of Data

The use of data in behavioural models can quickly blow up a state space. Therefore, data should always be looked at with extra care, and if its use can be avoided, this should be done. If data is essential (and it almost always is), then there are several methods to reduce its footprint. Below we give two examples, one where data is categorised and one where buffers are ordered.

In order to reduce the state space of a behavioural model, it sometimes helps to categorise the data in categories, and formulate the model in terms of these categories, instead of individual values. From the perspective of verification, this technique is called abstract interpretation [5]. Using this technique, a given data domain is interpreted in categories, in order to assist the verification process. Here, we advice that the modeller uses the categories in the model, instead of letting the values be interpreted in categories during the verification process. As the modeller generally knows his model best, he also has a good intuition about the appropriate categories.

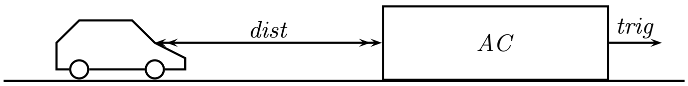

Fig. 6. An advanced approach controller

Consider for example an intelligent approach controller which measures the distance of an approaching car as depicted in figure 6. If the car is expected to pass distance 0 before the next measurement, a trigger signal is forwarded. The farthest distance the approach controller can observe is M. A quite straightforward description of this system is given below. Using the action *dist* the distance to a car is measured, and the action *trig* models the trigger signal. The $\diamond$ operator denotes the else part of a condition.

map $M : \mathbb{N};$
eqn $M = 100;$
proc $AC(d_{prev}{:}\mathbb{N}) = \sum_{d:\mathbb{N}}(d{<}M) \rightarrow (\underline{dist}(d) \cdot (2d{<}d_{prev}) \rightarrow \underline{trig} \cdot AC(M) \diamond AC(d));$
init $AC(M);$

The state space of this system is a staggering $M^2{+}1$ states big, or more concretely 10001 states. This is of course due to the fact that the values of d and d_{prev} must be stored in the state space to enable the evaluation of the condition $2d{<}d_{prev}$. But only the information needs to be recalled whether this condition holds, instead of both values of d and d_{prev}. So, a first improvement is to move the condition backward as is done below, leading to a required $M{+}1$ states, or 101 in this concrete case.

proc $AC_1(d_{prev}{:}\mathbb{N}) = \sum_{d:\mathbb{N}}(d{<}M) \rightarrow ((2d{<}d_{prev}) \rightarrow \underline{dist}(d) \cdot \underline{trig} \cdot AC_1(M)$
$\diamond \underline{dist}(d) \cdot AC_1(d));$
init $AC_1(M);$

But we can go much further, provided it is possible to abstract from the concrete distances. Let us assume that the only relevant information that we obtain from the individual distances is whether the car is far from the sensor or nearby. Note that we abstract from the concrete speed of the car which was used above. The specification of this abstract approach controller AAC is given by:

sort $\mathit{Distance} = \textbf{struct}\ \mathit{near} \mid \mathit{far};$
proc $AAC = \sum_{d:Distance} \underline{\mathit{dist}}(d)\cdot((d{\approx}\mathit{near}){\rightarrow}\underline{\mathit{trig}}\cdot AAC\diamond AAC);$
init $AAC;$

Note that M does not occur anymore in this specification. The state space is now reduced to two states.

Table 2. Number of states of an non ordered/ordered buffer with max. N elements

N	non ordered	ordered
1	2	2
2	5	4
3	16	8
4	65	16
5	326	32
6	$2.0\ 10^3$	64
7	$14\ 10^3$	128
8	$110\ 10^3$	256
9	$986\ 10^3$	512
10	$9.9\ 10^6$	$1.02\ 10^3$
11	$109\ 10^6$	$2.05\ 10^3$
12	$1.30\ 10^9$	$4.10\ 10^3$

As a last example we show the effect of ordering buffers. With queues and buffers different contents can represent the same data. If a buffer is used as a set, the ordering in which the elements are put into the buffer is irrelevant. In such cases it helps to maintain an order on the data structure. As an example we provide a simple process that reads arbitrary natural numbers smaller than N and puts them in a set. The process doing so is given below. The operator $\triangleright$ puts an element in front of a list.

map $N : \mathbb{N};$
$\quad \mathit{insert}, \mathit{ordered\_insert} : \mathbb{N} \times \mathit{List}(\mathbb{N}) \rightarrow \mathit{List}(\mathbb{N});$

var $n, n' : \mathbb{N}; b : \mathit{List}(\mathbb{N});$
eqn $\mathit{insert}(n, b) = \mathit{if}(n \in b, b, n{\triangleright}b);$
$\quad \mathit{ordered\_insert}(n, []) = [n];$
$\quad \mathit{ordered\_insert}(n, n'{\triangleright}b) = \mathit{if}(n{<}n', n{\triangleright}n'{\triangleright}b, \mathit{if}(n{\approx}n', n'{\triangleright}b, n'{\triangleright}$
$\qquad \mathit{ordered\_insert}(n, b)));$
$\quad N = 10;$
proc $B(\mathit{buffer}{:}\mathit{List}(\mathbb{N})) = \sum_{n:\mathbb{N}}(n{<}N){\rightarrow}\underline{\mathit{read}}(n)\cdot B(\mathit{insert}(n, \mathit{buffer}));$

init $B([]);$

If the function *insert* is used, the elements are put into a set in an arbitrary order (more precisely, the elements are prepended). If the function *ordered_insert* is used instead of *insert*, the elements occur in ascending order in the buffer. In table 2 the effect of ordering is shown. Although the state spaces with ordering also grow exponentially, the beneficial effect of ordering does not need further discussion.

7 Guideline V: Specify External Behaviour of Sets of Sub-components

We observed that sometimes the composed behaviour of sets of components can be overly complex, and contains far too many states, even after applying a behavioural reduction. In order to keep the behaviour of such sets of components small, it is useful to first design the desired external behaviour of this set of components, and to subsequently design the behaviour of the components such that they meet this external behaviour. The situation is quite comparable to the implementation of software. If the behaviour is governed by the implementation, a system is often far less understandable and usable, than when a precise specification of the software has been provided first, and the software has been designed to implement exactly the specified behaviour.

The use of external behaviour for various purposes was most notably defended in the realm of protocol specification [13], although keeping the state space small was not one of these purposes. The word service was commonly used in this setting for the external behaviour. More recently, the ASD development method has been proposed, where a system is to be defined by first specifying the external behaviour of a system, which is subsequently implemented [4]. The purpose here is primarily to allow a designer to keep control over his system.

In order to illustrate how specifications can be used to keep external behaviour small, we provide a simple example, and we show how a small difference in the behaviour of the components has a distinctive effect on the complexity in terms of states. From the perspective of the task that the components must perform, the difference in the description looks relatively minor. The example is inspired by the third sliding window protocol in [12] which is a fine example of a set of components that provides the intended task but has a virtually incomprehensible external behaviour.

Our system is depicted in figure 7. The first specification has a complex external behaviour whereas the external behaviour of the second is straightforward. The system consists of a device-monitor and a controller that can be started (*start*) or stopped (*stop*) by an external source. The device-monitor observes the status of a number of devices and sends the defected device number to the controller via the action *broken*. The controller comprises a buffer that stores the status of the devices.

The first specification can be described as follows. The device monitor is straightforward in the sense that it continuously performs actions $broken_s(n)$ for numbers $n<M$. The parameter *buff* represents the buffer by a function from

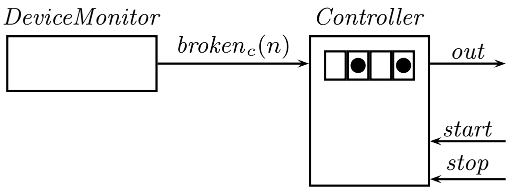

Fig. 7. A system comprises a controller and a device-monitor

natural numbers to booleans. If *buff*(i) is true, it indicates that a fault report has been received for device i. The boolean parameter *bool* indicates whether the controller is switched on or off and the natural number i is the current position in the buffer, which the controller uses to cycle through the buffer elements. It sends an action *out* whenever it encounters an element that is set to *true*. The internal action *int* takes place when the controller moves to investigate the next buffer place.

map $M{:}\mathbb{N}^+$;
eqn $M{=}2$;
map $\mathit{buff}_0{:}\mathbb{N}{\to}\mathbb{B}$;
eqn $\mathit{buff}_0 = \lambda n{:}\mathbb{N}.\mathit{false}$;
proc $\mathit{DeviceMonitor} = \sum_{n:\mathbb{N}}(n{<}M){\to}\mathit{broken}_s(n).\mathit{DeviceMonitor}$;
$\mathit{Controller}(\mathit{buff}{:}\mathbb{N}{\to}\mathbb{B}, \mathit{bool}{:}\mathbb{B}, i{:}\mathbb{N})$
$\quad = \sum_{n:\mathbb{N}} \mathit{broken}_r(n)\cdot \mathit{Controller}(\mathit{buff}[n{\to}\mathit{true}], \mathit{bool}, i)$
$\quad + (\neg\mathit{buff}(i){\wedge}\mathit{bool}){\to}\underline{\mathit{stop}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{false}, i)$
$\quad + (\neg\mathit{bool}){\to}\underline{\mathit{start}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{true}, i)$
$\quad + (\mathit{buff}(i){\wedge}\mathit{bool}){\to}\underline{\mathit{out}}\cdot \mathit{Controller}(\mathit{buff}[i{\to}\mathit{false}], \mathit{bool}, (i{+}1) \bmod M)$
$\quad + (\neg\mathit{buff}(i){\wedge}\mathit{bool}){\to}\mathit{int}\cdot \mathit{Controller}(\mathit{buff}, \mathit{bool}, (i{+}1) \bmod M)$
init $\tau_{\{\mathit{broken}_c, \mathit{int}\}}(\nabla_{\{\mathit{broken}_c, \underline{\mathit{out}}, \underline{\mathit{start}}, \underline{\mathit{stop}}, \mathit{int}\}}(\Gamma_{\{\mathit{broken}_r|\mathit{broken}_s \to \mathit{broken}_c\}}($
$\qquad \mathit{Controller}(\mathit{buff}_0, \mathit{false}, 0)||\mathit{DeviceMonitor})))$;

The total number of devices is denoted by M. All positions of *buff* are initially set to *false* as indicated by the lambda expression $\lambda n{:}\mathbb{N}.\mathit{false}$. In this specification the controller blocks the *stop* request if there is a defected device at index i of the buffer forming a dependency between external and internal behaviour. If we calculate the state space of the external behaviour of this system with $M = 2$ and apply a branching bisimulation reduction [7], we obtain the state space depicted in figure 8 at the left. Note that the behaviour is remarkably complex. In particular a number of τ-transitions complicate the transition system. But they cannot be removed as they are essential for the perceived external behaviour of the system. Table 3 provides the number of states produced as a function of the number of devices monitored in the system. The table shows that the state space of the original system and the state space capturing the external behaviour are

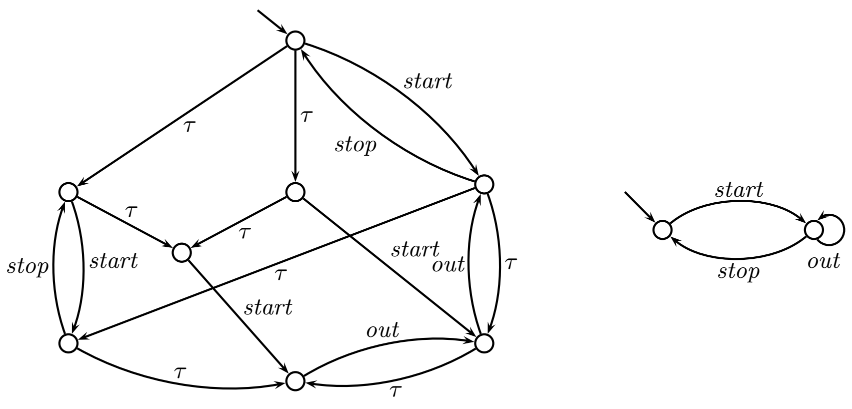

Fig. 8. The system external behaviour

comparable. This indicates a complex external behaviour that might complicate verification with external parties and makes understanding the behaviour quite difficult. It might be amazing that the external state space of the system is large. Actual expectation is that it should be small, matching the specification below, depicted in the transition system in figure 8 at the right.

proc $Stopped = \underline{start}{\cdot}Started;$
$\quad\quad Started = \underline{out}{\cdot}Started + \underline{stop}{\cdot}Stopped;$
init $Stopped;$

Investigation of the cause of the difference between the actual and the expected sizes of the transition systems leads to the conclusion that blocking the *stop* action when $buff(i)$ is true is the cause of the problem. If we remove this from the condition of the stop action, we obtain the mCRL2 specification of the *DeviceMonitor* process below. In this specification the *stop* request is processed independently from the rest of the behaviour.

Table 3. Sizes of the original and external state space of the monitor controllers

M	No. of original states		No. of external states	
	1st spec	2nd spec	1st spec	2nd spec
1	4	4	2	2
2	16	16	8	2
3	48	48	16	2
4	128	128	32	2
5	320	320	64	2
6	768	768	128	2
10	$20.5\ 10^3$	$20.5\ 10^3$	$2.48\ 10^3$	2

$$\begin{array}{lll}
\textbf{proc} & \multicolumn{2}{l}{\mathit{DeviceMonitor} = \sum_{n:\mathbb{N}}(n{<}M)\to \mathit{broken}_s(n).\mathit{DeviceMonitor};} \\
& \multicolumn{2}{l}{\mathit{Controller}(\mathit{buff}{:}\mathbb{N}\to\mathbb{B}, \mathit{bool}{:}\mathbb{B}, i{:}\mathbb{N})} \\
& = & \sum_{n:\mathbb{N}} \mathit{broken}_r(n)\cdot \mathit{Controller}(\mathit{buff}[n\to \mathit{true}], \mathit{bool}, i) \\
& + & \mathit{bool}\to \underline{\mathit{stop}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{false}, i) \\
& + & (\neg \mathit{bool})\to \underline{\mathit{start}}\cdot \mathit{Controller}(\mathit{buff}, \mathit{true}, i) \\
& + & (\mathit{buff}(i)\wedge \mathit{bool})\to \underline{\mathit{out}}\cdot \mathit{Controller}(\mathit{buff}[i\to \mathit{false}], \mathit{bool}, (i{+}1) \bmod M) \\
& + & (\neg \mathit{buff}(i)\wedge \mathit{bool})\to \mathit{int}\cdot \mathit{Controller}(\mathit{buff}, \mathit{bool}, (i{+}1) \bmod M)
\end{array}$$

As can be seen from table 3, the number of states of the non-reduced model remains the same. However, the reduced behaviour is exactly the one depicted in figure 8 at the right for any constant M.

8 Conclusion

We have shown that different specification styles can substantially influence the number of states of a system. We believe that an essential skill of a behavioural modellist is to make models such that the insight that is required can be obtained. If a system is to be designed such that it provably satisfies a number of behavioural requirements, then the behaviour must be sufficiently small to be verified. If an existing system is modelled to obtain insight in its behaviour, then on the one hand the model should reflect the existing system sufficiently well, but on the other hand the model of the system should be sufficiently simple to allow to answer relevant questions about the behaviour of the system.

As far as we can see hardly any attention has been paid to the question how to make behavioural models such that they can be analysed. All attention appears to be directed to the question of how to analyse given models better. But it is noteworthy that it is very common in other modelling disciplines to let models be simpler than reality. For instance in electrical engineering models are as much as possible reduced to sets of linear differential equations. In queueing theory, only a few queueing models can be studied analytically, and therefore, it is necessary to reduce systems to these standard models if analytical results are to be obtained.

We provided five guidelines, based on our experience with building models of various systems. There is no claim that this set is complete, or even that these five guidelines are the most important model reduction techniques. What we hope is that this paper will induce research such that more reduction techniques will be uncovered, described, classified and subsequently become a standard ingredient in teaching behavioural modelling.

Acknowledgements. We thank Sjoerd Cranen, Helle Hansen, Jeroen Keiren, Matthias Raffelsieper, Frank Stappers, Ron Swinkels, Marco van der Wijst, and Tim Willemse for their useful comments on the text.

References

1. Acharya, S., Franklin, M., Zdonik, S.: Balancing push and pull for data broadcast. In: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, pp. 183–194 (1997)

2. Arbab, F.: Reo: A Channel-based coordination model for component composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming 19, 87–152 (1992)
4. Broadfoot, G.H.: ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 548–551. Springer, Heidelberg (2005)
5. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(2), 253–291 (1997)
6. Groote, J.F., Kouters, T.W.D.M., Osaiweran, A.A.H.: Specification Guidelines to avoid the State Space Explosion Problem. Technical Report 10-14, Computer Science Reports, Department of Computer Science, Eindhoven University of Technology, Eindhoven, The Netherlands (2010)
7. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3), 555–600 (1996)
8. Groote, J.F., Mathijssen, A.H.J., Reniers, M.A., Usenko, Y.S., van Weerdenburg, M.J.: Analysis of distributed systems with mCRL2. In: Alexander, M., Gardner, W. (eds.) Process Algebra for Parallel and Distributed Processing, pp. 99–128. Chapman and Hall (2009)
9. Holzmann, G.J.: The SPIN model checker. Primer and reference manual. Addison-Wesley (2003)
10. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. Int. Journal on Software Tools for Technology Transfer 1(12), 134–152 (1997)
11. Lin, F.J., Chu, P.M., Liu, M.T.: Protocol verification using reachability analysis: The state space explosion problem and relief strategies. ACM SIGCOMM Computer Communication Review 17(5), 126–135 (1987)
12. Tanenbaum, A.S.: Computer networks, 2nd edn. Prentice Hall (1988)
13. Vissers, C.A., Logrippo, L.: The importance of the service concept in the design of data communications protocols. In: Diaz, M. (ed.) Protocol Specification, Testing and Verification (Proc. of the IFIP WG 6.1 Fifth International Workshop on Protocol Sepcification, Testing and Verification), pp. 3–17. Elsevier North Holland (1986)
14. Vissers, C.A., Scollo, G., van Sinderen, M., Brinksma, E.: Specification styles in distributed systems design and verification. Theoretical Computer Science 89, 179–206 (1991)
15. (2010), `http://www.mcrl2.org`