Specification guidelines to avoid the state space explosion problem

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2015; **25**:4–33
Published online 12 May 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1536

Specification guidelines to avoid the state space explosion problem

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

Department of Mathematics and Computer Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands

SUMMARY

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 systems. 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 summarize our experience by providing seven specification guidelines to keep state spaces small. For each guideline, we provide an application, generally 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. The good and bad models are both suitable for their purpose but are *not* behaviourally equivalent. For all guidelines, we discuss circumstances under which it is reasonable to apply the guidelines. Copyright © 2014 John Wiley & Sons, Ltd.

Received 9 July 2012; Revised 13 March 2014; Accepted 7 April 2014

KEY WORDS: design for verifications; specification guidelines; state space explosion; model checking

1. INTRODUCTION

Behavioural specification of computer systems, distributed algorithms, communication protocols, business processes, and so on 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 [1, 2]. 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.

*Correspondence to: Jan Friso Groote, Department of Computer Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands.
†E-mail: J.F.Groote@tue.nl

Behavioural verification usually employs the generation of a state/transition diagram, generally called the state space in one or another form. A problem of these state spaces is that they often become too large to be verified even when using clever verification algorithms and very powerful computers. This problem is called the state space explosion problem. It receives lots of attention in the literature, generally from the perspective of semantics-preserving and property-preserving reductions, or smart compression techniques. Although these techniques are absolutely powerful and often very ingenious, they do 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 with 'design for testability', which is mainly used in especially microelectronics to allow to test a product for production flaws [3], and which is slowly finding its way into software engineering [4].

What we propose is that systems are designed such that the state spaces of their behavioural models are small. For instance, if the situation allows it, it is generally a good idea to avoid parallel behaviour in favour of the sequential execution of various tasks because the state space of the latter is often much smaller. But note that these two behaviours can be very different from a behavioural perspective. They are in general certainly not bisimilar or trace equivalent. But they may both serve the intended purpose equally well. This could for instance be formalized by saying that certain modal formulas must be valid. In this article, we leave the adequacy of the models to the judgement of the designer of the behaviour and primarily make the point that different styles of modelling can make a huge difference in the number of states of a system, and therefore its analysability. A designer of behaviour generally has lots of freedom to shape his artefact while meeting all desires and constraints on the design. We only state that keeping the state space as small as possible is one of the desires he should have, to allow formal analysis of his design, which is always beneficial.

Compared with the development of state space reduction techniques that preserve a given equivalence, design for verifiability is a rather novel concept because the validity of the reduction is in the hand of the designer. The best we could find is [5], but it primarily addresses improvements in verification technology, too. A recent plea similar to ours is [6], where it is argued that systems must be programmed such that analysis of those systems becomes much simpler. Specification styles from the perspective of expressiveness have been addressed [7], but verifiability is also not really an issue here. There are many papers treating state space reductions of systems preserving a given equivalence, for example, [8, 9], but the reductions obtainable in that way appear far less effective than those obtainable by the primary designer of the behaviour of a system.

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 taken from the domain of traffic light controllers (TLCs), except for guidelines II, which uses simple message passing, and guideline VII, where an example to monitor devices is used.

For each guideline, we give pairs of 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 [10, 11]. However, none of the guidelines are very specific for process algebras and can be applied quite generally, provided the relevant language concepts exist. For instance, for 'global synchronous communication', it is required that multiparty communication is available.

The guidelines are the result of our experience of designing and verifying a number of real industrial cases, developed under the control of various formal techniques [12–16]. Moreover, the guidelines were applied to design and verify an industrial case developed at Philips Healthcare [12], and hence, the guidelines could provide an effective framework to design verifiable components. What we see is that formal verification techniques assist in delivering systems of high quality, but

DOI: 10.1002/stvr

developers frequently run into the pitfalls caused by the state space explosion problem, and we hope that the guidelines in this article will help to avoid these.

This paper extends the guidelines of [17] with two key guidelines, namely ‘confluence and determinacy’ and ‘compositional design and reduction’. Additionally, the other guidelines are more extensively presented, supported by a number of examples and design cases. This paper also includes a concise introduction to the mCRL2 language that we use in this work.

In hindsight, we can say that it is quite self evident why most 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, when 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 designers 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 behavioural designers.

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

2. 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 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 [10, 11]. Especially, at the website `www.mcrl2.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 [18], 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. For example, 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 to avoid confusion with $=$, which we use as equality between processes. We also use $\mathit{if}(c,t,u)$ representing the term t if the condition c holds, and u if c is not valid.

For any sort D, the sorts $\mathit{List}(D)$ and $\mathit{Set}(D)$ contain the lists and sets over domain D. Prepending an element d to a list l is denoted by $d \triangleright l$. Getting the last element of a list is denoted as $\mathit{rhead}(l)$. The remainder of the list after removing the last element is denoted as $\mathit{rtail}(l)$. The length of a list is denoted by $\#(l)$. Testing whether an element d 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 $D_1 \to D_2$ contains all functions from the elements from D_1 to elements of D_2. We use standard lambda notation to represent functions. For example, $\lambda x{:}\mathbb{N}.x+1$ is the function that adds 1 to its argument. For a function f, we use the notation $f[t \to u]$ to represent the function f, except that if $f[t \to u]$ is applied to t, the value u is returned. We call $f[t \to u]$ 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.

DOI: 10.1002/stvr

sort $\quad Aspect = \textbf{struct}\ green \mid yellow \mid red;$

A more complex user-defined sort that we use is a message containing a number that can either be active or passive, for example, typical messages are $active(6)$ and $passive(234)$. 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. Such a datatype can be defined in the following way.

sort $\quad Message = \textbf{struct}\ active(get\_number{:}\mathbb{N})?is\_active \mid passive(get\_number{:}\mathbb{N});$

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

$$
\begin{array}{ll}
\textbf{map} & n : \mathbb{N}; \\
 & f : \mathbb{N} \to \mathbb{B}; \\
\textbf{eqn} & n = 3; \\
 & f = \lambda\, x{:}\mathbb{N}.false;
\end{array}
$$

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 $1.8\ 10^{19}$ states, which for quite some time to come will not fit into the memory of any computer (unless compression techniques are used). This emphasizes the importance to restrict the possible values data types can have. Often, it is wise to model data domains in abstract categories. For example, instead of using a height in millimetres, one can abstract this to the three values *low*, *middle* and *high*.

The behaviour of systems is characterized 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 in the succeeding text, where the types indicate the sorts of the data arguments that they carry. Note that *my_turn* has no arguments.

$$
\begin{array}{ll}
\textbf{act} & trig : \mathbb{N} \times \mathbb{B}; \\
 & send : Message; \\
 & my\_turn;
\end{array}
$$

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 carried out 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, $cond \to p \diamond q$, allows the condition *cond* to determine whether the process p or q is selected. The else part can always be omitted. We then obtain the conditional

DOI: 10.1002/stvr

operator of the form *cond* $\rightarrow$ p. If *cond* is not valid, this process cannot perform an action. It deadlocks. This does not need to be a problem as 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), that is, the counter starting with argument 0. Initial states are declared using the keyword **init**. As explained in the succeeding text, we underline actions, if they are not involved in communication between processes.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Counter}(n{:}\mathbb{N}) \\
& \quad = (n{<}4) \rightarrow \underline{\mathit{tick}}{\cdot}\mathit{Counter}(n{+}1) \diamond \underline{\mathit{tick}}{\cdot}\mathit{Counter}(0) \\
& \quad + \underline{\mathit{reset}}{\cdot}\mathit{Counter}(0); \\
\textbf{init} & \mathit{Counter}(0);
\end{array}
$$

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 [19]. We will not go into this precise mapping, because it is quite straightforward. The transition systems referred to in this article are all generated using the mCRL2 toolset [11].

Sometimes, it is required to allow a choice in behaviour, depending on the data. For example, 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

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

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 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 aforementioned example uses such a restriction and becomes

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

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

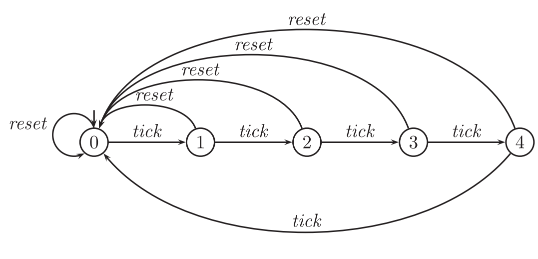

Figure 1. The transition system of the process *Counter*.

DOI: 10.1002/stvr

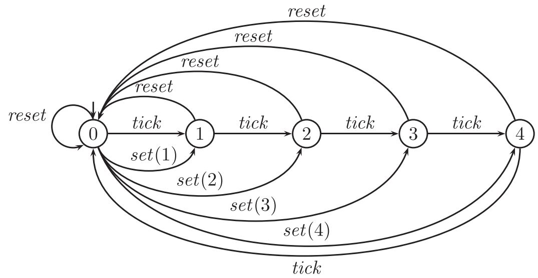

Figure 2. The *Counter* extended with *set* transitions.

typical sequential process (sequential in the meaning of not parallel). In Figure 2, we provide the state space of the extended counter.

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

Processes can be put in parallel with the parallel operator $\|$ 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 the 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 parallel 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, which is 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.

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 $p \parallel q$. 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

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

We provide a simple example of two parallel processes X_1 and X_2 of which the first reads a number, forwards it to the second and the second accepts it and sends it to the output. Both processes are single place buffers, and their parallel combination constitutes a two place buffer. There are five actions, two for the input and output of the first and second process (*input*, *output*), and three to

DOI: 10.1002/stvr

arrange handing data over between the processes (ho_c, ho_s and ho_r). The actions ho_s and ho_r represent sending a number and reading a number between the two processes. By using a communication operator $\Gamma_{\{ho_s|ho_r \to ho_c\}}$, it is expressed that actions ho_s and ho_r can synchronize to ho_c. By using the allow operator $\nabla_{\{ho_c\}}$, it is expressed that only action ho_c can happen, forcing ho_s and ho_r to communicate into ho_c. The whole system is described by the following three equations:

proc $X_1 = \sum_{n:\mathbb{N}} \underline{input}(n) \cdot ho_s(n) \cdot X_1;$
$X_2 = \sum_{m:\mathbb{N}} ho_r(m) \cdot \overline{output}(m) \cdot X_2;$
init $\nabla_{\{ho_c\}}(\Gamma_{\{ho_s|ho_r \to ho_c\}}(X_1 \parallel X_2));$

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, which is carried out by simply replacing each action in the set I of actions by the internal action τ. So, for a process that consists of a single action a, $\tau_{\{a\}}(a)$ is the internal action τ, an action that does happen, but which cannot directly be observed. In the aforementioned example, hiding ho_c would look like

init $\tau_{\{ho_c\}}(\nabla_{\{ho_c\}}(\Gamma_{\{ho_s|ho_r \to ho_c\}}(X_1 \parallel X_2)));$

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 [20, 21]. We will not explain these equivalences here in detail. 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 one or more illustrative systems for which a behaviour must be designed. For each system, a behaviour is provided where the guideline is not used, and subsequently, a description is given where the guideline is used and that is also adequate for the system in the example. 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 component 1 communicates with component 2, and subsequently, component 2 informs 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 component 3. This only requires one transition. By synchronizing 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

DOI: 10.1002/stvr

instance be carried out by introducing an arbiter, or by letting a process higher up in the process hierarchy allow only one of its sub-processes to operate at any time while putting the others temporarily to a halt.

IV **Confluence and determinacy**. When parallel behaviour cannot be avoided, it is useful to model such that the behaviour is τ-confluent [9]. Behaviour is τ-confluent if whenever an internal τ-action and an action a can happen in a state, then it does not matter in which order they are executed. The notion of τ-confluency is strongly related to partial order reduction. If a system is τ-confluent, always a τ-action can be taken ignoring the other actions, while the resulting transition system is still branching bisimilar to the original, obtaining a substantially reduced 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 categorize 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-prioritization. Finally, take care that data is only stored in one way. For example, 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 organized 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 are 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. For example, 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 [22]. Here, the goal is to minimize 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.

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

DOI: 10.1002/stvr

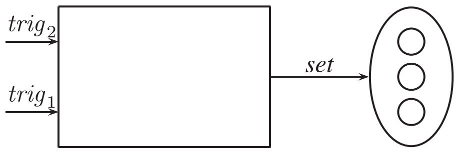

Figure 3. A simple traffic light with two sensors.

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 $\quad Aspect = \textbf{struct}\ green \mid yellow \mid red;$

The pushing controller is very straightforward. The occurrence of $trig_1$ and $trig_2$ indicate that the respective sensors have been triggered. In the pushing strategy, the controller must be able to always deal with incoming signals and store their occurrence for later use. In the succeeding text, the pushing process has two booleans b_1 and b_2 for this purpose. Initially, these booleans are false, and the traffic light is assumed to be red. The booleans become *true* if a trigger is received and are set to *false*, when the traffic light starts with a *green*, *yellow* and *red* cycle.

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

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

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

The transition systems of both systems are drawn in Figure 4. At the left, the diagram for the pushing system is drawn, and at the right, the behaviour of the polling TLC is depicted. The diagram at the left has 12 states and allows a $trig_1$ and a $trig_2$ in every state. The diagram at the right has 5, showing that even for this very simple system polling leads to a smaller state space. In the diagram at the right, the actions $trig_1(b)$ and $trig_2(b)$ only happen when the controller needs to know what the status of the sensor is. Note that not all traces at the left are possible at the right. For instance, the $trig_2$ action is possible in the initial state at the left, but not at the right. Still, both designs for the TLC serve the purpose equally well.

DOI: 10.1002/stvr

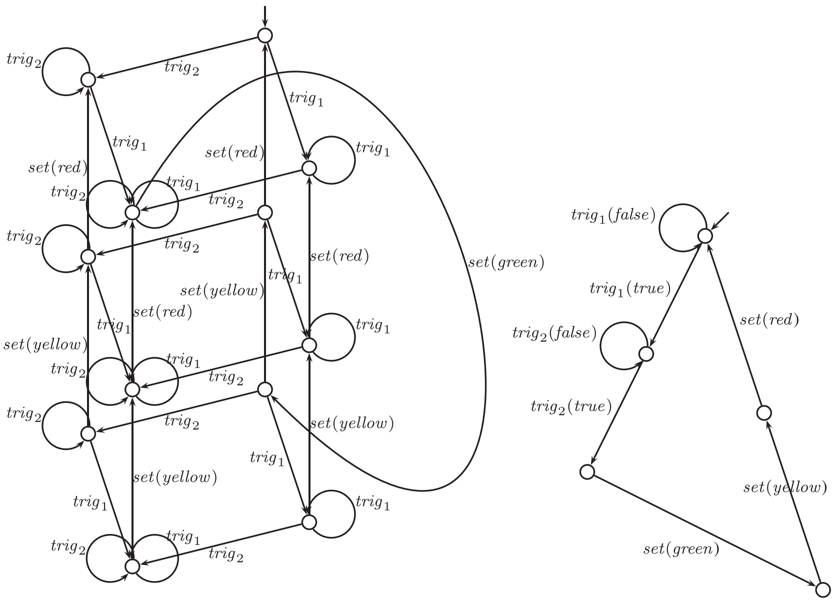

Figure 4. Transition systems of push/poll processes.

Whether information polling can be used very much depends on the system. The sensors should be able to keep the information that they are triggered until this information is requested. Communication should be sufficiently fast, as otherwise the system can become slow and the bandwidth of the communication channel must support a potentially huge number of polling messages.

5. GUIDELINE II: USE GLOBAL SYNCHRONOUS COMMUNICATION

Communication along different components can sometimes be modelled by synchronizing 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 required for every time the message is forwarded. In the second case, the total communication only requires one extra state.

Several formalisms use global synchronous interactions as a way to keep the state space of a system small. The coordination language REO uses the concept very explicitly [23]. A derived form can be found in Uppaal, which uses committed locations [24].

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

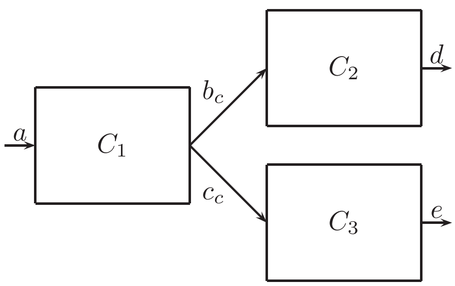

Figure 5. Synchronous/asynchronous message passing.

DOI: 10.1002/stvr

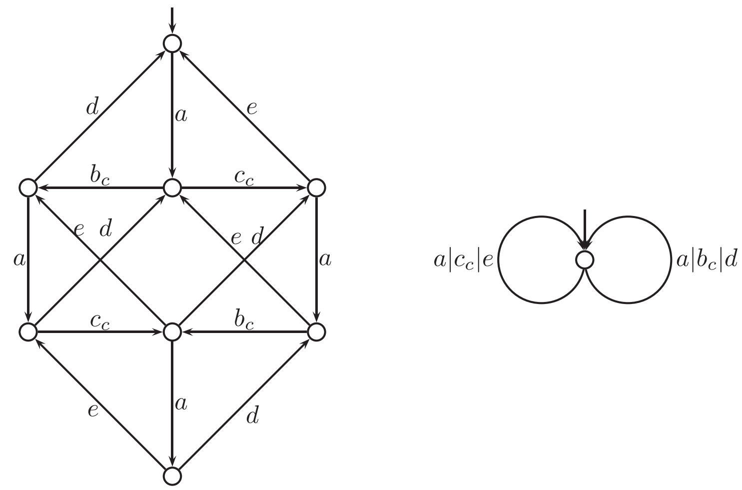

Figure 6. Transition systems of a synchronous and an asynchronous process.

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 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.

The asynchronous variant is described in the succeeding text. Process C_1 performs a, and subsequently performs b_s or c_s, that is, sending via b or c. The process C_2 reads via b by b_r, and then performs a d. The behaviour of C_3 is similar. The whole system consists of the processes C_1, C_2 and C_3, where b_r and b_s synchronize to become b_c and c_r and c_s become c_c. The behaviour of this system contains eight 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 $\nabla_{\{\underline{a},b_c,c_c,\underline{d},\underline{e}\}}(\Gamma_{\{b_r|b_s\to b_c,c_r|c_s\to c_c\}}(C_1||C_2||C_3));$

The synchronous behaviour of this system can be characterized by the following mCRL2 specification. Process C_1 can perform a multi-action $a|b_s$ (i.e. action a and b_s happen exactly at the same time) or a multi-action $a|c_s$. This represents the instantaneous receiving and forwarding of a message. Similarly, C_2 and C_3 read and forward the message instantaneously. The effect is that the state space only consists of one state as depicted in Figure 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 $\nabla_{\{\underline{a}|c_c|\underline{e},\underline{a}|b_c|\underline{d}\}}(\Gamma_{\{b_r|b_s\to b_c,c_r|c_s\to c_c\}}(C_1||C_2||C_3));$

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

There are various justifications of the use of global synchronous communication. The communications modelled by global synchronous communication can be much faster than the other activities of the components. Or these communications can be insignificant relative to the other activities. Of course, those interactions in a system that are the primary object of study in a model are bad candidates to be modelled as global synchronous communications.

6. GUIDELINE III: AVOID PARALLELISM AMONG COMPONENTS

When models have many concurrent components that can independently perform activity, the state space of the given model can be reduced by limiting the number of components that can simultaneously perform their actions. 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.

In some specification languages, explicit avoidance of parallel behaviour between components has been used. For instance, Esterel [25] uses micro steps that 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 [26].

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 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 coordinator and a traffic light cycle starts. Otherwise, the $enter_s$ action is blocked. The *Coordinator* has an internal counter, counting the number of cars. When a *leave* action takes place, the counter is decreased. When a car is allowed to enter (via $enter_r$), the counter is increased.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Coordinator}(\mathit{count}{:}\mathbb{N}) \\
 & \quad = (\mathit{count}>0)\rightarrow \underline{\mathit{leave}} \cdot \mathit{Coordinator}(\mathit{count}-1) \\
 & \quad + (\mathit{count}<N)\rightarrow \mathit{enter}_r \cdot \mathit{Coordinator}(\mathit{count}+1); \\
 & \\
 & \mathit{TLC}(\mathit{id}{:}\mathbb{N}^+) \\
 & \quad = \underline{\mathit{trig}}(\mathit{id}) \cdot \mathit{enter}_s \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{green}) \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{red}) \cdot \mathit{TLC}(\mathit{id}); \\
 & \\
\textbf{init} & \nabla_{\{\underline{\mathit{trig}},\underline{\mathit{show}},\mathit{enter}_c,\underline{\mathit{leave}}\}}(\Gamma_{\{\mathit{enter}_s|\mathit{enter}_r\rightarrow \mathit{enter}_c\}}(\mathit{Coordinator}(0) \| \mathit{TLC}(1) \| \mathit{TLC}(2) \| \\
 & \qquad \mathit{TLC}(3)));
\end{array}
$$

The state space of this control system grows exponentially with the number of TLCs. In columns 2 and 4 of Table I, 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 TLCs. So, we put the initiative in the hands of the coordinator in the second model. It assigns the task of monitoring a sensor to one of the TLCs at a time. The traffic controller will poll the sensor, and only if it has been triggered, it switches the traffic light to green. After it has done its task, the TLC will return control

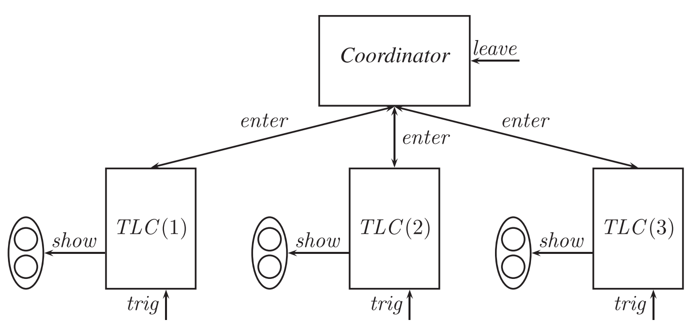

Figure 7. A parking lot with three entrances.

DOI: 10.1002/stvr

Table I. State space sizes of parking lot controllers.

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

N, no. of traffic lights; M, no. of parking places.

to the co-ordinator. Of course, if the parking lot is full, the TLCs are not activated. Note that in this second example, only one traffic light can show green at any time, which might not be desirable.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Coordinator}(\mathit{count}{:}\mathbb{N}, \mathit{active\_id}{:}\mathbb{N}^+) \\
& \quad = (\mathit{count}>0) \to \underline{\mathit{leave}} \cdot \mathit{Coordinator}(\mathit{count}-1, \mathit{active\_id}) \\
& \quad + (\mathit{count}<N) \to \mathit{enter}_s(\mathit{active\_id}) \cdot \sum_{b:\mathbb{B}} \mathit{enter}_r(b) \cdot \\
& \qquad\qquad \mathit{Coordinator}(\mathit{count}+\mathrm{if}(b,1,0), \mathrm{if}(\mathit{active\_id} \approx M, 1, \mathit{active\_id}+1)); \\
& \\
& \mathit{TLC}(\mathit{id}{:}\mathbb{N}^+) \\
& \quad = \mathit{enter}_r(\mathit{id}) \cdot \\
& \qquad (\; \underline{\mathit{trig}}(\mathit{id}, \mathit{true}) \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{green}) \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{red}) \cdot \mathit{enter}_s(\mathit{true}) + \\
& \qquad\;\; \underline{\mathit{trig}}(\mathit{id}, \mathit{false}) \cdot \mathit{enter}_s(\mathit{false}) \\
& \qquad) \cdot \\
& \qquad \mathit{TLC}(\mathit{id}); \\
& \\
\textbf{init} & \nabla_{\{\underline{\mathit{trig}}, \underline{\mathit{show}}, \mathit{enter}_c, \underline{\mathit{leave}}\}}(\Gamma_{\{\mathit{enter}_s | \mathit{enter}_r \to \mathit{enter}_c\}} \\
& \qquad\qquad\qquad (\mathit{Coordinator}(0,1) \,||\, \mathit{TLC}(1) \,||\, \mathit{TLC}(2) \,||\, \mathit{TLC}(3)));
\end{array}
$$

As can be seen in Table I, the state space of the second model only grows linearly with the number of traffic lights.

It very much depends on the nature of the system whether it is allowed to sequentialise parallel behaviour. If the primary purpose of a system is the calculation of values, sequentialising appears to be defendable. If the parallel tasks are relatively small or when a system is not very interactive, it also appears defendable to eliminate parallel behaviour. On the other hand, if the sub-components are controlling all kinds of devices, especially in real time, then the parallel behaviour of the sub-components might be the primary purpose of the system, and sequentialisation cannot be used.

7. GUIDELINE IV: CONFLUENCE AND DETERMINACY

In [9, 21], 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 [9, 21] are slightly different; we use the formulation from [9] because it is more suitable for verification purposes.

A transition system is *τ-confluent* if for every state s that has an outgoing τ-transition 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. It is useful to know that it can be determined on the basis of a behavioural description whether a transition system is τ-confluent without generating the transition system explicitly [9].

When generating 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-prioritization*. It preserves branching bisimulation equiva-

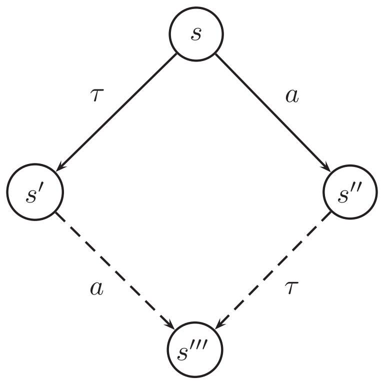

Figure 8. Confluent case.

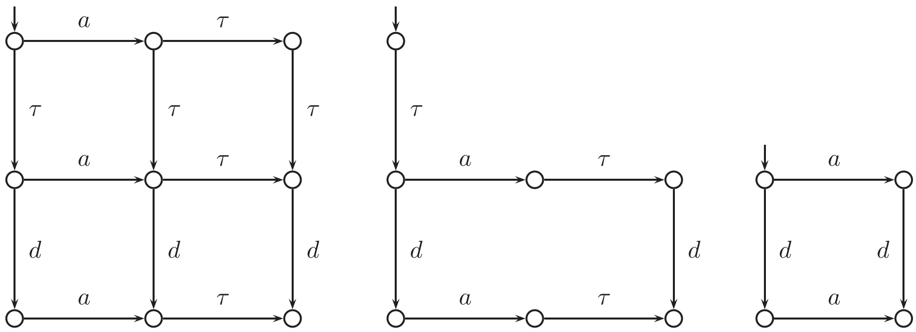

Figure 9. The effect of τ-prioritization and branching bisimulation compression.

lence [20] 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-prioritization is described in [27]. When τ-prioritization 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-prioritization can have a quite dramatic reduction of the size of the state space. This technique allows to generate the prioritized 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-prioritization, and the subsequent merging of branching bisimilar states.

To employ τ-prioritization, 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 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.

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 six components (see Figure 10).

DOI: 10.1002/stvr

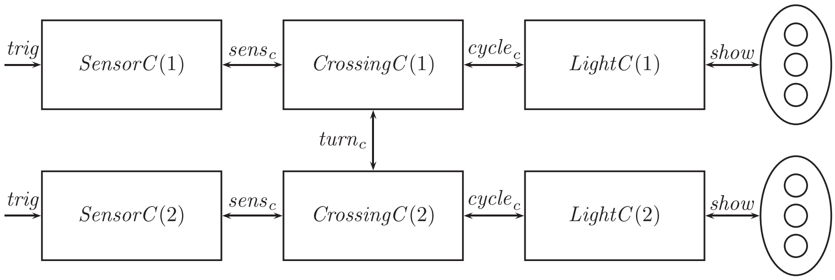

Figure 10. A simple traffic light with two triggers.

For each traffic light, we have a sensor controller *SensorC*, a crossing controller *CrossingC* and a TLC *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).

In the succeeding text, 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 TLC 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 TLC 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.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{SensorC}(id{:}\mathbb{N}^+) = \underline{\mathit{trig}}(id)\cdot \mathit{sens}_s(id)\cdot \mathit{SensorC}(id);\\[1em]
& \mathit{LightC}(id{:}\mathbb{N}^+)\\
& \quad = \mathit{cycle}_r(id)\cdot\\
& \qquad \underline{\mathit{show}}(id, \mathit{green})\cdot\\
& \qquad \underline{\mathit{show}}(id, \mathit{yellow})\cdot\\
& \qquad \underline{\mathit{show}}(id, \mathit{red})\cdot\\
& \qquad \mathit{cycle}_s(id)\cdot\\
& \qquad \mathit{LightC}(id);\\[1em]
& \mathit{CrossingC}(id{:}\mathbb{N}^+, \mathit{my\_turn}, \mathit{sensor\_triggered}, \mathit{cycle}{:}\mathbb{B})\\
& \quad = \mathit{sens}_r(id)\cdot \mathit{CrossingC}(id, \mathit{my\_turn}, \mathit{true}, \mathit{cycle})\\
& \quad + (\mathit{sensor\_triggered} \wedge \mathit{my\_turn} \wedge \neg \mathit{cycle}) \rightarrow \mathit{cycle}_s(id)\cdot\\
& \qquad\qquad \mathit{CrossingC}(id, \mathit{my\_turn}, \mathit{false}, \mathit{true})\\
& \quad + \mathit{cycle}_r(id)\cdot \mathit{CrossingC}(id, \mathit{my\_turn}, \mathit{sensor\_triggered}, \mathit{false})\\
& \quad + \mathit{turn}_r\cdot \mathit{CrossingC}(id, \mathit{true}, \mathit{sensor\_triggered}, \mathit{cycle})\\
& \quad + (\neg \mathit{sensor\_triggered} \wedge \mathit{my\_turn} \wedge \neg \mathit{cycle}) \rightarrow \mathit{turn}_s\cdot\\
& \qquad\qquad\qquad \mathit{CrossingC}(id, \mathit{false}, \mathit{sensor\_triggered}, \mathit{cycle});
\end{array}
$$

Table II. The number of states of the transitions systems for a simple crossing.

	No reduction	After τ-prioritization	Mod branch bis
Non-confluent controller	160	128	124
Simple confluent controller	20	8	8
Complex confluent controller	310	56	56

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

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. The reason for this 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.

In the uppermost row of Table II, the sizes of the state space are given: of the full state space, after applying tau-prioritization and after applying branching bisimulation reduction.

To employ the effect of confluence, we must make the hidden actions $turn_c$ and $sense_c$ confluent, too. We can do this by making the behaviour of the crossing controller *CrossingC* deterministic. A very simple way of doing this is given in the succeeding text. 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 in the succeeding text 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 $SensorC(id{:}\mathbb{N}^+) = sens_r(id) \cdot \sum_{b:\mathbb{B}} \underline{trig}(id, b) \cdot sens_s(id, b) \cdot SensorC(id)$;

$CrossingC(id{:}\mathbb{N}^+, my\_turn{:}\mathbb{B}) =$
my_turn
$\to sens_s(id) \cdot$
$(sens_r(id, true) \cdot$
$cycle_s(id) \cdot$
$cycle_r(id)$
$+$
$sens_r(id, false)$
$) \cdot$
$turn_s \cdot$
$CrossingC(id, false)$
$\diamond\ turn_r \cdot$
$CrossingC(id, true)$;

The state space of this system turns out to be small, namely 20 states (see Table II, second row). It is even smaller after applying τ-prioritization, namely eight states and this is also the size of the state space after branching bisimulation minimisation. This is remarkable, as in general, the state space generated using τ-priorisation can be further reduced by applying branching bisimulation.

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 sensors 1 and 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

DOI: 10.1002/stvr

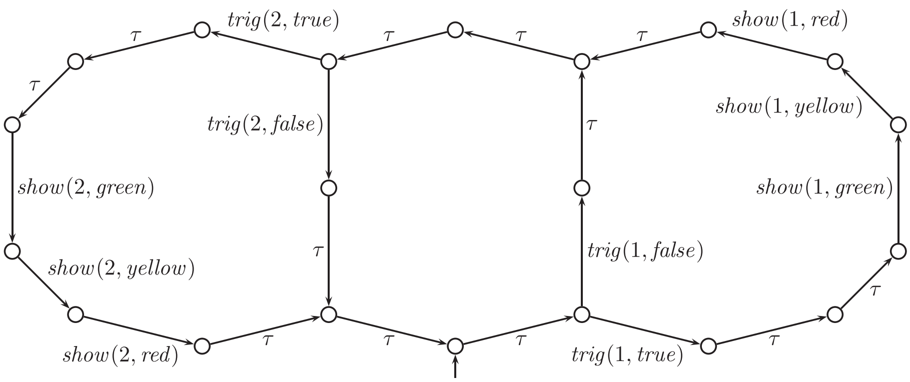

Figure 11. The state space of a simple confluent traffic light controller.

see that this property also holds for more complex controllers, where the polling order is not strictly predetermined.

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-prioritization 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 TLCs inform each other that the turn does not have to switch side. If the crossing controller that has the turn 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.

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 succeeding specification.

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

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

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

DOI: 10.1002/stvr

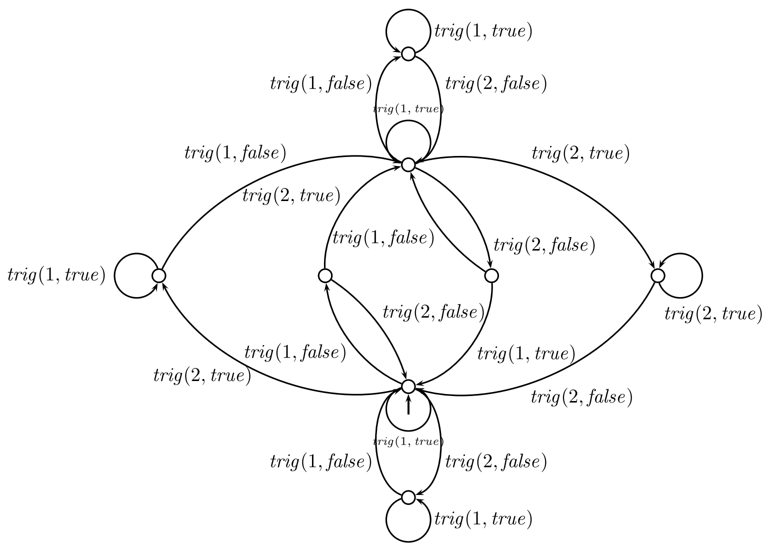

Figure 12. The sensor polling pattern of a more complex confluent controller.

This more complex TLC has a substantially larger state space of 310 states. However, when the state space is generated with τ-prioritization, 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. For example, 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 almost always is), then there are several methods to reduce its footprint. In the succeeding text, we give three examples, one where data is categorized, one where the content of queues is reduced and one where buffers are ordered.

To reduce the state space of a behavioural model, it sometimes helps to group 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 a simple form of abstract interpretation [28, 29]. A given data domain is interpreted in categories where all data elements in one category behave the same. So, the verification only has to be carried out for one value in each category. Generally, a modeller who understands the intended purpose of a model is generally quite capable to define appropriate categories. Our experience is that a tool that employs abstract interpretation is hardly capable to beat the modeller, and only very rarely abstract interpretation tools are capable of categorizing data that a modeller failed to think of.

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 in the succeeding text. Using the action *dist,* the distance to a car is measured, and the action *trig* models the trigger signal.

DOI: 10.1002/stvr

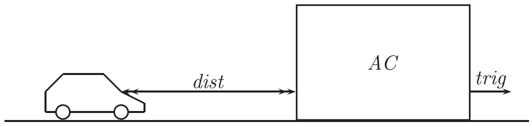

Figure 13. An intelligent approach controller.

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

The state space of this system is a staggering $M^2{+}1$ states big, or more concretely 10 001 states. This is of course because 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 next, leading to a required $M+1$ states, or 101 states in this concrete case.

proc $AC_1(d_{prev}{:}\mathbb{N}) = \sum_{d:\mathbb{N}}(d<M)\to((2d<d_{prev})\to\underline{dist}(d)\cdot\underline{trig}\cdot AC_1(M)\diamond\underline{dist}(d)\cdot AC_1(d))$;
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. The specification of this abstract approach controller *AAC* is given by:

sort $Distance =$ **struct** $near \mid far$;
proc $AAC = \sum_{d:Distance}\underline{dist}(d)\cdot((d\approx near)\to\underline{trig}\cdot AAC\diamond AAC)$;
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 TLCs. 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 organized 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.

We make a straightforward protocol, where we do not look into efficiency. Whenever a TLC 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 to show green, and $started$ indicates that it has started with the protocol described earlier. 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.

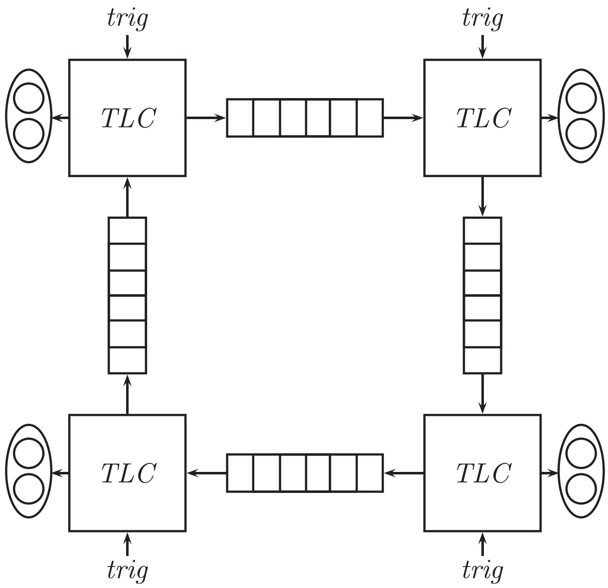

Figure 14. Process communication via unbounded queues.

$$
\begin{array}{ll}
\textbf{sort} & \mathit{Aspect} = \textbf{struct}\ \mathit{green} \mid \mathit{red};\\
 & \mathit{Message} = \textbf{struct}\ \mathit{active}(\mathit{get\_number} : \mathbb{N})?\mathit{is\_active} \mid \mathit{passive}(\mathit{get\_number} : \mathbb{N});\\
\textbf{map} & N : \mathbb{N}^+;\\
\textbf{eqn} & N = 2;\\
\textbf{proc} & \mathit{Queue}(id{:}\mathbb{N}, q{:}\mathit{List}(\mathit{Message})) =\\
 & \quad \sum_{m:\mathit{Message}} q_{in_r}(id, m)\cdot \mathit{Queue}(id, m \triangleright q)+\\
 & \quad (\#q>0)\to q_{out_s}((id+1) \bmod N, \mathit{rhead}(q))\cdot \mathit{Queue}(id, \mathit{rtail}(q));\\
 & \\
 & \mathit{TLC}(id{:}\mathbb{N}, \mathit{triggered}, \mathit{started}{:}\mathbb{B}) =\\
 & \quad \underline{trig}(id)\cdot \mathit{TLC}(id, \mathit{true}, \mathit{started})+\\
 & \quad (\mathit{triggered} \wedge \neg \mathit{started})\\
 & \qquad \to q_{in_s}(id, \mathit{active}(id))\cdot \mathit{TLC}(id, \mathit{false}, \mathit{true})+\\
 & \quad \sum_{m:\mathit{Message}} q_{out_r}(id, m)\cdot\\
 & \qquad ((\mathit{started} \wedge \mathit{is\_active}(m) \wedge \mathit{get\_number}(m) \not\approx id)\\
 & \qquad\quad \to q_{in_s}(id, \mathit{passive}(\mathit{get\_number}(m)))\cdot \mathit{TLC}(id, \mathit{triggered}, \mathit{started})\\
 & \qquad\quad \diamond((\mathit{started} \wedge \mathit{get\_number}(m) \approx id)\\
 & \qquad\qquad \to (\mathit{is\_active}(m) \to \underline{show}(id, \mathit{green})\cdot \underline{show}(id, \mathit{red})\cdot \mathit{TLC}(id, \mathit{triggered}, \mathit{false})\\
 & \qquad\qquad\qquad\qquad\qquad \diamond \mathit{TLC}(id, \mathit{true}, \mathit{false})\\
 & \qquad\qquad\quad)\\
 & \qquad\qquad \diamond q_{in_s}(id, m)\cdot \mathit{TLC}(id, \mathit{triggered}, \mathit{started})\\
 & \qquad\quad));\\
 & \\
\textbf{init} & \tau_{\{q_{in_C}, q_{out_C}\}}(\nabla_{\{\underline{trig}, \underline{show}, q_{in_C}, q_{out_C}\}}(\Gamma_{\{q_{in_r}|q_{in_s}\to q_{in_C}, q_{out_r}|q_{out_s}\to q_{out_C}\}}(\\
 & \quad \mathit{TLC}(0, \mathit{false}, \mathit{false}) || \mathit{TLC}(1, \mathit{false}, \mathit{false}) || \mathit{Queue}(0, []) || \mathit{Queue}(1, []))));
\end{array}
$$

Note that the state space of this system is growing very dramatically with the number of processes. See the second column in Table III. 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 1 terabyte main memory linux machine).

DOI: 10.1002/stvr

Table III. Traffic lights connected with queues.

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

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 next for two processes. The specification of the queue is omitted, as it is exactly the same as the one of the previous specification

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

proc $\mathit{TLC}(\mathit{id}{:}\mathbb{N}, \mathit{active}{:}\mathbb{B}) =$
$\mathit{active} \to (\underline{\mathit{trig}}(\mathit{id}, \mathit{true}) \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{green}) \cdot \underline{\mathit{show}}(\mathit{id}, \mathit{red}) + \underline{\mathit{trig}}(\mathit{id}, \mathit{false})) \cdot$
$q_{in_s}(\mathit{id}, \mathit{token}) \cdot \mathit{TLC}(\mathit{id}, \mathit{false})$
$\diamond\ q_{out_r}(\mathit{id}, \mathit{token}) \cdot \mathit{TLC}(\mathit{id}, \mathit{true});$

init $\tau_{\{q_{in_C}, q_{out_C}\}}(\nabla_{\{\underline{trig}, \underline{show}, q_{in_C}, q_{out_C}\}}(\Gamma_{\{q_{in_r} | q_{in_S} \to q_{in_C}, q_{out_r} | q_{out_S} \to q_{out_C}\}}($
$\mathit{TLC}(0, \mathit{true}) || \mathit{TLC}(1, \mathit{false}) || \mathit{Queue}(0, []) || \mathit{Queue}(1, []))));$

The number of states of the state space for different number of processes are given in the fourth column of Table III. In the fifth and sixth columns, the number of states after τ-prioritization and branching bisimulation reduction are given. Note that the number of states after τ-prioritization is equal to the number of states after applying branching bisimulation. Note also that the differences in the sizes of the state spaces are 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 in the succeeding text.

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

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

init $B([]);$

DOI: 10.1002/stvr

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

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

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 IV, 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.

There are hardly any rules on whether or not it is allowed to reduce data in designs. An obvious statement is that reduction is possible when the particular data is not important. But sometimes, if state spaces are too big, and after all applicable state space reduction guidelines have been applied, it can pay off to overabstract. Of course, any conclusion that can be drawn about the overreduced state space is not valid and should be treated with caution. But despite being possibly invalid, such analysis results may provide insight, more insight at least than no analysis results at all.

9. GUIDELINE VI: COMPOSITIONAL DESIGN AND REDUCTION

When a design of a system is decomposed in several components, it can be wise to organize 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 $C_1, \ldots, C_5$ is depicted. In the middle, the interfaces $I_1, \ldots, I_7$ are also shown. At the right, the system has a tree structure.

When calculating the behaviour of the whole system, a characterization 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 may have too many states. An alternative is to combine

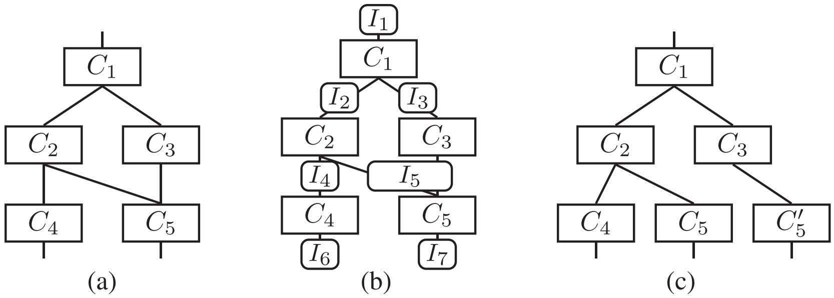

Figure 15. The compositional design and verification steps.

DOI: 10.1002/stvr

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 [30].

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 a smaller 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 the behaviour at dependent but unrestricted interfaces as much as possible.

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 *Controller*s) to 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

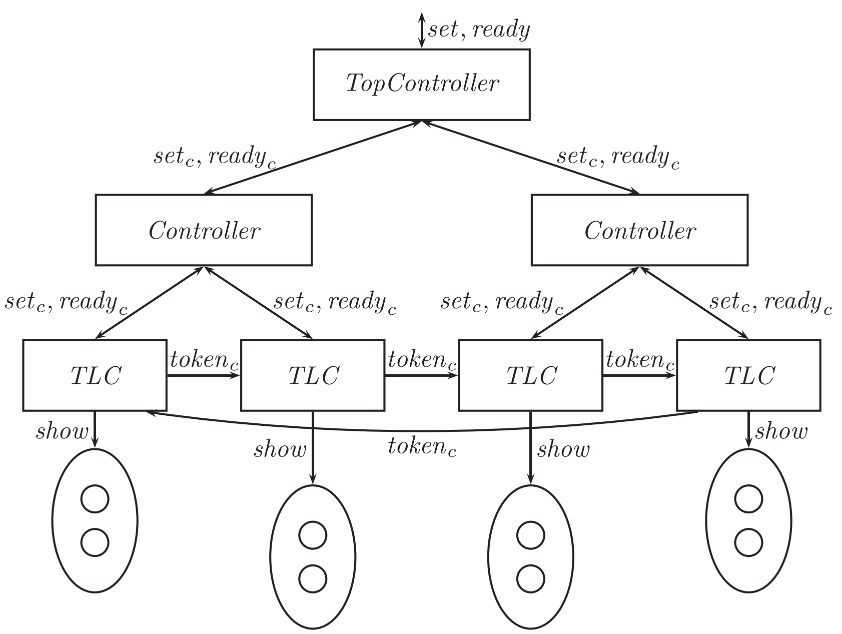

Figure 16. Distribution of system components.

DOI: 10.1002/stvr

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 TLCs 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 TLCs that they must set the light to green, and they transport the ready messages back to the central controller.

The TLCs 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 TLCs. The top controller has level 0. In Figure 16, the level of the TLCs is 2. Furthermore, *has_token* indicates that this TLC 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 succeeding description describes a system with four TLCs.

$$
\begin{array}{ll}
\textbf{sort} & \mathit{Aspect} = \textbf{struct}\ \mathit{green} \mid \mathit{red};\\
\textbf{proc} & \mathit{ControllerTop}(id_{low}, id_{high}{:}\mathbb{N}) =\\
& \quad \sum_{n:\mathbb{N}} (id_{low} \leq n \wedge n \leq id_{high}) \rightarrow (\underline{set}(n)\cdot set_s(n,1) + ready_r(n,1)\cdot \underline{ready}(n))\cdot\\
& \qquad\qquad\qquad \mathit{ControllerTop}(id_{low}, id_{high});\\
& \mathit{Controller}(id_{low}, id_{high}, level{:}\mathbb{N}) =\\
& \quad \sum_{n:\mathbb{N}} (id_{low} \leq n \wedge n \leq id_{high}) \rightarrow\\
& \qquad (set_r(n, level)\cdot set_s(n, level+1)\cdot \mathit{Controller}(id_{low}, id_{high}, level) +\\
& \qquad\ ready_r(n, level+1)\cdot ready_s(n, level))\cdot \mathit{Controller}(id_{low}, id_{high}, level);\\
& \mathit{TLC}(id, level{:}\mathbb{N}, has\_token, busy{:}\mathbb{B}) =\\
& \quad set_r(id, level)\cdot \mathit{TLC}(id, level, has\_token, true) +\\
& \quad (has\_token \wedge busy) \rightarrow \underline{show}(id, green)\cdot \underline{show}(id, red)\cdot ready_s(id, level)\cdot\\
& \qquad\qquad\qquad \mathit{TLC}(id, level, has\_token, false) +\\
& \quad (has\_token \wedge \neg busy) \rightarrow token_s((id+1) \bmod 4)\cdot \mathit{TLC}(id, level, false, busy) +\\
& \quad (\neg has\_token) \rightarrow token_r(id)\cdot \mathit{TLC}(id, level, true, busy);\\
\textbf{init} & \nabla_{\{set_c, ready_c, token_c, \underline{show}, \underline{set}, \underline{ready}\}}(\\
& \quad \Gamma_{\{set_r|set_s \rightarrow set_c,\, ready_r|ready_s \rightarrow ready_c,\, token_r|token_s \rightarrow token_c\}}(\\
& \qquad \mathit{ControllerTop}(0,3) \,||\, \mathit{Controller}(0,1,1) \,||\, \mathit{Controller}(2,3,1) \,||\\
& \qquad \mathit{TLC}(0,2,true,false) \,||\, \mathit{TLC}(1,2,false,false) \,||\\
& \qquad \mathit{TLC}(2,2,false,false) \,||\, \mathit{TLC}(3,2,false,false)));
\end{array}
$$

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 V for a system with four and eight TLCs. 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

Table V. State space sizes for a hierarchical traffic light controller.

	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	$3.06\ 10^3$

DOI: 10.1002/stvr

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 next.

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

The resulting state spaces are given in Table V 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 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 exchange of the token as low level control is not needed anymore. This is possible because the top controller now guarantees that at most one traffic light shows green. This is carried out by replacing the specification of the TLC by the simple succeeding specification. Note that the communication topology of the system now has a tree structure.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{TLC}(id, \mathit{level}{:}\mathbb{N}) =\\
& \qquad \mathit{set}_r(id, \mathit{level}) \cdot \underline{show}(id, \mathit{green}) \cdot \underline{show}(id, \mathit{red}) \cdot \mathit{ready}_s(id, \mathit{level}) \cdot \mathit{TLC}(id, \mathit{level});
\end{array}
$$

We are not interested anymore in the behaviour of the system with all the TLCs 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.

DOI: 10.1002/stvr

Designing tree like communication topologies is very helpful when the state space is constructed in a stepwise manner. But such a hierarchical structure can not always be used, not only because a system has a ring or mesh topology, but also because a complex system receives information from a few sensor and controls a few actuators.

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.

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 with 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 [31], 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 [32]. The purpose here is primarily to allow a designer to keep in control over his system.

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 [33], 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 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 $bool$ indicates whether the controller is switched on or off and the natural number i is the current position in the buffer, which the controller uses to cycle through the buffer elements. It sends an action *out* whenever it encounters an element that is set to *true*. The internal action *int* takes place when the controller moves to investigate the next buffer place.

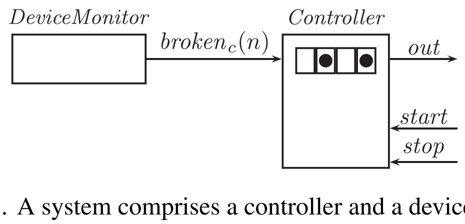

Figure 17. A system comprises a controller and a device monitor.

DOI: 10.1002/stvr

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

The total number of devices is denoted by M. All positions of *buff* are initially set to *false* as indicated by the lambda expression $\lambda n{:}\mathbb{N}.\mathit{false}$. In this specification, the controller blocks the *stop* request if there is a defected device at index i of the buffer forming a dependency between external and internal behaviour. If we calculate the state space of the external behaviour of this system with $M = 2$ and apply a branching bisimulation reduction [20], we obtain the state space depicted in Figure 18 at the left. Note that the behaviour is remarkably complex. In particular, a number of τ-transitions complicate the transition system. But they cannot be removed as they are essential for the perceived external behaviour of the system. Table VI provides the number of states produced as a function of the number of devices monitored in the system. The table shows that the state space of the original system and the state space capturing the external behaviour are comparable. This indicates a complex external behaviour that might complicate verification with external parties and

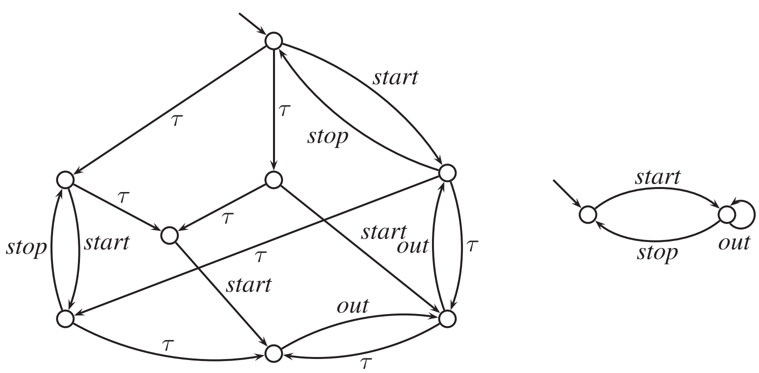

Figure 18. The system external behaviour.

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

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

DOI: 10.1002/stvr

makes understanding the behaviour quite difficult. It might come as a surprise that the external state space of the system is large. Actual expectation is that it should be small, matching the succeeding specification, depicted in the transition system in Figure 18 at the right.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Stopped} = \underline{\mathit{start}}\cdot \mathit{Started};\\
 & \mathit{Started} = \underline{\mathit{out}}\cdot \mathit{Started} + \underline{\mathit{stop}}\cdot \mathit{Stopped};\\
\textbf{init} & \mathit{Stopped};
\end{array}
$$

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 $\mathit{buff}(i)$ is true is the cause of the problem. If we remove this from the condition of the stop action, we obtain the mCRL2 specification of the *DeviceMonitor* process. In this specification, the *stop* request is processed independently from the rest of the behaviour.

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

As can be seen from Table VI, the number of states of the non-reduced model remains the same. However, the reduced behaviour is exactly the one depicted in Figure 18 at the right for any constant M.

We feel that people hardly realize how small changes in an implementation have a far reaching influence on the complexity of the external behaviour of a system. But it is always possible to make an explicit description of the external behaviour of a system, and by doing so the understanding and awareness of this external behaviour grows. Comparing the external behaviour with an implementation using branching bisimulation quickly reveals where they differ. It depends on the actual application whether an implementation can be adapted such that its behaviour becomes branching bisimilar to the external behaviour. But if this would be done systematically, this would not only yield smaller state spaces for external behaviour, it would also lead to far less intricate behaviour of some systems.

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 insights in the models that are 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, and components such as coils and capacitors are made to fit the linearity of the models. 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. Although we illustrated the guidelines with process algebraic models in mCRL2, this does not mean that they are only applicable in a process algebraic context. On the contrary, the guidelines are applicable in any behavioural modelling context, as long as this context provides sufficiently strong language primitives.

There is no claim that our set of guidelines is complete, or even that these seven guidelines are the most important model reduction techniques. It might even be that some of the guidelines may have an adverse effect in combination with certain verification techniques, for example, avoiding parallelism may combine badly with symbolic state space representations. 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 and designing the behaviour of systems of which the behavioural properties can be formally analysed.

ACKNOWLEDGEMENTS

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

REFERENCES

1. Formal methods for industrial critical systems. *Conference Proceedings*; 1996–2014.
2. Formal Methods in System Design. *Journal,* vol. 1–44, Springer Verlag,1992–2014.
3. Wang L-T, Wu C-W, Wen X. *Design for Testability. VLSI Test Principles and Architectures*. Morgan Kaufmann Publishers: San Francisco, California, 2006.
4. Voas JM, Miller KW. Software testability: the new verification. *IEEE Software* 1995; **12**(3):17–28. DOI: 10.1109/52.382180.
5. Lin FJ, Chu PM, Liu MT. Protocol verification using reachability analysis: The state space explosion problem and relief strategies. *ACM SIGCOMM Computer Communication Review* 1987; **17**(5):126–135. DOI: 10.1145/55483.55496.
6. Holzmann GJ. Reliable software development: analysis-aware design. In *Proceedings of TACAS 2011*, Abdulla PA, Leino KRM (eds), LNCS 6605. Springer Verlag: Berlin, 2011; 1–2. DOI: 10.1007/978-3-642-19835-9_1.
7. Vissers CA, Scollo G, van Sinderen M, Brinksma E. Specification styles in distributed systems design and verification. *Theoretical Computer Science* 1991; **89**:179–206. DOI: 10.1016/0304-3975(90)90111-T.
8. Baumgartner J, Mony H. Scalable liveness checking via property-preserving transformations. In *Proceedings of Design, Automation and Test in Europe, Nice*. IEEE: France, 2009; 1680–1685. DOI 10.1109/DATE.2009.5090933.
9. Groote JF, Sellink MPA. Confluence for process verification. *Theoretical Computer Science* 1996; **170**(1-2):47–81. DOI: 10.1016/S0304-3975(96)80702-X.
10. Groote JF, Mathijssen AHJ, Reniers MA, Usenko YS, van Weerdenburg MJ. Analysis of distributed systems with mCRL2. In *Process Algebra for Parallel and Distributed Processing*, Alexander M, Gardner W (eds). Chapman Hall: London, 2009; 99–128. DOI: 10.1007/978-3-642-36742-7_15.
11. MCRL2 homepage. 2014. Available from: www.mcrl2.org. [10 March 2014].
12. Groote JF, Osaiweran A, Schuts MTW, Wesselius JH. Investigating the effects of designing industrial control software using push and poll strategies. *Computer Science Report 11-16*, Eindhoven University of Techology, 2011.
13. Groote JF, Osaiweran A, Wesselius JH. Analyzing a controller of a power distribution unit using formal methods. In *Proceedings of ICST2012 (Montreal, Canada)*. IEEE: Los Alamitos, CA, USA, 2012; 420–428. DOI: 10.1016/j.entcs.2013.04.005.
14. Groote JF, Osaiweran A, Wesselius JH. Experience report on developing the Front-end client unit under the control of formal methods. In *SAC2012 (Software Engineering Track)*. ACM: Riva del Garda, Italy, 2012; 1183–1190. DOI: 10.1145/2245276.2231962.
15. Groote JF, Osaiweran A, Wesselius JH. Analyzing the effects of formal methods on the development of industrial control software. In *Proceedings of ICSM2012*. IEEE: Williamsburg, VA, 2011; 467–472. DOI: 10.1109/ICSM.2011.6081983.
16. Osaiweran A, Fransen T, Groote JF, van Rijnsoever BJ. Experience report on designing and developing control components using formal methods. In *Proceedings of Formal Methods*, Giannakopoulou D, Mery D (eds), LNCS 7436. Springer-Verlag: Berlin, 2012; 341–355. DOI: 10.1007/978-3-642-32759-9_29.
17. Groote JF, Kouters TWDM, Osaiweran AAH. Specification guidelines to avoid the state space explosion problem. In *Proceedings of the 4th International Conference on Fundamentals of Software Engineering (FSEN 2011)*, vol. 7141, Lecture Notes in Computer Science. Springer Verlag: Berlin, 2012; 112–127. DOI: 10.1007/978-3-642-29320-7_8.
18. Baeten JCM, Weijland WP. Process algebra. *Cambridge Tracts in Theoretical Computer Science,* Cambridge University Press: Cambridge, 1990; **18**.
19. Groote JF, Mousavi MR. *Modeling and Analysis of Communicating Systems*. MIT-press. 2014.

20. van Glabbeek RJ, Weijland WP. Branching time and abstraction in bisimulation semantics. *Journal of the ACM* 1996; **43**(3):555–600. DOI: 10.1145/233551.233556.
21. Milner R. *A Calculus of communicating systems*, Lecture Notes in Computer Science, vol. 92. Springer Verlag: Berlin, 1980.
22. Acharya S, Franklin M, Zdonik S. Balancing push and pull for data broadcast. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data,* Tucson, Arizona, 1997:183–194. DOI: 10.1145/253262.253293.
23. Arbab F. Reo: A Channel-based coordination model for component composition. *Mathematical Structures in Computer Science;* Cambridge University Press 2004; **14**(3):329–366. DOI: 10.1017/S0960129504004153.
24. Larsen KG, Pettersson P, Yi W. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer* 1997; **1**(12):134–152. DOI: 10.1007/s100090050010.
25. Berry G, Gonthier G. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 1992; **19**:87–152. DOI: 10.1016/0167-6423(92)90005-V.
26. Holzmann GJ. *The SPIN Model Checker. Primer and Reference Manual*. Addison-Wesley: Boston, 2003.
27. Blom SCC, van de Pol JC. State space reduction by proving confluence. In *Proceedings of 14th International Conference on Computer Aided Verification (CAV'02)*, vol. 2404, Brinksma E, Larsen KG (eds), Lecture Notes in Computer Science. Springer Verlag, 2002; 596–609.
28. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by constrction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press: New York, 1977; 238–252. DOI: 10.1145/512950.512973.
29. Dams D, Gerth R, Grumberg O. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1997; **19**(2):253–291. DOI: 10.1145/244795.244800.
30. Garavel H, Lang F, Mateescu R, Serwe W. CADP 2006: A toolbox for the onstruction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'2007, Berlin, Germany)*, vol. 4590, Lecture Notes in Computer Science. Springer Verlag; Berlin, 2007; 158–163. DOI: 10.1007/978-3-540-73368-3_18.
31. Vissers CA, Logrippo L. The importance of the service concept in the design of data communications protocols. In *Protocol Specification, Testing and Verification (Proceedings of the IFIP WG 6.1 Fifth International Workshop on Protocol Sepcification, Testing and Verification)*, Diaz M (ed.) Elsevier: North Holland, 1986; 3–17.
32. Broadfoot GH. ASD case notes: costs and benefits of applying formal methods to industrial control software. In *Proceedings of Formal Methods Conference (FM 2005)*, LNCS 3582. Springer Verlag: Berlin, 2005; 548–551. DOI: 10.1007/11526841_39.
33. Tanenbaum AS. *Computer Networks*, Second edition. Prentice Hall, 1988.