

Specification Guidelines to avoid the State Space Explosion Problem

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

J.F.Groote@tue.nl, T.W.D.M.Kouters@student.tue.nl, A.A.H.Osaiweran@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 article we summarise our experience by providing seven specification guidelines. For each guideline we provide an application from the realm of traffic light controllers for which we provide a ‘bad’ model with a large state space, and a ‘good’ model with a small state space.

1 Introduction

Behavioural specification of computer systems, distributed algorithms, communication protocols, business processes, etc. is gaining popularity. Behavioural specification refers here to discrete behaviour, such as the exchange of messages, reading digital sensors and switching lights on and off. Specifying the discrete behaviour of systems before construction helps focussing on the behaviour, without simultaneously being bothered with programming or other implementation details. This allows for clearer specification of systems, both increasing usability and reducing flaws in the code. Very importantly, it also helps to provide adequate documentation.

These days, we and others have ample experience in system design through behavioural specification. There are for instance well-established workshops and journals on this topic [8, 9]. 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.

As most systems are constructed without using any behavioural verification, it is often the case that the behaviour of existing systems is problematic and not well understood. This provides the second use of behavioural specification, namely to model existing systems to obtain a better understanding of what they are doing. The model can be investigated to prove that the system always satisfies certain requirements. There are no other ways to obtain such insight. For instance exhaustive testing can increase the confidence that a system satisfies a certain requirement, but it will never provide certainty.

When verifying system behaviour, the state space explosion problem kicks in. If we do not pay attention, 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 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*. This is comparable to ‘design for testability’, which is mainly used in esp.

microelectronics to allow to test a product for production flaws [24], and which is slowly finding its way into software engineering [23].

What we propose is that systems are designed such that the state spaces of their behavioural models are small. This does impose certain restrictions on how systems can behave. For instance, maintaining local copies of data throughout a system blows up the state space, and is therefore not recommended. When modelling existing systems, we advocate that sometimes the models are shaped such that the state space does not grow too much, even if this means that the actual system is not completely faithfully modelled. It is better to obtain insight with an approximate model, than getting no insight at all. Note that this approach is very common in other engineering disciplines.

Compared to the development of state space reduction techniques, design for verifiability is a barely addressed issue. The best we could find is [17], but it primarily addresses improvements in verification technology, too. Specification styles from the perspective of expressiveness have been addressed [22], but verifiability is also not really an issue here.

In this article we provide seven specification guidelines that we learned by specifying complex realistic systems (e.g. traffic control systems, medical equipment, domestic appliances, communication protocols). For each specification guideline we provide an application from the domain of traffic light controllers. The reason for taking this domain is that we felt the need to write this article, when working on a traffic light controller for a crossing with 12 traffic lights and 24 road sensors. The initial model was so complex that it was even difficult to verify the correctness requirements when traffic was restricted to 2 lanes. After rewriting the model, all correctness requirements for the full control system could be verified without any restriction on the use of traffic lanes, road sensors or traffic lights.

For each guideline we give two examples. The first one does not take the guideline into account and the second does. Generally, the first specification is very natural, but leads to a large state space. Then we provide a second specification that uses the guideline. 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’ specification 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 [12, 25].

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 confluence and determinacy, and external specifications may be foreign to most specifiers.

Although we provide a number of guidelines that we believe are really important for the behavioural modeller, we do not claim completeness. Without doubt we have overlooked a number of specification strategies that are helpful in keeping state spaces small. 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.

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.

2 A short introduction into mCRL2

Before getting to the design guidelines for avoiding state space explosion we give a short exposition of the specification language mCRL2. We only restrict ourselves to the those parts of the language that we need in this paper. Further information can be obtained from various sources, but good places to start are [12, 25]. Especially, at the website www.mcr12.org the toolset for mCRL2 is available, as well as lots of documentation and examples.

The abbreviation mCRL2 stands for micro Common Representation Language 2. It is a specification language that can be used to specify and analyse the behaviour of distributed systems and protocols.

mCRL2 is based on the Algebra of Communicating Processes (ACP, [3]), which is extended to include data and time.

We first describe the data types. Data types consist of sorts and functions working upon these sorts. There are standard data types such as the booleans (\mathbb{B}), the positive numbers (\mathbb{N}^+) and the natural numbers (\mathbb{N}). All sorts represent their mathematical counterpart. E.g. the number of natural numbers is unbounded.

All common operators on the standard data sorts are available. We use \approx for equality between elements of a data type in order to avoid confusion with $=$ which we use as equality between processes. We also use $if(c, t, u)$ representing the term t if the condition c holds, and u if c is not valid.

For any sort D , the sorts $List(D)$ and $Set(D)$ contain the lists and sets over domain D . Prepending an element d to a list l is denoted by $d\blacktriangleright l$. Getting the last element of a list is denoted as $rhead(l)$. The remainder of the list after removing the last element is denoted as $rtail(l)$. The length of a list is denoted by $\#(l)$. Testing whether an element is in a set s is denoted as $d \in s$. The set with only element d is denoted by $\{d\}$. Set union is written as $s_1 \cup s_2$ and set difference as $s_1 \setminus s_2$.

Given two sorts D_1 and D_2 , the sort $D_1 \rightarrow D_2$ contains all functions from the elements from D_1 to elements of D_2 . We use standard lambda notation to represent functions. E.g. $\lambda x:\mathbb{N}.x+1$ is the function that adds 1 to its argument. For a function f we use the notation $f[t \rightarrow u]$ to represent the function f , except that if $f[t \rightarrow u]$ is applied to t , the value u is returned. We call $f[t \rightarrow u]$ a function update.

Besides using standard types and type constructors such as *List* and *Set*, users can define their own sorts. In this paper we most often use user defined sorts with a finite number of elements. A typical example is the declaration of a sort containing the three aspects *green*, *yellow* and *red* of a traffic light.

```
sort   Aspect = struct green | yellow | red;
```

A more complex user defined sort that we use is a message containing a number that can either be active or passive. The number in each message can be obtained by applying the function *get_number* to a message. The function *is_active* is true when applied to a message of the form *active*(n) and false otherwise.

```
sort   Message = struct active(get_number: $\mathbb{N}$ )?is_active | passive(get_number: $\mathbb{N}$ );
```

Using the **map** keyword elements of data domains can be declared. By introducing an equation the element can be declared equal to some expression. An example of its use is the following. The constant n is declared to be equal to 3 and f is equal to the function that returns false for any natural number.

```
map   n :  $\mathbb{N}$ ;
        f :  $\mathbb{N} \rightarrow \mathbb{B}$ ;
eqn   n = 3;
        f =  $\lambda x:\mathbb{N}.false$ ;
```

This concise explanation of data types is enough to understand the paper.

The use of data is the primary source why state spaces grow out of hand. A system with only two 32 bit integers has $1.8 \cdot 10^{19}$ states which for quite some time to come will not fit into the memory of any computer (unless compression techniques are used). It is therefore very important to restrict the possible values data types can have. Often it is wise to model data domains in abstract categories. E.g. instead of using a height in millimetres, one can abstract this to the three values *low*, *middle* and *high*.

The behaviour of systems is characterised by atomic actions. Actions can represent any elementary activity. Here, they typically represent setting a traffic light to a particular colour, getting a signal from a sensor or communicating among components. Actions can carry data parameters. For example *trig*($id, false$) could typically represent that the sensor with identifier id was not triggered (indicated by the boolean *false*).

In an mCRL2 specification, actions must be declared as indicated below, where the types indicate the sorts of the data parameters that they carry.

```
act   trig :  $\mathbb{N} \times \mathbb{B}$ ;
        send : Message;
        my_turn;
```

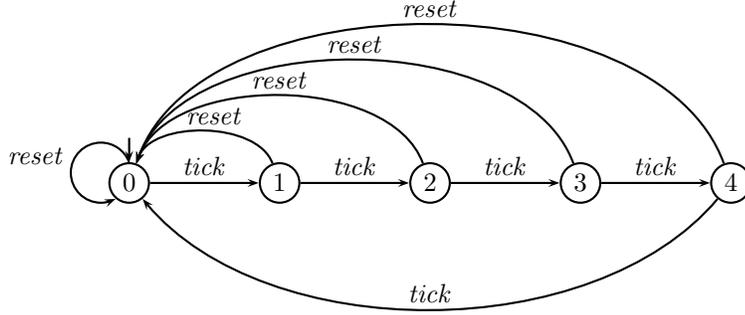


Figure 1: The transition system of the process *Counter*

In the examples in this article we have omitted these declarations as they are clear from the context.

If two actions a and b happen at the same time, then this is called a multi-action, which is denoted as $a|b$. The operator ‘|’ is called the multi-action composition operator. Any number of actions can be combined into a multi-action. The order in which the actions occur has no significance. So, $a|b|c$ is the same multi-action as $c|a|b$. The empty multi-action is written as τ . It is an action that can happen, but which cannot directly be observed. It is also called the hidden or internal action. The use of multi-actions can be quite helpful in reducing the state space, as indicated in guideline II in section 5.

Actions and multi-actions can be composed to form processes. The choice operator, used as $p + q$ for processes p and q , allows the process to choose between two processes. The first action that is done determines the choice. The sequential operator, denoted by a dot (‘.’), puts two behaviours in sequence. So, the process $a \cdot b + c \cdot d$ can either perform action a followed by b , or c followed by d .

The *if-then-else* operator, $c \rightarrow p \diamond q$, allows the condition c to determine whether the process p or q is selected. The else part can always be omitted. We then get the conditional operator of the form $c \rightarrow p$. If c is not valid, this process cannot perform an action. It deadlocks. This does not need to be a problem because using the $+$ operator alternative behaviour may be possible.

The following example shows how to specify a simple recursive process. It is declared using the keyword **proc**. It is a timer that cyclically counts up till four using the action *tick*, and can be *reset* at any time. Note that the name of a process, in this case *Counter*, can carry data parameters. The initial state of the process is *Counter*(0), i.e., the counter starting with argument 0. Initial states are declared using the keyword **init**. As explained below, we underline actions, if they are not involved in communication between processes.

```

proc   Counter( $n:\mathbb{N}$ )
        = ( $n < 4$ )  $\rightarrow$  tick · Counter( $n+1$ )  $\diamond$  tick · Counter(0)
        + reset · Counter(0);
init   Counter(0);

```

In figure 1 the transition system of the counter is depicted. It consists of five states and ten transitions. By following the transitions from state to state a run through the system can be made. Note that many different runs are possible. A transition system represents all possible behaviours of the system, rather than one or a few single runs. The initial state is state 0, which has a small incoming arrow to indicate this. The precise mapping from algebraic processes is given by the operational semantics described in [13]. We will not go into this precise mapping, but it is quite straightforward. The transition systems referred to in this article are all generated using the mCRL2 toolset [25].

Sometimes, it is required to allow a choice in behaviour, depending on data. E.g., for the counter it can be convenient to allow to set it to any value larger than zero and smaller than five. Using the choice operator this can be written as

$$\underline{set}(1) \cdot \textit{Counter}(1) + \underline{set}(2) \cdot \textit{Counter}(2) + \underline{set}(3) \cdot \textit{Counter}(3) + \underline{set}(4) \cdot \textit{Counter}(4)$$

Especially, for larger values this is inconvenient. Therefore, the sum operator has been introduced. It is written as $\sum_{x:\mathbb{N}} p(x)$ and it represents a choice among all processes $p(x)$ for any value of x . The sort \mathbb{N} is

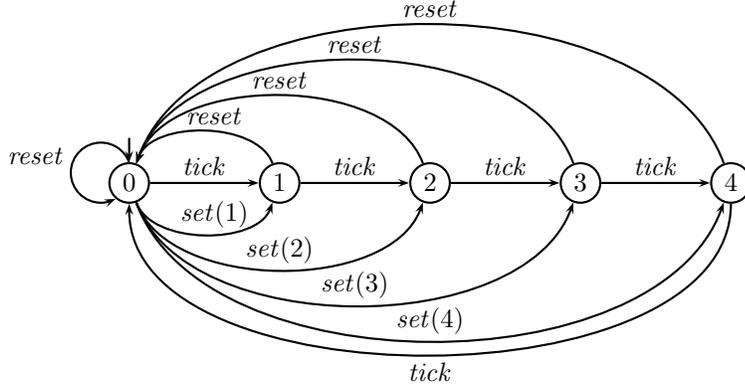


Figure 2: The *Counter* extended with *set* transitions

just provided here as an example, but can be any arbitrary sort. Note that the sort in the sum operator can be infinite. To generate a finite state space, this infinite range must be restricted, for instance by a condition. The example above uses such a restriction and becomes:

$$\sum_{x:\mathbb{N}} (0 < x \wedge x < 5) \rightarrow \underline{set}(x) \cdot Counter(x)$$

Just for the sake of completeness, we formulate the example of the counter again, but now with this additional option to set the counter, which can only take place if n equals 0. This example is a very typical sequential process (sequential in the meaning of not parallel). In figure 2 we provide the state space of the extended counter.

```

proc   Counter( $n:\mathbb{N}$ )
        = ( $n < 4$ )  $\rightarrow$  tick · Counter( $n+1$ )  $\diamond$  tick · Counter(0)
        +  $\sum_{x:\mathbb{N}} (n \approx 0 \wedge 0 < x \wedge x < 5) \rightarrow \underline{set}(x) \cdot Counter(x)$ 
        + reset · Counter(0);
init   Counter(0);

```

Processes can be put in parallel with the parallel operator \parallel to model a concurrent system. The behaviour of $p \parallel q$ represents that the behaviour of p and q is parallel. It is an interleaving of the actions of p and q where it is also possible that the actions of p and q happen at the same time in which case a multi-action occurs. So, $a \parallel b$ represents that actions a and b are executed in parallel. This behaviour is equal to $a \cdot b + b \cdot a + a|b$.

Parallel behaviour is the second main source of a state space explosion. The number of states of $p \parallel q$ is the product of the number of states of p and q . The state space of n processes that each have m states is m^n . For n and m larger than 10 this is too big to be stored in the memory of almost any computer in an uncompressed way. Using the allow operator introduced in the next paragraph, the number of reachable states can be reduced substantially. But without care the number of states of parallel systems can easily grow out of control.

In order to let two parallel components communicate, the communication operator Γ_C and the allow operator ∇_V are used where C is a set of communications and V is a set of data free multi-actions. The idea behind communication is that if two actions happen at the same time, and carry the same data parameters, they can communicate to one action. In this article we use the convention that actions with a subscript r (from receive) communicate to actions with a subscript s (from send) into an action with subscript c (from communicate). Typically, we write $\Gamma_{\{a_r|a_s \rightarrow a_c\}}(p \parallel q)$ to allow action a_r to communicate with a_s resulting in a_c in a process $p \parallel q$. In order to make the distinction between internal communicating actions and external actions clearer, we underline all external actions in specifications (but not in the text or in the diagrams). 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.

To enforce communication, we must also express that actions a_s and a_r cannot happen on their own. The allow operator explicitly allows certain multi-actions to happen, and blocks all others. So, in the example from the previous paragraph, we must add $\nabla_{\{a_c\}}$ to block a_r and a_s enforcing them to communicate into a_c . So, a typical expression putting behaviours p and q in parallel, letting them communicate via action a , is:

$$\nabla_{\{a_c\}}(\Gamma_{\{a_r|a_s \rightarrow a_c\}}(p \parallel q))$$

Of course, more processes can be put in parallel, and more actions can be allowed to communicate.

Actions that are the result of a communication are in general internal actions in the sense that they take place between components of the system and do not communicate with the outside world. Using the hiding operator τ_I actions can be made invisible. So, for a process that consists of a single action a , $\tau_{\{a\}}(a)$ is the empty multi-action τ , an action that does happen, but which cannot directly be observed.

If a system has internal actions, then the behaviour can be reduced. For instance in the process $a \cdot \tau \cdot p$ it is impossible to observe the τ , and this behaviour is equivalent to $a \cdot p$. The most common behavioural reductions are weak bisimulation and branching bisimulation [18, 11]. We will not explain these equivalences here in detail. For us it suffices to know that they reduce the behaviour of a system to a unique minimal transition system preserving the essence of the external behaviour. This result is called the transition system modulo weak/branching bisimulation. This reduction is often substantial.

3 Overview of design guidelines

In this section we give a short description of the seven guidelines that we present in this paper. Each guideline is elaborated in its own section with an example where the guideline is not used, and an intuitively equivalent description where the guideline is used. We provide information on the resulting state spaces, showing why the use of the guideline is advantageous.

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 Confluence and determinacy. When parallel behaviour cannot be avoided, it is useful to model such that the behaviour is τ -confluent. In this case τ -prioritisation can be applied when generating the state space, substantially reducing the size of the state space. Modelling a system such that it is τ -confluent is not easy. A good strategy is to strive for determinacy of behaviour. This means that the ‘output’ behaviour of a system must completely be determined by the ‘input’. This is guaranteed whenever an internal action (e.g. receiving or sending a message from/to another component) can be done in a state of a single component, then no other action can be done in that state.

- V **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*. Avoid buffers and queues getting filled, and if not avoidable try to apply confluence and τ -prioritisation. 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, substantially reducing the state footprint.
- VI **Compositional design and reduction.** If a system is composed out of more components, it can be fruitful to combine them in a stepwise manner, and reduce each set of composed components using an appropriate behavioural equivalence. This works well if the composed components do not have different interfaces that communicate via not yet composed components. So typically, this method does not work when the components communicate in a ring topology, but it works very nicely when the components are organised as a tree.
- VII **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.

4 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.

Furthermore, we note that there is also a discussion of information pulling versus information pushing in distributed system design from a completely different perspective [1]. Here, the goal is to minimise response times of distributed systems. If information when needed must be requested (=pulled) from other processes in a system, the system can become sluggish. But on the other hand, if all processes inform all other processes about every potentially interesting event, communication networks can be overloaded, also leading to insufficient responsiveness. Note that we prefer the verb ‘to poll’ over ‘to pull’, because it describes better that information is repeatedly requested.

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 to red again. For setting the aspect 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 3 this system is drawn.

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

```
sort   Aspect = struct green | yellow | 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. 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

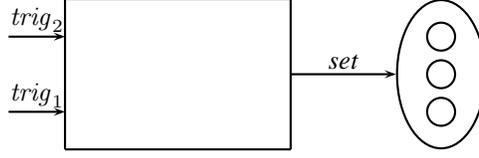


Figure 3: A simple traffic light with two sensors

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.

```

proc   Push( $b_1, b_2: \mathbb{B}, c: \text{Aspect}$ )
        =  $\text{trig}_1 \cdot \text{Push}(\text{true}, b_2, c)$ 
        +  $\text{trig}_2 \cdot \text{Push}(b_1, \text{true}, c)$ 
        +  $(b_1 \wedge b_2 \wedge c \approx \text{red}) \rightarrow \text{set}(\text{green}) \cdot \text{Push}(\text{false}, \text{false}, \text{green})$ 
        +  $(c \approx \text{green}) \rightarrow \text{set}(\text{yellow}) \cdot \text{Push}(b_1, b_2, \text{yellow})$ 
        +  $(c \approx \text{yellow}) \rightarrow \text{set}(\text{red}) \cdot \text{Push}(b_1, b_2, \text{red});$ 
init   Push(false, false, red);

```

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 $\text{trig}_1(b)$ and $\text{trig}_2(b)$. 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*.

```

proc   Poll =  $\text{trig}_1(\text{false}) \cdot \text{Poll} + \text{trig}_1(\text{true}) \cdot \text{Poll}_1;$ 
        Poll1 =  $\text{trig}_2(\text{false}) \cdot \text{Poll}_1 + \text{trig}_2(\text{true}) \cdot \text{Poll}_2;$ 
        Poll2 =  $\text{set}(\text{green}) \cdot \text{set}(\text{yellow}) \cdot \text{set}(\text{red}) \cdot \text{Poll};$ 
init   Poll;

```

The transition systems of both systems are drawn in figure 4. 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.

5 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 [16].

To illustrate the effectiveness of global synchronous communication, we provide the system in figure 5. 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. Non-deterministic forwarding is used, to make the application of confluence impossible (see guideline IV). 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

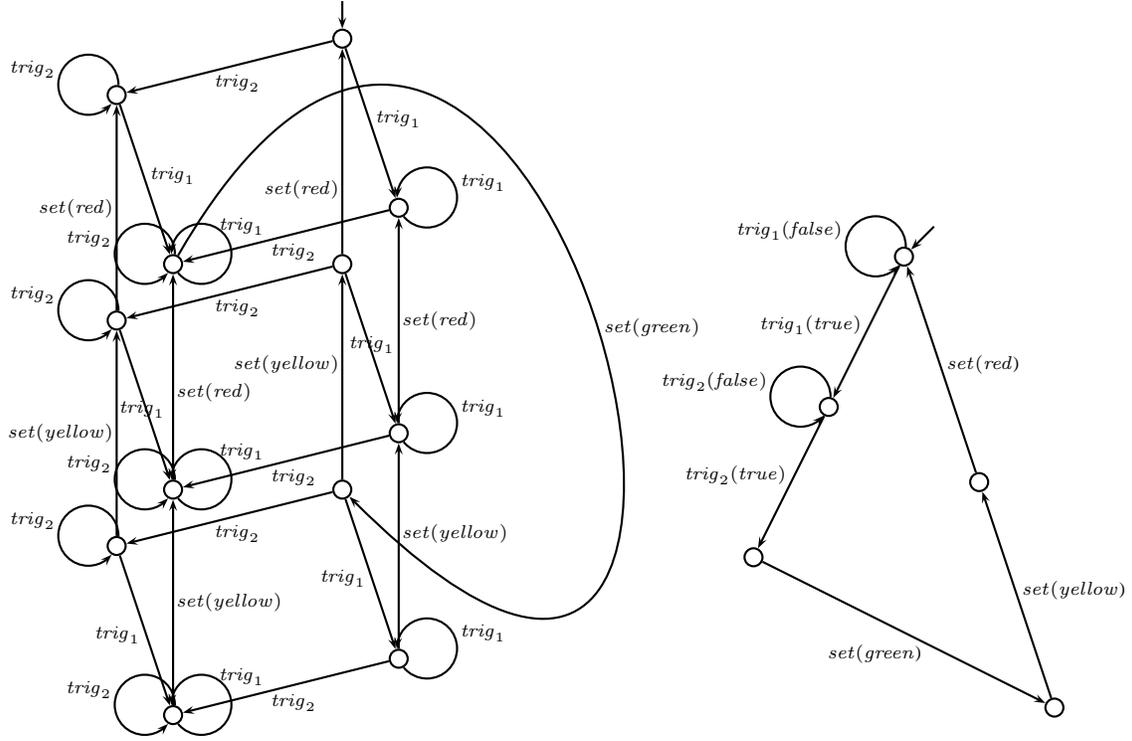


Figure 4: Transition systems of push/poll processes

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.

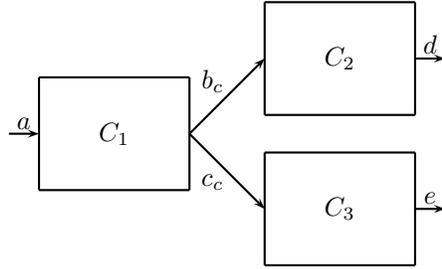


Figure 5: Synchronous/asynchronous message passing

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 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 to become b_c , and c_r and c_s become c_c . The behaviour of this system contains 8 states and is depicted in figure 6 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 $\nabla_{\{\underline{a}, b_c, c_c, \underline{d}, \underline{e}\}} (\Gamma_{\{b_r | b_s \rightarrow b_c, c_r | c_s \rightarrow c_c\}} (C_1 || C_2 || C_3));$

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

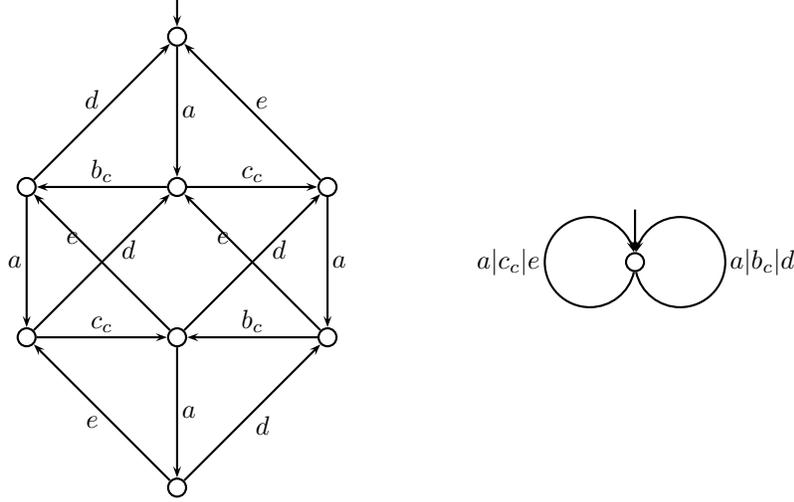


Figure 6: Transition systems of a synchronous and an asynchronous process

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 6 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 $\nabla_{\{\underline{a}|c_c|\underline{e}, \underline{a}|b_c|\underline{d}\}}$ 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.

6 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 defensible. 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 [4] 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 [15].

As an example we consider M traffic lights guarding the same number of entrances of a parking lot. See figure 7 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 .

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

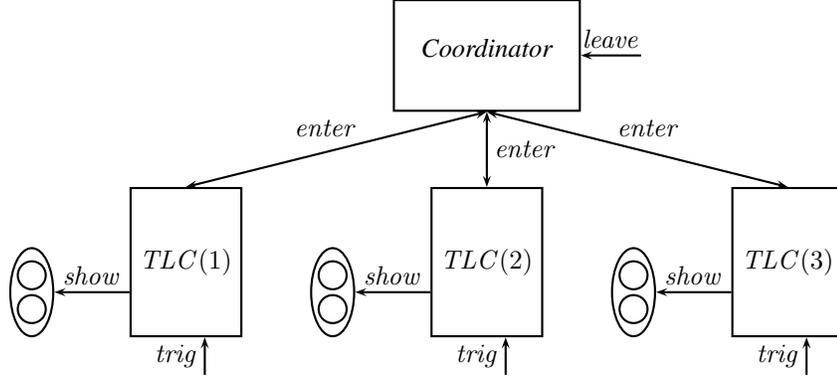


Figure 7: A parking lot with three entrances

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.

proc $Coordinator(count:\mathbb{N})$
 $= (count > 0) \rightarrow \underline{leave} \cdot Coordinator(count - 1)$
 $+ (count < N) \rightarrow enter_r \cdot Coordinator(count + 1);$

$TLC(id:\mathbb{N}^+)$
 $= \underline{trig}(id) \cdot enter_s \cdot \underline{show}(id, green) \cdot \underline{show}(id, red) \cdot TLC(id);$

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

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.

proc $Coordinator(count:\mathbb{N}, active_id:\mathbb{N}^+)$
 $= (count > 0) \rightarrow \underline{leave} \cdot Coordinator(count - 1, active_id)$
 $+ (count < N) \rightarrow enter_s(active_id) \cdot \sum_{b:\mathbb{B}} enter_r(b) \cdot$
 $Coordinator(count + \text{if}(b, 1, 0), \text{if}(active_id \approx M, 1, active_id + 1));$

$TLC(id:\mathbb{N}^+)$
 $= enter_r(id) \cdot$
 $(\underline{trig}(id, true) \cdot \underline{show}(id, green) \cdot \underline{show}(id, red) \cdot enter_s(true) +$
 $\underline{trig}(id, false) \cdot enter_s(false)$
 $) \cdot$
 $TLC(id);$

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

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

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	$11.5 \cdot 10^6$	610	$106 \cdot 10^6$	6,010

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

7 Guideline IV: Confluence and determinacy

In [14, 18] it is described how τ -confluence and determinacy can be used to assist process verification. By modelling such that a system is τ -confluent, verification can become substantially easier. The formulations in [14, 18] are slightly different; we use the formulation from [14] because it is more suitable for verification purposes.

A transition system is τ -confluent iff for every state s that has an outgoing τ and an outgoing a -transition, $s \xrightarrow{\tau} s'$ and $s \xrightarrow{a} s''$, respectively, there is a state s''' such that $s' \xrightarrow{a} s'''$ and $s'' \xrightarrow{\tau} s'''$. This is depicted in figure 8. Note that a can also be a τ , but then the states s' and s'' must be different.

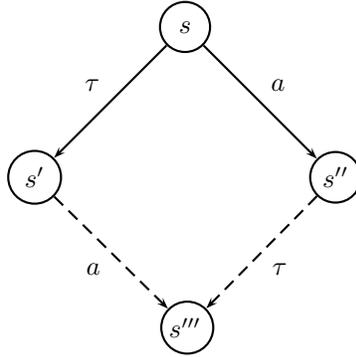


Figure 8: Confluent case

When traversing a state space of a τ -confluent transition system, it is allowed to ignore all outgoing transitions from a state that has at least one outgoing τ -transition, except one outgoing τ -transition. This operation is called τ -prioritisation. It preserves branching bisimulation equivalence [11] and therefore almost all behaviourally interesting properties of the state space. There is one snag, namely that if the resulting τ transitions form a loop, then the outgoing transitions of one of the states on the loop must be preserved. The first algorithm to generate a state space while applying τ -prioritisation is described in [5]. When τ -prioritisation has been applied to a transition system, large parts of the ‘full’ state space have become unreachable. Of the remaining reachable states, many have a single outgoing τ -transition $s \xrightarrow{\tau} s'$. The states s and s' are branching bisimilar and can be mapped onto each other, effectively removing one more state. Furthermore, all states on a τ -loop are branching bisimilar and can therefore be merged into one state, too.

If a state space is τ -confluent, then τ -prioritisation can have a quite dramatic reduction of the size of the state space. This technique allows to generate the prioritised state space of highly parallel systems with thousands of components. In figure 9 a τ -confluent transition system is depicted before and after application of τ -prioritisation, and the subsequent merging of branching bisimilar states.

To employ τ -prioritisation, a system must be defined such that it is τ -confluent. The main rule of thumb is to take care that if an internal action can be performed in a state of a component, no other action can be done in that state. These internal actions include sending information to other components. If data is

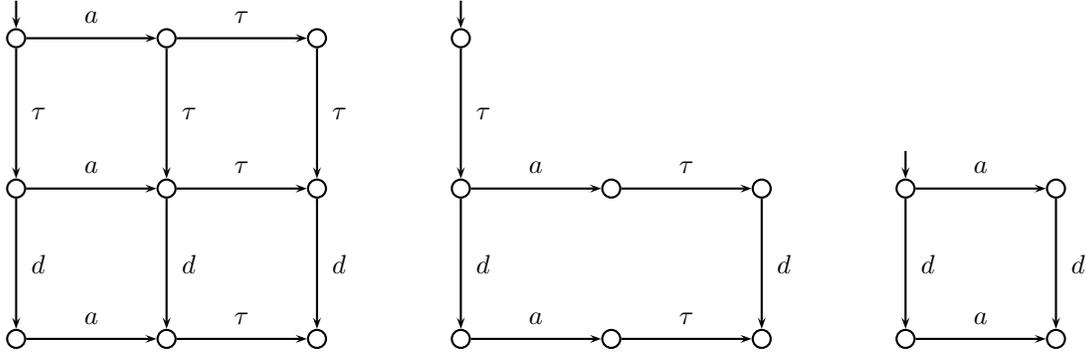


Figure 9: The effect of τ -prioritisation and branching bisimulation compression

received, it must be received from only one component. A selection among different components offering data is virtually always non confluent. Note that in particular pushing information generally destroys confluence. Pushed information must always be received, so, in particular it must be received while internal choices are made and information is forwarded.

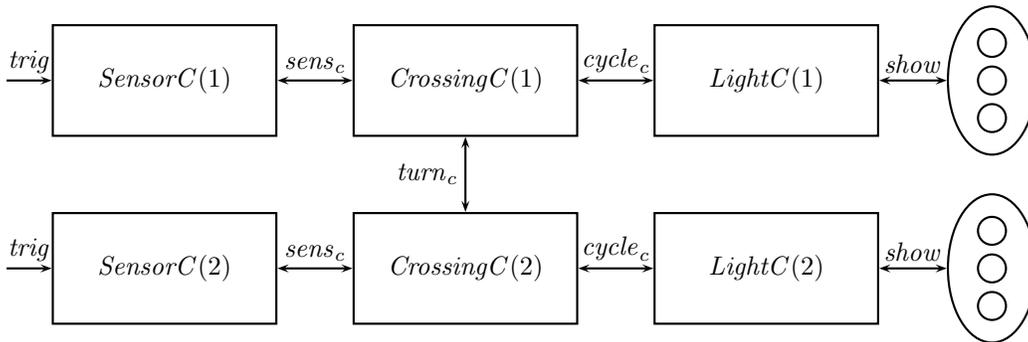


Figure 10: A simple traffic light with two triggers

We model a simple crossing system that contains two traffic lights. First, we are not bothered about confluence. Each traffic light has a sensor indicating that traffic is waiting. We use a control system with 6 components (see figure 10).

For each traffic light we have a sensor controller $SensorC$, a crossing controller $CrossingC$, and a traffic light controller $LightC$. The responsibility of the first is to detect whether the sensor is triggered, using the action $trig$, and forward its occurrence using the action $sens$ to $CrossingC$. The crossing controller takes care that after receiving a $sens$ message, it negotiates with the other crossing controller whether it can turn the traffic light to green (using the $turn$ action), and informs $LightC$ using the action $cycle$ to set the traffic light to green. The light controller will switch the traffic light to green, yellow and red, and subsequently informs the crossing controller that it has finished (by sending a $cycle$ message back).

Below a straightforward model of this system is provided. The sensor controller $SensorC$ gets a trigger via the action $trig$ and forwards it using $sens_s$. The traffic light controller is equally simple. After a trigger (via $cycle_r$), it cycles through the colours, and indicates through a $cycle_s$ message that it finished.

The crossing controller $CrossingC$ is a little more involved. It has four parameters. The first one is id which holds an identifier for this controller (i.e. 1 or 2). The second parameter my_turn indicates whether this controller has the right to set the traffic light to green. The third parameter is $sensor_triggered$ which stores whether a sensor trigger has arrived. The fourth one is $cycle$ indicating whether the traffic light controller is currently going through a traffic light cycle. The most critical actions are allowing the traffic

light to become green ($cycle_s$) and giving ‘my turn’ to the other crossing controller ($turn_s$). Both can only be done if no traffic light cycle is going on and it is ‘my turn’.

Note that at the *init* clause all components are put in parallel, and using the communication operator Γ and allow operator ∇ it is indicated how these components must communicate.

proc $SensorC(id:\mathbb{N}^+) = \underline{trig}(id) \cdot \underline{sens}_s(id) \cdot SensorC(id);$

$$\begin{aligned} &LightC(id:\mathbb{N}^+) \\ &= cycle_r(id) \cdot \\ &\quad \underline{show}(id, green) \cdot \\ &\quad \underline{show}(id, yellow) \cdot \\ &\quad \underline{show}(id, red) \cdot \\ &\quad cycle_s(id) \cdot \\ &\quad LightC(id); \end{aligned}$$

$$\begin{aligned} &CrossingC(id:\mathbb{N}^+, my_turn, sensor_triggered, cycle:\mathbb{B}) \\ &= \underline{sens}_r(id) \cdot CrossingC(id, my_turn, true, cycle) \\ &+ (\underline{sens}_r(id) \wedge my_turn \wedge \neg cycle) \rightarrow cycle_s(id) \cdot \\ &\quad CrossingC(id, my_turn, false, true) \\ &+ cycle_r(id) \cdot CrossingC(id, my_turn, sensor_triggered, false) \\ &+ \underline{turn}_r \cdot CrossingC(id, true, sensor_triggered, cycle) \\ &+ (\neg \underline{sens}_r(id) \wedge my_turn \wedge \neg cycle) \rightarrow \underline{turn}_s \cdot \\ &\quad CrossingC(id, false, sensor_triggered, cycle); \end{aligned}$$

init $\nabla_{\{trig, show, sens_c, cycle_c, turn_c\}}$
 $(\Gamma_{\{sens_r | sens_s \rightarrow sens_c, cycle_r | cycle_s \rightarrow cycle_c, turn_r | turn_s \rightarrow turn_c\}}$
 $(SensorC(1) || SensorC(2) ||$
 $CrossingC(1, true, false, false) || CrossingC(2, false, false, false) ||$
 $LightC(1) || LightC(2)));$

This straightforward system description has a state space of 160 states. We are interested in the behaviour of the system where *trig* and *show* are visible, and the other actions are hidden. We can do this by applying the hiding operator $\tau_{\{sens_c, cycle_c, turn_c\}}$ to the process. The system is confluent with respect to the hidden $cycle_c$ action. The hidden $sens_c$ and $turn_c$ actions are not contributing to the confluence of the system.

In the uppermost row of table 2 the sizes of the state space are given: of the full state space, after applying tau-prioritisation and after applying branching bisimulation reduction.

In order to employ the effect of confluence, we must make the hidden actions $turn_c$ and $sens_c$ confluent, too. The reason that these actions are not confluent is that handing over a turn and triggering a sensor are possible in the same state, and they can take place in any order. But the exact order in which they happen causes a different traffic light go to green.

We can prevent this by making the behaviour of the crossing controller *CrossingC* deterministic. A very simple way of doing this is given below. We only provide the definition of *SensorC* and *CrossingC* as *LightC* remains the same and the *init* line is almost identical. The idea of the specification below is that the controllers *CrossingC* are in charge of the sensor and light controllers. When the crossing controller has the turn, it polls the sensor. And only if it has been triggered, it initiates a traffic light cycle. In both cases it gives the turn to the other crossing controller.

proc $SensorC(id:\mathbb{N}^+) = sens_r(id) \cdot \sum_{b:\mathbb{B}} trig(id, b) \cdot sens_s(id, b) \cdot SensorC(id);$

$$\begin{aligned}
CrossingC(id:\mathbb{N}^+, my_turn:\mathbb{B}) = & \\
& my_turn \\
& \rightarrow sens_s(id) \cdot \\
& (sens_r(id, true) \cdot \\
& \quad cycle_s(id) \cdot \\
& \quad cycle_r(id) \\
& + \\
& \quad sens_r(id, false) \\
&) \cdot \\
& turn_s \cdot \\
& CrossingC(id, false) \\
& \diamond turn_r \cdot \\
& CrossingC(id, true);
\end{aligned}$$

The state space of this system turns out to be small, namely 20 states (see table 2, second row). It is even smaller after applying τ -prioritisation, namely 8 states. Remarkably, this is also the size of the state space after branching bisimulation minimisation. As the state space is small, it is possible to inspect the state space in full (see figure 11). An important property of this system is that the relative ordering in which the triggers at sensor 1 and sensor 2 are polled does not influence the ordering in which the traffic lights go to green. This sequence is only determined by the booleans that indicate whether the sensor is triggered or not. This effect is not very clear here, because the sensors are polled in strict alternation. But in the next example we see that this property also holds for more complex controllers, where the polling order is not strictly predetermined.

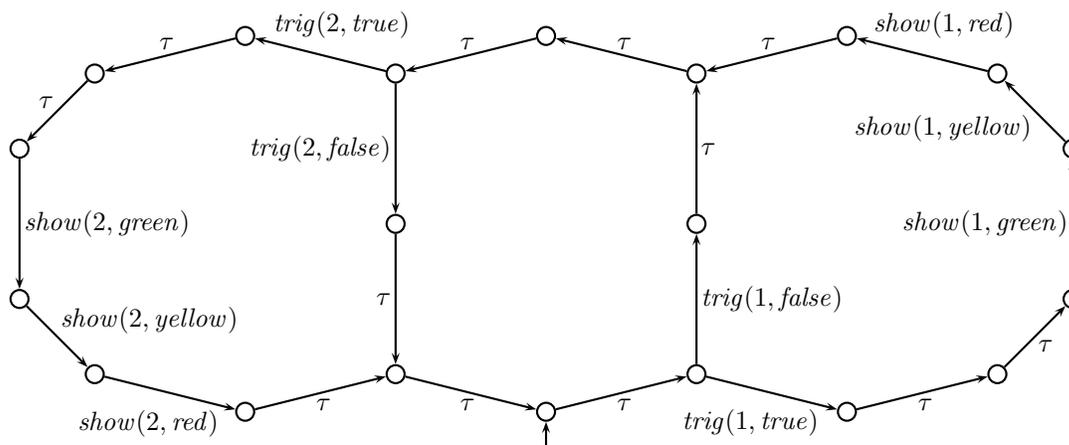


Figure 11: The state space of a simple confluent traffic light controller

The previous solution can be too simple for certain purposes. We show that the deterministic specification style can still be used for more complex systems, and that the state space that is generated using τ -prioritisation is still much smaller than state spaces generated without the use of confluence.

So, for the sake of the example we assume that it is desired to check the sensors while a traffic light cycle is in progress. Both crossing controllers continuously request the sensors to find out whether they have been triggered. If none is triggered the traffic light controllers inform each other that the turn does not have to switch side. If the crossing controller whose turn it is, gets the signal that its sensor has been triggered, it awaits the end of the current traffic light cycle ($cycle_r(id)$), and simply starts a new cycle ($cycle_s(id)$). If the sensor of the crossing controller that does not have the turn is triggered, this controller indicates using $turn_s(true)$ that it wants to get the turn. It receives the turn by $turn_r$. Subsequently, it starts its own traffic light cycle.

	no reduction	after τ -prioritisation	mod branch bis
Non-confluent controller	160	128	124
Simple confluent controller	20	8	8
Complex confluent controller	310	56	56

Table 2: The number of states of the transitions systems for a simple crossing

The structure of the system is the same as in the non-confluent traffic light cycle, and therefore the init part is not provided in the specification below.

proc $SensorC(id:\mathbb{N}^+) = sens_r(id) \cdot \sum_{b:\mathbb{B}} trig(id, b) \cdot sens_s(id, b) \cdot SensorC(id);$

$$\begin{aligned}
CrossingC(id:\mathbb{N}^+, my_turn:\mathbb{B}) = & \\
& sens_s(id) \cdot \\
& (sens_r(id, true) \cdot \\
& (my_turn \rightarrow cycle_r(id) \diamond turn_s(true) \cdot turn_r) \cdot \\
& cycle_s(id) \cdot \\
& CrossingC(id, true) \\
& + \\
& sens_r(id, false) \cdot \\
& (my_turn \\
& \rightarrow (turn_r(true) \cdot \\
& cycle_r(id) \cdot \\
& turn_s \cdot \\
& CrossingC(id, false) \\
& + \\
& turn_r(false) \cdot \\
& CrossingC(id, true) \\
&) \\
& \diamond turn_s(false) \cdot \\
& CrossingC(id, false) \\
&) \\
&);
\end{aligned}$$

$$\begin{aligned}
LightC(id:\mathbb{N}^+, active:\mathbb{B}) = & \\
& active \\
& \rightarrow cycle_s(id) \cdot LightC(id, false) \\
& \diamond cycle_r(id) \cdot show(id, green) \cdot show(id, yellow) \cdot show(id, red) \cdot LightC(id, true);
\end{aligned}$$

This more complex traffic light controller has a substantially larger state space of 310 states. However, when the state space is generated with τ -prioritisation, it has shrunk to 56 states, which is also its minimal size modulo branching bisimulation or even weak trace equivalence.

The complexity of the system is in the way the sensors are polled. Figure 12 depicts the behaviour where showing the aspects of the traffic lights is hidden. As in the simple confluent controller, the relative ordering of the incoming triggers does not matter for the state the system ends up in. E.g., executing sequences $trig(2, false) trig(1, true)$ and $trig(1, true) trig(2, false)$ from the initial state lead to the lowest state in the diagram. This holds in general. Any allowed reordering of the triggers from sensor 1 and 2 with respect to each other will bring one to the same state.

8 Guideline V: 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

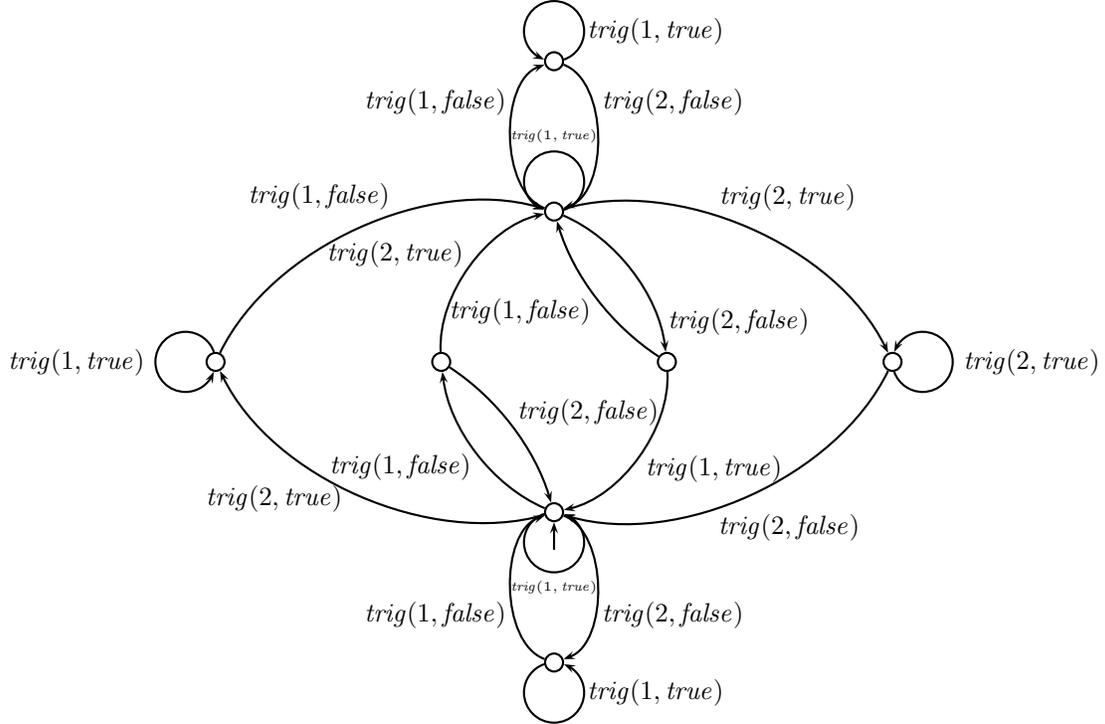


Figure 12: The sensor polling pattern of a more complex confluent controller

almost always is), then there are several methods to reduce its footprint. Below we give three examples, one where data is categorised, one where the content of queues is reduced 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 [7]. 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.

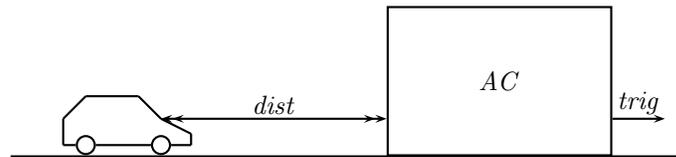


Figure 13: An advanced approach controller

Consider for example an intelligent approach controller which measures the distance of an approaching car as depicted in figure 13. 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.

```

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    $Distance = \mathbf{struct} \text{ near } | \text{ far};$ 
proc    $AAC = \sum_{d:Distance} \underline{dist}(d) \cdot ((d \approx \text{near}) \rightarrow \underline{trig} \cdot AAC \diamond AAC);$ 
init    $AAC;$ 

```

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

We now provide an example showing how to reduce the usage of buffers and queues. Polling and τ -confluence are used, to achieve the reduction. We model a system with autonomous traffic light controllers. Each controller has one sensor and controls one traffic light that can be red or green. If a sensor is triggered, the traffic light must show green. At most one traffic light can show green at any time. The controllers are organised in a ring, where each controller can send messages to its right neighbour, and receive messages from its left neighbour. For reasons of efficiency we desire that there are unbounded queues between the controllers, such that no controller is ever hampered in forwarding messages to its neighbour. The situation is depicted in figure 14.

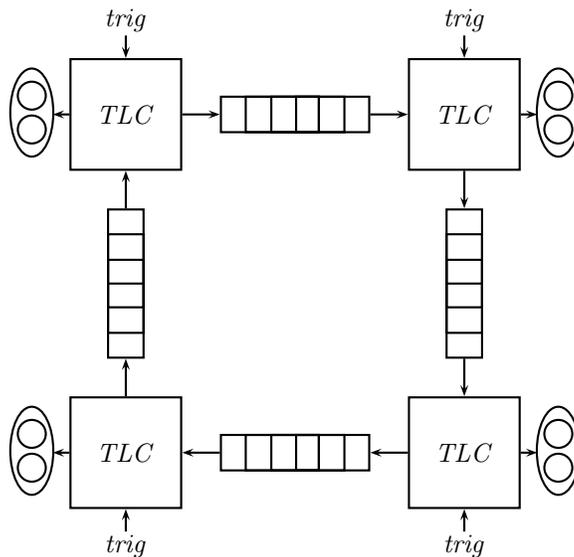


Figure 14: Process communication via unbounded queues

We make a straightforward protocol, where we do not look into efficiency. Whenever a traffic light controller receives a trigger, it wants to know from the other controllers that they are not showing green. For this reason it sends its sequence number with an ‘active’ tag around. If it makes a full round without altering the ‘active’ tag, it switches its own traffic light to green. Otherwise, if the tag is switched to ‘passive’, it retries sending the message around. A formal description is given by the following specification. The process $Queue(id, q)$ describes an infinite queue between the processes with identifiers id and $id+1$ (modulo the number of processes). The parameter q contains the content of the queue. The process $TLC(id, triggered, started)$ is the process with id id where $triggered$ indicates that it has been triggered

N	Non confluent	After branching bis	Confluent	With τ -prioritisation	After branching bis
2	116	58	10	6	6
3	$3.2 \cdot 10^3$	434	15	9	9
4	$122 \cdot 10^3$	$3 \cdot 10^3$	20	12	12
5	$5.9 \cdot 10^6$	$21 \cdot 10^3$	25	15	15
6	$357 \cdot 10^6$	-	30	18	18
20	-	-	100	60	60

Table 3: Traffic lights connected with queues

to show green, and *started* indicates that it has started with the protocol sketched above. In the initialisation we describe the situation where there are two processes and two queues, but the protocol is suited for any number of processes and an equal number of queues.

```

sort   Aspect = struct green | red;
         Message = struct active(get_number : N)?is_active | passive(get_number : N);
map    N :  $\mathbb{N}^+$ ;
eqn    N = 2;
proc   Queue(id:N, q>List(Message)) =
          $\sum_{m:Message} q_{in_r}(id, m) \cdot Queue(id, m \triangleright q) +$ 
          $(\#q > 0) \rightarrow q_{out_s}((id+1) \bmod N, rhead(q)) \cdot Queue(id, rtail(q));$ 

         TLC(id:N, triggered, started: $\mathbb{B}$ ) =
          $trig(id) \cdot TLC(id, true, started) +$ 
          $(triggered \wedge \neg started)$ 
          $\rightarrow q_{in_s}(id, active(id)) \cdot TLC(id, false, true) +$ 
          $\sum_{m:Message} q_{out_r}(id, m) \cdot$ 
          $((started \wedge is\_active(m) \wedge get\_number(m) \neq id)$ 
          $\rightarrow q_{in_s}(id, passive(get\_number(m))) \cdot TLC(id, triggered, started)$ 
          $\diamond ((started \wedge get\_number(m) \approx id)$ 
          $\rightarrow (is\_active(m) \rightarrow show(id, green) \cdot show(id, red) \cdot TLC(id, triggered, false)$ 
          $\diamond TLC(id, true, false)$ 
          $))$ 
          $\diamond q_{in_s}(id, m) \cdot TLC(id, triggered, started)$ 
          $));$ 

init    $\tau_{\{q_{in_c}, q_{out_c}\}}(\nabla_{\{trig, show, q_{in_c}, q_{out_c}\}}(\Gamma_{\{q_{in_r} | q_{in_s} \rightarrow q_{in_c}, q_{out_r} | q_{out_s} \rightarrow q_{out_c}\}}($ 
          $TLC(0, false, false) || TLC(1, false, false) || Queue(0, []) || Queue(1, []))));$ 

```

Note that the state space of this system is growing very dramatically with the number of processes. See the second column in table 3. In the third column the state space is given after a branching bisimulation reduction, where only the actions *show* and *trig* are visible. Even the state space after branching bisimulation reduction is quite large. A dash indicates that the mCRL2 toolset failed to calculate the state space or the reduction thereof (running out of space on a 1Tbyte main memory linux machine).

We will reduce the number of states by making the system confluent. We replace data pushing by polling. The structure of the protocol becomes quite different. Each process must first obtain a mutually exclusive ‘token’, then polls whether a trigger has arrived, and if so, switches the traffic light to green. Subsequently, it hands the token over to the next process. The specification is given below for two processes. The specification of the queue is omitted, as it is exactly the same as the one of the previous specification.

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

Table 4: Number of states of an non ordered/ordered buffer with max. N elements

```

sort   Aspect = struct green | red;
         Message = struct token;
map    N :  $\mathbb{N}^+$ ;
eqn    N = 2;

proc   TLC(id: $\mathbb{N}$ , active: $\mathbb{B}$ ) =
         active  $\rightarrow$  (trig(id, true) · show(id, green) · show(id, red) + trig(id, false)) ·
         qin_s(id, token) · TLC(id, false)
          $\diamond$  qout_r(id, token) · TLC(id, true);

init    $\tau_{\{q_{in_c}, q_{out_c}\}}(\nabla_{\{\text{trig}, \text{show}, q_{in_c}, q_{out_c}\}}(\Gamma_{\{q_{in_r} | q_{in_s} \rightarrow q_{in_c}, q_{out_r} | q_{out_s} \rightarrow q_{out_c}\}}(
         TLC(0, true) || TLC(1, false) || Queue(0, []) || Queue(1, []))))$ ;

```

The number of states of the state space for different number of processes are given in the fourth column of table 3. In the fifth and sixth columns the number of states after τ -prioritisation and branching bisimulation reduction are given. Note that the number of states after τ -prioritisation is equal to the number of states after application of branching bisimulation. Note also that the differences in the sizes of the state spaces is quite striking.

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.

```

map    N :  $\mathbb{N}$ ;
         insert, ordered_insert :  $\mathbb{N} \times List(\mathbb{N}) \rightarrow List(\mathbb{N})$ ;

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

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 4 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.

9 Guideline VI: Compositional design and reduction

When a system that must be designed consists of several components, it can be wise to organise these components in such a way that stepwise composition and reduction are possible. The idea is depicted in figure 15. At the left hand side of figure 15 a set of communicating components C_1, \dots, C_5 is depicted. In the middle, the interfaces I_1, \dots, I_7 are also shown. At the right the system has a tree structure.

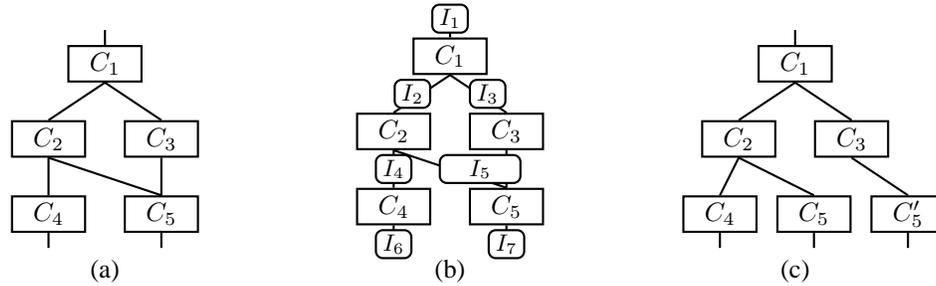


Figure 15: The compositional design and verification steps

When calculating the behaviour of the whole system, a characterisation of the simultaneous behaviour at the interfaces I_1, I_6 and I_7 is required where all communication at the other interfaces is hidden. Unfortunately, calculating the whole behaviour before hiding internal communication may not work, because the whole behaviour has too many states. An alternative is to combine and hide in an alternating fashion. After each hiding step a behavioural reduction is applied, which results in a reduced transition system.

For instance, the interface behaviour at I_2, I_5 and I_6 can be calculated from the behaviour of C_2 and C_4 by hiding the behaviour at I_4 . Subsequently, C_3 and C_5 can be added, after which the communication at I_5 can be hidden. At last adding C_1 and hiding the actions at the interfaces I_2 and I_3 finishes the calculation of the behaviour. This alternation of composing behaviour and hiding actions is quite commonly known and some toolsets even developed a script language to allow for an optimal stepwise composition of the whole state space [10].

In order to optimally employ this stepwise sequence of composition, hiding and reduction, it is desired that as much communication as possible can be hidden to allow for a maximal reduction of behaviour. But there is something even more important. If a subset of components has more interfaces that will be closed off by adding more components later, it is very likely that there is some relationship between the interactions at these interfaces. As long as the set of components has not been closed, the interactions at these interfaces are unrelated, often leading to a severe growth in the state space of the behaviour of this set of sub-components. When closing the dependent interfaces, the state space is brought to its expected size. If such dependent but unrestricted interfaces occur, the use of stepwise composition and reduction is generally ineffective.

As an example consider figure 15 again. If C_2, C_3, C_4 and C_5 have been composed, the system has interactions at interfaces I_2 and I_3 that can happen independently. Adding C_1 restricts the behaviour at these interfaces. For instance, C_1 can strictly alternate between sending data via I_2 and I_3 , but without C_1 any conceivable order must be present in the behaviour of C_2, C_3, C_4 and C_5 .

Dependent but unrestricted interfaces can be avoided by using a tree topology. See figure 15 (c) where the dependency at interfaces I_2 and I_3 has been removed by duplicating component C_5 . If a tree topology is not possible, then it is advisable to restrict behaviour at dependent but unrestricted interfaces as much as possible from inside sets of components.

As an example we provide yet another distributed traffic controller (see figure 16). There are a certain number N of traffic lights. At the central component (the *TopController*) requests arrive using a *set(m)* action to switch traffic light m to green. This request is forwarded via intermediate components (called

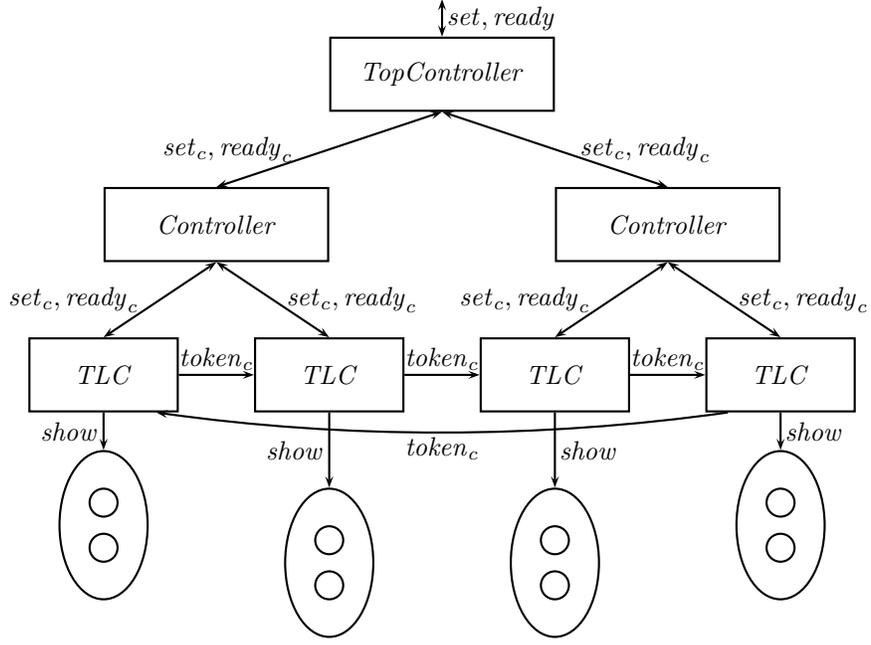


Figure 16: Distribution of system components

Controllers) to traffic light controllers (*TLCs*). If a traffic light has been set to green and subsequently to red again, an action $ready(n)$ indicates that the task has been accomplished. The system must guarantee that one traffic light can be green at any time but the order in which this happens is not prescribed.

We start presenting a solution that does not have a tree topology. Using the principle of separation of concerns, we let the traffic light controllers be responsible for taking care that no two traffic lights are showing green at the same time. The top- and other controllers have as task to inform the traffic light controllers that they must set the light to green, and they transport the ready messages back to the central controller.

The traffic light controllers use a simple protocol as described in the queue example in section 8. They continuously exchange a token. The owner of the token is allowed to set the traffic light to green. The parameter id is the identifier of the traffic light. The parameter $level$ indicates the level of the traffic light controllers. The top controller has level 0. In figure 16 the level of the traffic light controllers is 2. Furthermore, has_token indicates that this traffic light controller owns the token, and $busy$ indicates that it must let the traffic light go through a green-red cycle.

The controllers and the top controller are more straightforward. They pass set commands from top to bottom, and send ready signals from bottom to top. The parameters id_{low} and id_{high} indicate the range of traffic lights over which this controller has control. The description below describes a system with four traffic light controllers.

sort $Aspect = \mathbf{struct} \ green \ | \ red;$

proc $ControllerTop(id_{low}, id_{high}:\mathbb{N}) =$
 $\sum_{n:\mathbb{N}}(id_{low} \leq n \wedge n \leq id_{high}) \rightarrow (set(n) \cdot set_s(n, 1) + ready_r(n, 1) \cdot \underline{ready}(n)) \cdot$
 $ControllerTop(id_{low}, id_{high});$

$Controller(id_{low}, id_{high}, level:\mathbb{N}) =$
 $\sum_{n:\mathbb{N}}(id_{low} \leq n \wedge n \leq id_{high}) \rightarrow$
 $(set_r(n, level) \cdot set_s(n, level+1) \cdot Controller(id_{low}, id_{high}, level) +$
 $ready_r(n, level+1) \cdot ready_s(n, level)) \cdot Controller(id_{low}, id_{high}, level);$

	bottom control		bottom and top control		top control	
	4 nodes	8 nodes	4 nodes	8 nodes	4 nodes	8 nodes
Total system	10.0 10 ³	236 10 ⁶	1.09 10 ³	96.3 10 ³	368	15.6 10 ³
Mod branch. bis.	3.84 10 ³	39.8 10 ⁶	236	7.42 10 ³	236	7.42 10 ³
Without top controller	1.80 10 ³	25.3 10 ⁶	1.80 10 ³	25.3 10 ⁶	-	-
Mod branch. bis.	983	5.9 10 ⁶	983	5.9 10 ⁶	-	-
Half system	131	93.9 10 ³	131	93.9 10 ³	56	16.8 10 ³
Mod branch. bis.	107	44.1 10 ³	107	44.1 10 ³	33	3.06 10 ³

Table 5: State space sizes for a hierarchical traffic light controller

$$\begin{aligned}
TLC(id, level:\mathbb{N}, has_token, busy:\mathbb{B}) = & \\
& set_r(id, level) \cdot TLC(id, level, has_token, true) + \\
& (has_token \wedge busy) \rightarrow \underline{show}(id, green) \cdot \underline{show}(id, red) \cdot ready_s(id, level) \cdot \\
& \quad TLC(id, level, has_token, false) + \\
& (has_token \wedge \neg busy) \rightarrow token_s((id+1) \bmod 4) \cdot TLC(id, level, false, busy) + \\
& (\neg has_token) \rightarrow token_r(id) \cdot TLC(id, level, true, busy);
\end{aligned}$$

$$\begin{aligned}
\mathbf{init} \quad \nabla_{\{set_c, ready_c, token_c, \underline{show}, set, ready\}} (\Gamma_{\{set_r | set_s \rightarrow set_c, ready_r | ready_s \rightarrow ready_c, token_r | token_s \rightarrow token_c\}} (\\
& ControllerTop(0, 3) || Controller(0, 1, 1) || Controller(2, 3, 1) || \\
& TLC(0, 2, true, false) || TLC(1, 2, false, false) || \\
& TLC(2, 2, false, false) || TLC(3, 2, false, false));
\end{aligned}$$

In order to understand the state space of components and sets of sub-components, we look at the size of the whole state space, the size of the state space without the top controller, and the size of half the system with one controller and two TLCs. The results are listed in table 5 for a system with four and eight traffic light controllers. In case of four traffic lights, a half system has two traffic lights and one controller. In case of eight traffic lights, a half system has four traffic lights and three controllers. The results of the sizes of the state spaces are given in the columns under the header ‘bottom control’. In all cases the size of the state space modulo branching bisimulation is also given. Here all internal actions are hidden and the external actions *show*, *set* and *ready* are visible.

What we note is that the sizes of the state spaces are large. In particular the size of the state space modulo branching bisimulation of the system without the top controller multiplied with the size of the top controller is almost as large as the size of the total state space. The state space of the top controller for four traffic lights has 9 states and the one for eight traffic lights has 17 states. It makes little sense to use compositional verification in this case, but the fact that the top controller hardly restricts the behaviour of the rest of the system saves the day. If the top controller is more restrictive compositional verification makes no sense at all.

If we analyse the large state space of this system, we see that the independent behaviour of the controllers substantially adds to the size of the state space. We can restrict this by giving more control to the top controller. Whenever it receives a request to *set* a traffic light to green, it stores it in a set called *requests*. Whenever a traffic light is allowed to go to green, indicated by *busy* equals false, the top controller non-deterministically selects an index of a traffic light from *requests* and instruct it to go to green. The specification of the new top controller is given below.

$$\mathbf{proc} \quad ControllerTop(id_{low}, id_{high}:\mathbb{N}) = ControllerTop(id_{low}, id_{high}, \emptyset, false);$$

$$\begin{aligned}
ControllerTop(id_{low}, id_{high}:\mathbb{N}, requests:Set(\mathbb{N}), busy:Bool) = & \\
& \sum_{n:\mathbb{N}} (id_{low} \leq n \wedge n \leq id_{high} \wedge n \notin requests) \rightarrow \\
& \quad \underline{set}(n) \cdot ControllerTop(id_{low}, id_{high}, requests \cup \{n\}, busy) + \\
& \sum_{n:\mathbb{N}} (id_{low} \leq n \wedge n \leq id_{high} \wedge n \in requests \wedge \neg busy) \rightarrow \\
& \quad \underline{set}_s(n, 1) \cdot ControllerTop(id_{low}, id_{high}, requests \setminus \{n\}, true) + \\
& \sum_{n:\mathbb{N}} (id_{low} \leq n \wedge n \leq id_{high} \wedge n \in requests) \rightarrow \\
& \quad \underline{ready}_r(n, 1) \cdot \underline{ready}(n) \cdot ControllerTop(id_{low}, id_{high}, requests, false);
\end{aligned}$$

The resulting state spaces are given in table 5 under the header ‘bottom and top control’. The first observation is that the sizes of the state spaces without top control and of a half system have not changed. This is self evident, as only the top controller has been replaced. It is important to note that the sizes of the state space modulo branching bisimulation of the the system without top controller is almost as large as the unreduced state space of the full system for four traffic lights. For eight traffic lights the intermediate reduced state space is much larger than the unreduced system of the full state space.

We can remove the low level control via the exchange of the token. This is possible because the top controller now guarantees that at most one traffic light shows green. This is done by replacing the specification of the traffic light controller by the simple specification below. Note that the communication topology of the system now has a tree structure.

```
proc    $TLC(id, level:\mathbb{N}) =$   
         $set_r(id, level) \cdot \underline{show}(id, green) \cdot \underline{show}(id, red) \cdot ready_s(id, level) \cdot TLC(id, level);$ 
```

We are not interested anymore in the behaviour of the system with all the traffic light controllers and no top controller. We only need to look at the sizes of the half systems which can be reduced and both half systems can directly be combined with the top controller. Note that in this way we circumvent the blow-up of intermediate processes. Note also that the resulting state spaces modulo branching bisimulation for the system with ‘top control’ are the same as those for ‘bottom and top control’. This shows that the token exchange is really immaterial when the top controller guarantees that at most one traffic light goes to green. Finally, note that the half systems with bottom control are only slightly bigger than the half systems with top control. From this we can conclude that token exchange by itself does not contribute substantially to the size of the state space.

10 Guideline VII: Specify external behaviour of sets of sub-components

In the previous section we mentioned that stepwise composition and reduction might be a way to avoid a blow-up of the state space. But we observed that sometimes the composed behaviour of sets of components is 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 [21], 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 [6]. 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 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 [20] 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 17. 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

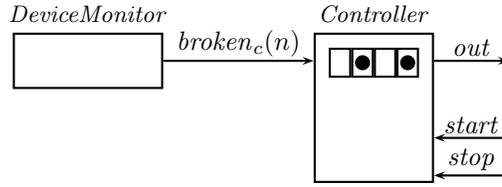


Figure 17: A system comprises a controller and a device-monitor

buffer by a function from natural numbers to booleans. If $buff(i)$ is true, it indicates that a fault report has been received for device i . The boolean parameter b 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    $buff_0:\mathbb{N}\rightarrow\mathbb{B}$ ;
eqn    $buff_0 = \lambda n:\mathbb{N}.false$ ;
proc    $DeviceMonitor = \sum_{n:\mathbb{N}}(n < M) \rightarrow broken_s(n).DeviceMonitor$ ;
         $Controller(buff:\mathbb{N}\rightarrow\mathbb{B}, b:\mathbb{B}, i:\mathbb{N})$ 
          =  $\sum_{n:\mathbb{N}} broken_r(n).Controller(buff[n \rightarrow true], b, i)$ 
          +  $(\neg buff(i) \wedge b) \rightarrow stop.Controller(buff, false, i)$ 
          +  $(\neg b) \rightarrow start.Controller(buff, true, i)$ 
          +  $(buff(i) \wedge b) \rightarrow out.Controller(buff[i \rightarrow false], b, (i+1) \bmod M)$ 
          +  $(\neg buff(i) \wedge b) \rightarrow int.Controller(buff, b, (i+1) \bmod M)$ 
init    $\tau_{\{broken_c, int\}}(\nabla_{\{broken_c, out, start, stop, int\}}(\Gamma_{\{broken_r | broken_s \rightarrow broken_c\}}(Controller(buff_0, false, 0) || DeviceMonitor)))$ ;

```

The total number of devices is denoted by M . All positions of $buff$ are initially set to $false$ as indicated by the lambda expression $\lambda n:\mathbb{N}.false$. 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, we obtain the state space depicted in figure 18. 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.

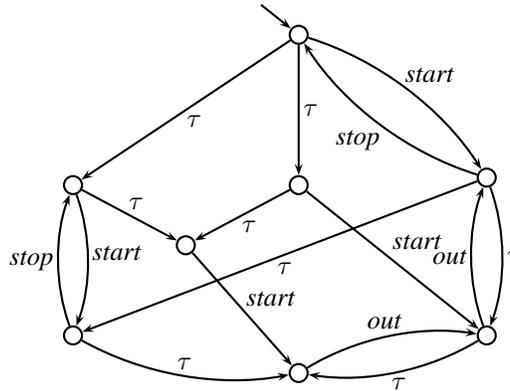


Figure 18: The system external behaviour (first specification)

Table 6 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

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

Table 6: Sizes of the original and external state space of the monitor controller (first specification)

behaviour are comparable. This indicates a complex external behaviour that might complicate verification with external parties and makes understanding the behaviour quite difficult.

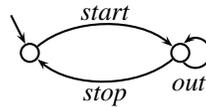


Figure 19: The system external behaviour (second specification)

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 19.

```

proc   Stopped = start.Started;
         Started = out.Started + stop.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 below. In this specification the *stop* request is processed independently from the rest of the behaviour.

M	No. of original states	No. of external states
1	4	2
2	16	2
3	48	2
4	128	2
5	320	2
6	768	2
10	$20.5 \cdot 10^3$	2

Table 7: Sizes of the original and external state space of the monitor controller (second specification)

```

map    $M:\mathbb{N}^+$ ;
eqn    $M=2$ ;
map    $buff:\mathbb{N}\rightarrow\mathbb{B}$ ;
eqn    $buff = \lambda n:\mathbb{N}.false$ ;
proc    $DeviceMonitor = \sum_{n:\mathbb{N}}(n < M) \rightarrow broken_s(n).DeviceMonitor$ ;
         $Controller(buff:\mathbb{N}\rightarrow\mathbb{B}, b:\mathbb{B}, i:\mathbb{N})$ 
        =  $\sum_{n:\mathbb{N}} broken_r(n) \cdot Controller(buff[n \rightarrow true], b, i)$ 
        +  $b \rightarrow stop \cdot Controller(buff, false, i)$ 
        +  $(\neg b) \rightarrow start \cdot Controller(buff, true, i)$ 
        +  $(buff(i) \wedge b) \rightarrow out \cdot Controller(buff[i \rightarrow false], b, (i+1) \bmod M)$ 
        +  $(\neg buff(i) \wedge b) \rightarrow int \cdot Controller(buff, b, (i+1) \bmod M)$ 
init    $\tau_{\{broken_c, int\}}(\nabla_{\{broken_c, out, start, stop, int\}}(\Gamma_{\{broken_r | broken_s \rightarrow broken_c\}}($ 
         $Controller(buff, false, 0) || DeviceMonitor))$ );

```

As can be seen from table 7, the number of states of the non-reduced model remains the same. However, the reduced behaviour is exactly the one depicted in figure 19 for any constant M . This means that it has only two states. This specification is much more usable for stepwise composition and reduction than the first one we provided.

11 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 seven guidelines, based on our experience with building models of various systems. There is no claim that this set is complete, or even that these seven 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.

References

- [1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. Proceedings of the 1997 ACM SIGMOD international conference on Management of data, pp. 183-194. 1997.
- [2] F. Arbab. Reo: A Channel-based coordination model for component composition. Mathematical Structures in Computer Science, Cambridge University Press, 14(3):329-366, 2004.
- [3] J.C.M. Baeten and W.P. Weijland. Process algebra. Cambridge Tracts in Theoretical Computer Science 18, 1990.
- [4] G. Berry, G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming, 19:87-152, 1992.

- [5] S.C.C. Blom and J.C. van de Pol. State space reduction by proving confluence. In E. Brinksma and K.G. Larsen, editors, Proceedings of 14th Int. Conf. on Computer Aided Verification (CAV'02). Lecture Notes in Computer Science 2404, pp. 596-609, Springer Verlag, 2002.
- [6] G.H. Broadfoot. ASD case notes: costs and benefits of applying formal methods to industrial control software. In proceedings of formal methods conference (FM 2005). LNCS 3582, pp. 548-551. Springer Verlag, 2005.
- [7] D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems (TOPLAS) 19(2):253-291, 1997.
- [8] Formal Methods for Industrial Critical Systems. Conference Proceedings. 1996–2010.
- [9] Formal Methods in System Design. Journal. Springer Verlag, 1992–2010.
- [10] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A toolbox for the onstruction and analysis of distributed processes. Proceedings of the 19th International Conference on Computer Aided Verification (CAV'2007, Berlin, Germany). Volume 4590 of Lecture Notes in Computer Science, pp. 158-163. Springer Verlag, 2007.
- [11] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. Journal of the ACM 43(3):555-600, 1996.
- [12] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander, W. Gardner, editors, Process Algebra for Parallel and Distributed Processing. Chapman Hall, pp. 99-128, 2009.
- [13] J.F. Groote and M.A. Reniers. Modelling and analysis of communicating systems. To appear 2011.
- [14] J.F. Groote and M.P.A. Sellink. Confluence for process verification. Theoretical Computer Science. 170(1-2):47–81, 1996.
- [15] G.J. Holzmann. The SPIN model checker. Primer and reference manual. Addison-Wesley, 2003.
- [16] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. Int. Journal on Software Tools for Technology Transfer, 1(12):134-152, October 1997.
- [17] F.J. Lin, P.M. Chu, and M.T. Liu. Protocol verification using reachability analysis: The state space explosion problem and relief strategies. ACM SIGCOMM Computer Communication Review. 17(5):126–135, 1987.
- [18] R. Milner. A Calculus of communicating systems. Lecture Notes in Computer Science 92, Springer Verlag, 1980.
- [19] A. Osaiweran, M. Boosten, and M.R. Mousavi. Analytical software design: Introduction and industrial experience report. Eindhoven University of Technology. Technical report CSR-10–01, 2010.
- [20] A.S. Tanenbaum. Computer networks. Second edition. Prentice Hall, 1988.
- [21] C.A. Vissers, and L. Logrippo. The importance of the service concept in the design of data communications protocols. In M. Diaz, editor, Protocol Specification, Testing and Verification (proc. of the IFIP WG 6.1 Fifth International Workshop on Protocol Sepcification, Testing and Verification), Elsevier North Holland, pp. 3-17, 1986.
- [22] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification styles in distributed systems design and verification. Theoretical Computer Science 89:179-206, 1991.
- [23] J.M. Voas. K.W. Miller. Software testability: the new verification. IEEE Software 12(3):17-28, 1995.
- [24] L.-T. Wang, C.-W. Wu and X. Wen. Design for testability. VLSI test principles and architectures. Morgan Kaufmann Publishers. 2006.
- [25] www.mcr12.org. 2010.