Verifying System-Wide Properties of Industrial Component-Based Software

Thomas Neele, Marijn Rol, and Jan Friso Groote

Eindhoven University of Technology, The Netherlands
{T.S.Neele,J.F.Groote}@tue.nl

Abstract. Analytical Software Design (ASD) enables model-based development of component software systems. Until now, functional verification of ASD systems is only possible on a per-component basis. There is no functional verification engine for ASD itself, so this verification relies on a translation of individual components to mCRL2, a process-algebraic model checker. We show how to extend the ASD-mCRL2 translation to support multiple components in order to enable checking of system wide functional properties. With our extended translation, we perform a case-study on a newly developed industrial system consisting of 26 communicating components. The results indicate that it is feasible to model check functional properties on this scale.

1 Introduction

Modern high-tech industry relies more and more on software to implement supervisory control logic. With the large number of software components in a typical machine, the software can become very complex. The industry not only wants software that meets high quality standards to assure safety and reliability, but the reduction of the costs and time of development also plays an important role. This is assured by model based software design accompanied with formal analysis where software problems are eradicated as soon as possible in the development process. Comparative research shows that it is possible to reduce the number of bugs by a factor 10 and the development time by a factor 3 [14].

Analytical Software Design (ASD) [3,4] is one of the model based engineering tools being used in industrial environments. Using ASD, software engineers develop models which can be checked for various properties such as deadlock/livelock freedom and interface compliance with a single press of a button. From the models, ASD generates executable code, e.g. Java or C++, that can be run in a production environment.

In contrast to many other tools that apply model checking techniques, ASD does not suffer severely from the *state-space explosion* problem. This is achieved through the application of compositional verification techniques: each component in a system is checked individually by comparing its implementation and interface using *failure divergence refinement* (FDR) [17]. A pleasant property of FDR is that deadlock/livelock freedom of each component guarantees deadlock/livelock freedom of the complete system. The ASD approach has been used to develop systems with over 200 components [12] (more than 300 models if interface and design models are counted separately), where total verification takes around 20 minutes.

It is also possible to check a broader range of properties on single components through a translation to mCRL2 [5,11]. The process algebraic description language mCRL2 comes with a toolset for simulating, visualising, manipulating and model checking behavioural specifications. We call the functional properties that span a single component *local properties*.

There are however many global properties, also called 'end-to-end' properties, that are not covered by only checking local properties. Typical examples are:

- If the software control is instructed to manufacture a product, then the appropriate associated low level instructions are always issued.
- If one of the actuators reports an error, the control system always reports the error to the higher software layers.
- If the control software reports that the machine is off, it will never instruct any of its controlled actuators to move.

In this paper we report on how we verify such global properties on newly developed industrial control software. For this purpose we extend the existing mCRL2 translation to support multiple components. Firstly, communication has to be restricted to take place between the right components. Furthermore, important functionality that was implemented in C++, instead of ASD, needs to be translated manually to mCRL2. Finally, we must add several mechanisms to preserve the single-threaded execution as defined by the semantics, *i.e.*, we must ensure that only one component is active at a time.

We evaluate the approach by translating an ASD system that is newly developed and which consists of 26 components (together containing 5054 so called rule cases). On the resulting mCRL2 model, we check a complete set of end-to-end properties. Because the state-space of the system only consisted of 178 million states, we were able to establish whether each requirement was satisfied.

Our expectation was that we would encounter many hardships in the verification, especially because the state spaces would be excessively large. But the contrary turned out to be true. The state space remained well within acceptable limits for the available computer equipment. The reason for this appears to lie in the run-to-completion semantics employed in ASD, together with the strict use of interfaces. These coincide with design rules for systems to avoid the state space explosion [10]. There were a number of minor issues that had to be overcome, such as speeding up writing intermediate results to disk. It also turned out that applying a branching bisimulation reduction to the intermediate state space before applying model checking was much more time efficient than following the ordinary workflow.

Our conclusion is that it is very well possible to verify actual industrial-size software while it is under development. But for success, it needs to be written in an appropriate domain specific language whose semantics avoids a state space explosion.

Overview Section 2 introduces the basic concepts on which this paper is built. We explain our approach to multi-component translation in Section 3. The case study is introduced in Section 4 together with the properties we verify and Section 5 contains the results of the experiments we conducted. In Section 6, we give an overview of some related work. Finally, Section 7 presents a conclusion.

2 Background information

This section provides a short introduction to ASD, mCRL2 and the modal μ-calculus, which form the bases of our approach.

2.1 Analytical Software Design

Analytical Software Design (ASD) [4], developed by Verum, enables the development of software based on communicating components. The components are designed and verified using the ASD:Suite. Furthermore, the ASD:Suite can generate the executable code that can be used in a production environment. In ASD, there are two types of components:

– *Standard ASD component.* A standard component consists of an *interface* and a *design* model. The interface model specifies the externally visible behaviour of a component. It provides a more abstract view on the behaviour of a component. The design model specifies the inner working of a component, including how it interacts with lower level components. The design model always refines the interface model under failure-divergence refinement [17].
– *Foreign component.* A foreign component consists of only an interface model. It typically models the behaviour of a hardware component or another system that is implemented outside of ASD, *e.g.*, in C++.

If a component A relies on another component B for certain tasks, then we say A is a *client* of B and B is a *server* of A. Intuitively, an interface model serves as a contract on the behaviour of the corresponding design model. The interface model specifies exactly in which order a client can send calls to a server and which responses it can expect. Most of the decision logic of a component is contained in the design model. In ASD, the components must be structured as a tree, *i.e.*, a component cannot have more than one client. See Figure 1.

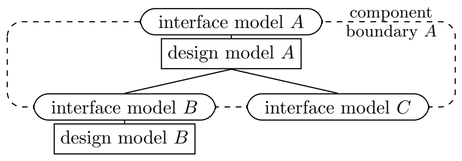


Fig. 1. An example system composed of two standard components, A and B, and a foreign component C.

Within ASD, we distinguish four types of communication: *call events*, *reply events*, *notification events* and *modelling events*. A call event happens when a client wants to request certain information or a certain action from one of its servers. While the server is handling that call event, it may choose to send one or more notification events to its client. These notifications are stored in the *notification queue* of the client until control is given back to the client. The notification queue will be explained in detail later. When the server is finished with a call event, it will always send a reply event to the client. When a reply contains data, we call it a *valued reply*, otherwise it is called a *void reply.*

A modelling event typically represents low level input, such as an interrupt, which can be guaranteed to come, or which can incidentally take place. It can only be performed by a foreign component. Similar to notification events, they are sent from a server to a client. However, whereas notification events always happen as the result of a call event, modelling events only happen spontaneously.

All ASD models are described using a formalism similar to *extended finite state machines*, which are state machines augmented with data. Software engineers develop these state-machines in the ASD:Suite in a format called *Sequence-based Specification* (SBS). A model can contain *state variables*, which store information about the state of the component. State variables can be Booleans or any other finite enumeration type. Transitions can be guarded with expressions over the state variables, and state variables can be updated after every transition. A guarded transition can only be taken when the guard evaluates to *true* given the current values of the state variables. A row in an SBS is called a *rule case.* We give a formal definition of a design model, which we will use later to highlight the most important aspects of the translation to mCRL2.

Definition 1. A design model is a tuple $DM = (S, V, T, (\hat{s}, \hat{v}))$, where

- S is a set of states;
- V is a set of state variables $v_1{:}D_1, \ldots, v_n{:}D_n$, and their types D_i are finite enumeration types;
- T is a set of transitions, defined as

  $$\begin{aligned} T \subseteq\; & S \times \Phi(V) \times \mathit{Call} \times (\mathit{Event}^* \times (\mathit{Reply} \cup \{\varnothing\}) \times \mathcal{A}(V)^* \times S \cup \{\mathit{Illegal}\}) \\ & \cup S \times \Phi(V) \times (\mathit{Reply} \cup \mathit{Notif}) \times (\mathit{Event}^* \times \mathcal{A}(V)^* \times S \cup \{\mathit{Illegal}\}) \end{aligned}$$

  A transition has a source state and a guard, and can originate either from a call event, from a notification that is stored in the queue or from a valued reply. If the transition is not illegal, it results in zero or more calls and/or notifications, a reply (when necessary), assignments to the state variables and a state update.
- $\hat{s} \in S$ and $\hat{v} \in D_1 \times \cdots \times D_n$ define the initial state and the initial value of the state variables, respectively.

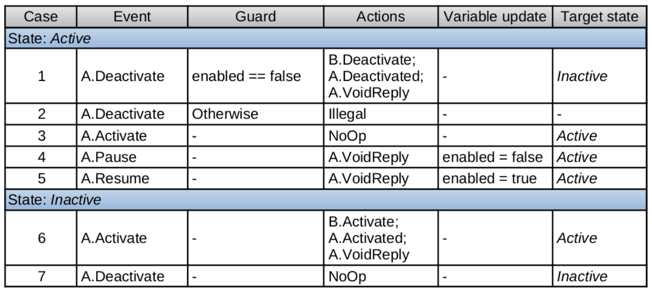
Here, *Call* and *Notif* are the set of all call events and notifications respectively, $\mathit{Event} = \mathit{Call} \cup \mathit{Notif}$ is the set of all event, $\Phi(V)$ is the set of all possible guards over V, *Reply* is the set of all valued replies, $\varnothing$ is a void reply and $\mathcal{A}(V)$ is the set of all possible assignments over V.

To simplify the reasoning, in the theory that is presented here, we assume that none of the events carry data values. In ASD, the data that is carried by events can only be forwarded and not inspected, so the assumption in the definition is not restrictive. In the case study of Section 4, we do consider all of ASD's features including communication of data.

Example 1. We consider a component A that can be activated and deactivated and also paused and resumed. Component A is a client of component B, which always needs to be deactivated before component A can become inactive. A sequence-based specification of the design model of component A is given in Figure 2: it shows the four rule cases of state *Active* and one rule case of the state *Inactive*. When one of the actions *Pause* or *Resume* is performed, an empty reply (*VoidReply*) is returned and the variable *Enabled* is updated (rule cases 4 and 5). Component A can only be deactivated when it is not *Enabled* (rule cases 1 and 2). Upon deactivation, it first sends a message to component B, and only then deactivates itself by going to the state *Inactive*. Activation from the state *Inactive* happens in a similar way. We assume the actions not shown for one of the states are *blocked* in that state, *i.e.*, they cannot happen. □

There are two possible semantics for ASD: the *multi-threaded execution model* and the *single-threaded execution model* [12]. In this paper, we only consider the latter. In the single-threaded execution model events cannot happen in parallel, but only in sequence. Therefore, these semantics should define clearly in which order events are processed. According to the documentation of ASD, the following actions take place in order when a component receives an event from a client:
- All actions from the SBS rule case are processed in order.
- State variables are updated.
- The transition to the target state is taken.
- The notifications in the queue are processed. No events other than those caused by these notifications may occur before the queue has been emptied.
- A void or valued reply takes place to give control back to the client.

These rules are also referred to as *run-to-completion semantics*, meaning that a component completes all of its tasks before relinquishing control to another component. Note that since events and notifications can arrive in different orders, there are still many potential runs in an ASD model.



Case	Event	Guard	Actions	Variable update	Target state
State: *Active*					
1	A.Deactivate	enabled == false	B.Deactivate; A.Deactivated; A.VoidReply	-	*Inactive*
2	A.Deactivate	Otherwise	Illegal	-	-
3	A.Activate	-	NoOp	-	*Active*
4	A.Pause	-	A.VoidReply	enabled = false	*Active*
5	A.Resume	-	A.VoidReply	enabled = true	*Active*
State: *Inactive*					
6	A.Activate	-	B.Activate; A.Activated; A.VoidReply	-	*Active*
7	A.Deactivate	-	NoOp	-	*Inactive*

Fig. 2. Example of an SBS for the design model of a component called A.

Wrapper Components. The tree structure of ASD is quite restrictive in practice, since it does not allow a component to have more than one client. That makes it impossible to implement bidirectional communication channels that have two clients, or database-like components that have many clients. One can work around this issue by manually implementing foreign components, as we explain below.

In the case study that we consider, wrapper components are used to implement symmetric communication channels, see Figure 3. In this case, there are three components A, B and C. The components A and B are both a client of C, albeit through two separate interfaces. The router – written manually in C++ – implements both these interfaces. Since ASD is not aware of the connection between both interfaces, this structure does not violate ASD's single-client constraint.

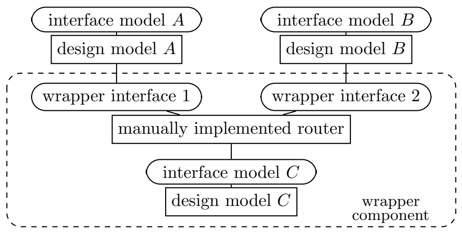


Fig. 3. The structure of a wrapper component.

The router forwards all requests from components A and B to C, and also sends responses from C back to the correct client. For all requests component C receives from component A, it sends a notification to B, and vice versa. In this way, the wrapper component C serves as a bidirectional communication channel.

2.2 mCRL2

The language mCRL2 is a process-algebraic language [11] which can be analysed using the accompanying toolset [5]. The main aim of mCRL2 is model checking of parallel processes. Additionally, mCRL2 can generate, reduce, compare and visualise state-spaces.

The mCRL2 modelling language is very flexible and despite a limited set of language primitives, very expressive. Therefore, it is very well suited as a target language for automatic translations. Several basic operators that we deal with in this paper are sequential composition (operator $\cdot$), choice (operator $+$), sum (operator $\sum$, which generalises choice) and conditional (operator $\_ \rightarrow \_ \diamond \_$).

Example 2. To illustrate some of the concepts behind mCRL2, we consider the following specification.

$$\begin{array}{ll}
\textbf{act} & \mathit{tick}, \mathit{reset}, \mathit{press};\\
\textbf{proc} & \mathit{Clock}(n{:}\mathit{Nat}) = \mathit{tick}.\mathit{Clock}(n{+}1) + \mathit{reset}.\mathit{Clock}(0);\\
 & \mathit{Button} = \mathit{press}.\mathit{reset}.\mathit{Button};\\
\textbf{init} & \textbf{allow}(\{\mathit{tick}, \mathit{reset}, \mathit{press}\},\\
 & \qquad \textbf{comm}(\{\mathit{reset}|\mathit{reset} \rightarrow \mathit{reset}\},\\
 & \qquad\qquad \mathit{Clock}(0) \,||\, \mathit{Button}));
\end{array}$$

In this system, we have two processes, a *Clock* and a *Button*. The clock can perform an action *tick*, after which it increases the time (stored in parameter n). When the button is pressed, it subsequently communicates with the clock via the action *reset*, and the clock resets its counter. Communication is enforced through the combination of the **allow** and **comm** operators, which in this case express that both *reset* actions must happen synchronously. For a more complete overview of the mCRL2 language, see [11]. □

2.3 Modal µ-calculus

To express formal properties, the mCRL2 toolset relies on the modal µ-calculus [13] with data, which is suitable to express virtually any conceivable behavioural property. It is far more expressive than LTL/CTL, but it is equally efficient when it comes to establishing those properties [6]. Here, we provide the core grammar of the µ-calculus, *i.e.*, without the use of data:

$$\phi ::= \mathit{false} \mid \mathit{true} \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \Rightarrow \phi \mid \langle a \rangle \phi \mid [a]\phi \mid \mu X.\, \phi \mid \nu X.\, \phi \mid X$$

Here a is an action, X is a fixpoint variable representing a set of states and $\mu X.\, \phi$ and $\nu X.\, \phi$ are the least and greatest fixpoints over X respectively. Formulae in the µ-calculus are interpreted over labelled transition systems. The semantics is roughly as follows. The Boolean operators have their usual semantics. The diamond modality $\langle a \rangle \phi$ is *true* if and only if an a-step is possible after which ϕ holds. The box modality $[a]\phi$ expresses that after every possible a-step, ϕ must hold. The least fixpoint $\mu X.\, \phi$ is true for the smallest set of states X such that ϕ holds for all states in X. Note that X can occur in ϕ. Dually, $\nu X.\, \phi$ is true for the largest set X that satisfies ϕ. The least fixpoint operator expresses that a property must be valid within a finite number of iterations, whereas the greatest fixpoint also allows for infinitely repeating behaviour.

The mCRL2 toolset also allows specifying modalities with sets of actions via so-called *action formulas*: *true* represents the set of all actions and *false* represents the empty set of actions. Supported operators are union, intersection and inverse ($\overline{a}$ is the set of all actions other than a). Furthermore, sequences of actions can be specified with *regular formulas*, which give the possibility to concatenate sequences (with the . operator), take their union (operator $+$) or iterate over them (operator *). The action formula a^* represents zero or more occurrences of the sequence a, and true^* represents any sequence of actions. For example, $[a.b]\mathit{false}$ means that a sequence $a\,b$ is not possible (since if it is possible, *false* must hold in the resulting state) and $\langle a^*.b \rangle \mathit{true}$ means that a sequence consisting of zero or more a's followed by a b is possible. Action formulas and regular formulas can always be expressed using the fixed point operators, but they are generally more convenient to specify concrete behavioural properties.

In this work, we do not consider µ-calculus formulae with data. In mCRL2, data variables can be bound in quantifiers, *i.e.*, $\exists$ or $\forall$, or as parameter of a fixpoint variable. They are used in conditions, and as parameters of actions and fixpoint variables. See [11] for a complete overview of the modal µ-calculus.

3 Approach

The existing translation from ASD to mCRL2 is only capable of translating the models within one component boundary at a time [12]. This translation yields two mCRL2 specifications: one containing the topmost interface model and one containing the design model and the interface models below it. On the latter specification, we can check local properties that concern only that component.

We define a new translation that yields a single mCRL2 specification that represents the behaviour of the complete system. The new translation takes as input all the design models and also the interface models of foreign components. Before we introduce the challenges introduced by the new translation in detail, we first introduce the basic single-component translation.

3.1 Translating single components

Due to the expressivity of mCRL2, ASD components can be mapped almost directly to mCRL2. First, for every state of the component, a recursive process is created. This process carries one parameter for each state variable of the ASD model. For each rule case, this process has one *action summand*, which contains the condition, actions, variable updates and target state deduced from the rule case. Furthermore, the mCRL2 specification contains a *Queue* process that represents the behaviour of the notification queue. A complete definition of the translation can be found in [12].

Definition 2. Given a design model $DM_A = (S, V, T, (\hat{s}, \hat{v}))$, the mCRL2 process that corresponds to its initial state is defined according to the function Tr:

$$Tr(S, (v_1{:}D_1, \ldots, v_n{:}D_n), T, (\hat{s}, \hat{v})) = P_{\hat{s}}(\hat{v}, \bot)$$

where for each state $s \in S$, the corresponding recursive process in mCRL2 is defined as $P_s(v_1{:}D_1, \ldots, v_n{:}D_n, rv{:}\mathit{Reply}) = \sum_{t \in T} Tr(t)$, with

$$\begin{aligned}
&Tr(s, \varphi, e^r, (e^s_1, \ldots, e^s_m), r, (a_1, \ldots, a_k), s') = \\
&\quad \varphi \to e^r \cdot Tr_s(e^s_1) \cdot \cdots \cdot Tr_s(e^s_m) \cdot \\
&\qquad (\mathit{qEmpty} \cdot \mathit{sendReply}(r) \cdot P_{s'}(a_1, \ldots, a_k) + \\
&\qquad\; \mathit{qNonEmpty} \cdot P_{s'}(a_1, \ldots, a_k, rv = r)) \\
&Tr(s, \varphi, e^r, (e^s_1, \ldots, e^s_m), (a_1, \ldots, a_k), s') = \\
&\quad \varphi \to Tr_r(e^r) \cdot Tr_s(e^s_1) \cdot \cdots \cdot Tr_s(e^s_m) \cdot \\
&\qquad (\mathit{qEmpty} \cdot ((rv \not\approx \bot) \to \mathit{sendReply}(rv) \diamond \mathit{skip}) \cdot P_{s'}(a_1, \ldots, a_k, rv = \bot) + \\
&\qquad\; \mathit{qNonEmpty} \cdot P_{s'}(a_1, \ldots, a_k)) \\
&Tr(s, \varphi, e^r, \mathit{Illegal}) = \varphi \to Tr_r(e^r) \cdot \mathit{illegal} \cdot \delta
\end{aligned}$$

$$Tr_s(e) = \begin{cases} \mathit{outwardNotif}(e) & \text{if } e \in \mathit{Notif} \\ e \cdot \mathit{recReply}(\varnothing) & \text{if } e \text{ is a void call event} \\ e & \text{if } e \text{ is a valued call event} \end{cases}$$

$$Tr_r(e) = \begin{cases} e & \text{if } e \in \mathit{Call} \\ \mathit{readNotif}(e) & \text{if } e \in \mathit{Notif} \\ \mathit{recReply}(e) & \text{if } e \in \mathit{Reply} \end{cases}$$

In the definitions above, rv is a process parameter that stores the reply value that needs to be returned after the queue is emptied. A special value, $\perp$, indicates that no reply is due to be sent. Whereas all call events are translated into communicating actions, notifications and replies are translated into arguments of the actions *outwardNotif* and *readNotif*, and *sendReply* and *recReply* respectively. Checking whether the queue is empty or not happens through the communicating actions *qEmpty* and *qNonEmpty*. Lastly, *skip* is the empty process and δ is the deadlock process.

Example 3. Recall component A from Example 1. We give the translation of the rule cases 1, 2 and 4 according to Definition 2:

$$\begin{aligned} & A_{\mathit{Active}}(\mathit{enabled}{:}\mathit{Bool}, rv{:}\mathit{Reply}) = \\ & \quad (\mathit{enabled} \approx \mathit{false}) \rightarrow \mathit{A\_deactivate\_called} \cdot \mathit{invoke\_B\_Deactivate} \cdot \mathit{recReply}(\varnothing) \cdot \\ & \qquad \mathit{outwardNotif}(\mathit{A\_Deactivated}) \cdot \\ & \qquad (\mathit{qEmpty} \cdot \mathit{sendReply}(\varnothing) \cdot A_{\mathit{Inactive}}() + \mathit{qNonEmpty} \cdot A_{\mathit{Inactive}}(rv = \varnothing)) + \\ & \quad (\mathit{enabled} \approx \mathit{true}) \rightarrow \mathit{A\_deactivate\_called} \cdot \mathit{illegal} \cdot \delta + \\ & \quad \mathit{A\_Pause\_called} \cdot (\mathit{qEmpty} \cdot \mathit{sendReply}(\varnothing) \cdot A_{\mathit{Active}}(\mathit{enabled} = \mathit{false}) + \\ & \qquad \mathit{qNonEmpty} \cdot A_{\mathit{Active}}(\mathit{enabled} = \mathit{false}, rv = \varnothing)) + \ldots \end{aligned}$$

For the state *Active*, we have a process definition A_{Active}, which carries the state variable *enabled*. The different types of events each have a prefix or suffix to distinguish them: received call events have the suffix *called* and sent call events have the prefix *invoke*. After an illegal event, the process deadlocks (operator δ). □

3.2 Communication

In the existing single-component translation, a void reply is represented with the action *sendReply*($\varnothing$). Since the scope of this translation is very limited – it only translates one design model and several interface models from one component boundary (cf. Figure 1) – synchronization on this action can only take place in one way: between the design model and one of the interface models. However, in the multi-component setting, we have to explicitly enforce synchronization to happen between the proper components. Therefore, every occurrence of an action *sendReply* does not only have an argument for the type of the reply, but also two arguments to indicate the source and destination of the reply. In this way, only those components will synchronize on that action. The same approach is applied to notifications.

3.3 Manual translations

The automatic translation to mCRL2 cannot handle wrapper components, since their router is implemented in C++ instead of ASD. The behaviour of a wrapper component as a symmetric communication channel is essential to the behaviour of the complete system. Therefore, it is desirable that the wrapper components are also present in the translation of the complete system. We manually translated the router to mCRL2, because this is far more efficient than performing an automatic translation from C++.

There is another component, called *AsyncCall*, of which the behaviour is manually translated from C++ to mCRL2. The component *AsyncCall* can be requested by any component to send a response at some later time. This is a workaround such that ASD components can awaken themselves to finish residual duties. Internally, these requests are stored in a queue in *AsyncCall*. The queue can only contain one request per component and components also have the option to cancel their request.

We remark that the wrapper components are partially generic and the *AsycCall* component is completely generic. Therefore, they do not need to be implemented from scratch when verifying several systems. Ideally, the mCRL2 specification of the wrapper components can be generated by the same program that generates their C++ implementation.

3.4 Queues

As defined in the semantics of ASD, every component contains its own queue to store notifications. This implies that the complete mCRL2 specification will have a queue for every design model in the system. To ensure that the run-to-completion semantics is preserved, we add a lock to every queue. A queue is unlocked exactly when the corresponding component is active processing a call from a client or a modelling event. In this way a client can only process the content of a queue when it is active.

3.5 Framework

Not only the ASD components themselves, but also the outside world, which we will refer to as the *framework*, should behave according to certain constraints. For example, the framework cannot send out another call or modelling event when the system did not yet finish the previous task. To encode this, we add the following features to our translation:

- A new action *emptyQ* that can only be executed when all queues are empty; all queues synchronise on this action.
- An additional process *Thread* controls the sending of calls to the uppermost components and sending of modelling events by the foreign components. At the moment it sends a call or modelling event, it checks whether the queues are empty. It can only send another message after the previous one completed processing.

4 Case study

We perform a case study to investigate the feasibility of our approach and to determine its applicability to industrial-size systems. The case study is based on a real-life ASD system found in the model stack of our industrial partner. Components and events have been renamed for confidentiality reasons. Figure 4 shows the high-level structure of the system. The system consists of two loosely-coupled subsystems, called A and B. A and B have to cooperate to execute an action together, which we will call *Exec*. The clients of A and B independently decide whether they are ready to do so. Moreover, after a client has requested for *Exec* to be performed, it can repeal its decision by sending a *Cancel* message. Both sides synchronize using wrapper components located on the *cancel layer* and the *control layer*. The cancel layer consists of the components responsible for cancelling the execution of *Exec* and the control layer is responsible for performing *Exec*. The cancel and control layers, including their direct server components, consist of 14 components. The complete system consists of 26 components, which contain 5054 rule cases in total.

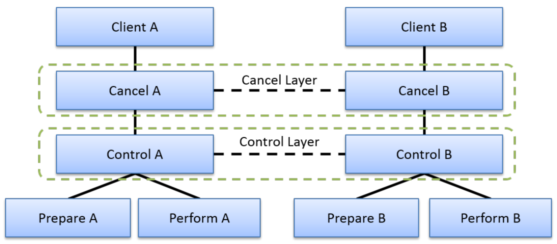


Fig. 4. The structure of the case study system. The dashed lines indicates bidirectional communication through wrapper components.

In the initial state, the clients of A and B can decide whether they want to perform *Exec* or cancel it. When both clients request to do *Exec* without sending a *Cancel* at some point, then this action is performed. If at least one client requests a cancellation before they both request *Exec*, then the action *Cancel* is performed. If a client asks for *Exec* to be cancelled after *Exec* has started, then it will not be cancelled. Both clients will receive a notification when an *Exec* or a *Cancel* has been completed. After a client requests *Exec*, it will immediately start preparing for *Exec* to happen. When *Exec* or *Cancel* has finished, the system should return to the initial state. We call the process between the first message and the performing of *Exec* or *Cancel* a *round*.

4.1 Subsystems

In our analysis, we consider four different variants of this system, to get a rough idea of the scalability. Firstly, we have the *layers subsystem*, which consists of the cancel layer, the control layer, the components directly below them and the wrapper components in between. We consider two variants of the layers subsystem: the regular implementation and an implementation where no *Cancel* request can be performed. Secondly, we have two variants of the complete system:

Table 1. Properties of the layers system written in modal µ-calculus

	Property	Formula
0	Initial state	$(\langle \mathit{A\_Request\_Exec}\rangle \mathit{true} \wedge \langle \mathit{A\_Request\_Cancel}\rangle \mathit{true} \wedge$ $\langle \mathit{B\_Request\_Exec}\rangle \mathit{true} \wedge \langle \mathit{B\_Request\_Cancel}\rangle \mathit{true})$
1	Cannot do *Exec* without two requests	$[\mathit{true}^*](\text{Initial state} \Rightarrow [(\overline{\mathit{A\_Request\_Exec}}^* + \overline{\mathit{B\_Request\_Exec}}^*).\mathit{Exec}]\mathit{false})$
2	Cannot request execution after a cancel request	$[\mathit{true}^*.\mathit{A\_Request\_Cancel}.\overline{(\mathit{outwardNotification}(\mathit{Cancelled}))}^*.\mathit{A\_Request\_Exec}]\mathit{false}$
3	Must perform execution after two execute requests	$[\mathit{true}^*](\text{Initial state} \Rightarrow [\mathit{A\_Request\_Exec}.\overline{(\mathit{A\_Request\_Cancel} + \mathit{B\_Request\_Cancel})}^*.\mathit{B\_Request\_Exec}]\mu X.([\overline{\mathit{Exec}}]X \wedge \langle \mathit{true}\rangle \mathit{true}))$
4	Raise cancel notification after two cancel requests	$[\mathit{true}^*.\mathit{A\_Request\_Cancel}.\overline{(\mathit{outwardNotification}(\mathit{Cancelled}))}^*.\mathit{B\_Request\_Cancel}]$ $\mu X.([\overline{\mathit{outwardNotification}(\mathit{Cancelled})}]X \wedge \langle \mathit{true}\rangle \mathit{true})$
5	After a cancel request, *Cancel* is performed	$[\mathit{true}^*](\text{Initial state} \Rightarrow [\mathit{A\_Request\_Cancel} + \mathit{B\_Request\_Cancel}]$ $\mu X.([\overline{\mathit{A\_Cancel}}]X \wedge \langle \mathit{true}\rangle \mathit{true}))$
6	Cannot make multiple *Exec* requests in a round	$[\mathit{true}^*.\mathit{A\_Request\_Exec}.\overline{(\mathit{outwardNotification}(\mathit{Exec\_finished}))}^*.\mathit{A\_Request\_Exec}]\mathit{false}$
7	No synchronization error during an execution	$[\mathit{true}^*](\text{Initial state} \Rightarrow [\mathit{A\_Request\_Exec}.\overline{(\mathit{B\_Request\_Exec})}^*.\mathit{B\_Request\_Exec}$ $.\overline{(\mathit{A\_Request\_Exec} + \mathit{B\_Request\_Exec})}^*.\mathit{Sync\_Error}(A)]\mathit{false})$
8	Synchronization error after requesting an execute too soon	$[\mathit{true}^*](((\langle \mathit{A\_Request\_Exec}\rangle \mathit{true} \wedge \langle \mathit{B\_Get\_Results}\rangle \mathit{true}) \Rightarrow$ $[(\overline{\mathit{B\_Get\_Results}})^*.\mathit{A\_Request\_Exec}]\mu X.([\overline{\mathit{Sync\_Error}(A)}]X \wedge \langle \mathit{true}\rangle \mathit{true}))$

one that does not allow errors to occur and one that does allow errors. We will call the former *good-weather behaviour* (GWB) and the latter *bad-weather behaviour* (BWB). The bad-weather behaviour system is rather rudimental, which means that clients can raise errors which are subsequently dealt with by the system to cause the least disturbance in the normal process operation.

4.2 Properties

After consulting the domain experts, we identified several system-wide properties that the system under study should adhere to. For the layers subsystem, we have eight properties, which are listed in Table 1. Furthermore, we have three properties that involve the complete system; they are listed in Table 2. Note that some properties are symmetric for both clients; in that case we only listed one of the two µ-calculus formulae. Many properties are concerned with behaviour from the moment that neither client has sent a message yet until the moment that *Exec* or *Cancel* is performed. Since we are dealing with a system that runs continuously, we should not only check what happens from the initial state, but in every round. Therefore, we formulated a property in the µ-calculus that expresses whether the system is at the start of a round (property 'Initial state' in Tables 1 and 2). This formula is used within other properties to check behaviour in all rounds. Most action names in these properties should be self-explanatory. The actions *outwardReply* and *outwardNotification* respectively represent a reply and a notification sent to one of the two clients. The inserting of a notification into the queue of a certain component is represented by the action *raiseNotification*.

Table 2. Properties of the full system written in modal µ-calculus

	Property	Formula
0	Initial state	$(\langle A\_Request\_Exec.\overline{(Protocol\_Error(A))}^{*}.outwardReply(VoidReply)\rangle true \wedge$ $\langle A\_Request\_Cancel.\overline{(Protocol\_Error(A))}^{*}.outwardReply(VoidReply)\rangle true \wedge$ $\langle B\_Request\_Exec.\overline{(Protocol\_Error(B))}^{*}.outwardReply(VoidReply)\rangle true \wedge$ $\langle B\_Request\_Cancel.\overline{(Protocol\_Error(B))}^{*}.outwardReply(VoidReply)\rangle true)$
1	Must perform execution after two execute requests	$[true^{*}]$(Initial state $\Rightarrow [A\_Request\_Exec.\overline{(A\_Request\_Cancel + B\_Request\_Cancel)}^{*}.$ $B\_Request\_Exec]\mu X.([\overline{Exec}]X \wedge \langle true\rangle true))$
2	Prepare steps are done before perform steps	$[true^{*}]$(Initial state $\Rightarrow [A\_Request\_Exec.\overline{(A\_Request\_Cancel + B\_Request\_Cancel)}^{*}.$ $B\_Request\_Exec.\overline{(raiseNotification(A\_Prepare\_Step\_Done))}^{*}.Exec]false)$
3	Perform steps are done before raising an execution notification	$[true^{*}]$(Initial state $\Rightarrow [A\_Request\_Exec.\overline{(A\_Request\_Cancel + B\_Request\_Cancel)}^{*}.$ $B\_Request\_Exec.\overline{(raiseNotification(A\_Perform\_Step\_Done))}^{*}$ $.outwardNotification(Exec\_finished)]false)$

In these properties, we use several common patterns. First, it is very common to write a property of the shape $[true^{*}]\varphi$, meaning that after any sequence of actions, φ has to hold. Building on that, the property $[true^{*}.a]false$ expresses that action a cannot occur anywhere and the property $[true^{*}.a.\bar{b}^{*}.c]false$ expresses that after every action a, an action b must happen before the action c happens.

A more complex, but important, pattern is $\mu X.([\bar{a}]X \wedge \langle true\rangle true)$, which means that action a unavoidably happens within a finite amount of steps. The intuition is as follows: as long as we do something other than a (subformula $[\bar{a}]$), we recurse through variable X. That may only happen finitely often, due to the least fixpoint (μX). Therefore, we must at some point end up in a state where actions other than a are not possible. This state cannot be a deadlock, since that is explicitly forbidden by $\langle true\rangle true$. Hence, we end up in a state where only a actions can be done, and at least one a is possible. This is the same as saying that ultimately, a must be done.

5 Results

In our experiments, we applied the workflow of Figure 5 to check each of the properties. First, the mCRL2 specification is normalised into a *linear process* (LPS), from which we generate the state space in the shape of a *labelled transition system* (LTS). This LTS is subsequently minimised under divergence-preserving branching bisimulation using the Groote-Jansen-Keiren-Wijs algorithm [9]. Combined with a µ-calculus formula, we construct a *Boolean equation system* (BES), which can be solved to obtain an answer to the model checking question. A benefit of using this particular workflow is that the state space does not need to be generated repeatedly for every property we want to check.

To run the experiments, we used a machine with multiple Xeon E5520 processors (56 cores in total), clocked at 2.27GHz and 935GB of memory. The mCRL2

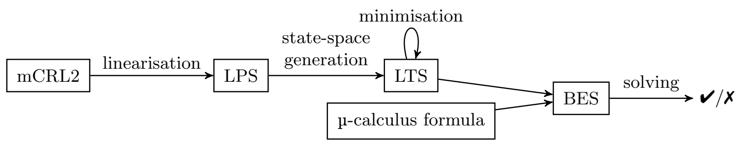


Fig. 5. The workflow used for model checking.

version we installed has Git commit hash `73241e378e`[1]. The mCRL2 analysis tools are all single threaded.

Table 3 gives an overview of the time required for state-space generation and the size of each of the state spaces. The time reported does not include the time required for bisimulation reduction, which is about 45 minutes for the BWB system. The full system under bad-weather behaviour is almost on the limit of what can be generated in a reasonable amount of time with 178.6 million states. At the same time, bisimulation reduction is very effective, and manages to bring the number of states back to 12.4 million.

Figure 6 shows a visualisation of the minimised state space of the layers subsystem. The initial state is at the top and every disk represents a (non-deterministic) choice. Initially, the system contains little branching behaviour (the red, yellow and green parts at the top). Only deeper in the state space, there is more choice to perform different actions (blue and purple parts at the bottom). The low amount of branching can be ascribed to the run-to-completion semantics of ASD.

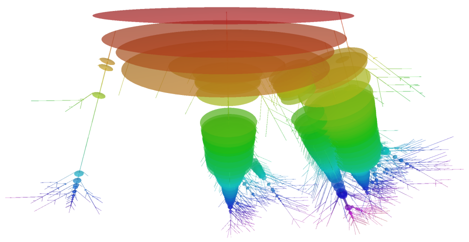


Fig. 6. Visual representation of the minimised state space of the layers subsystem.

Table 4 records for each of the four variants of the system the average time required to check one of the properties. All properties hold, except when checked

[1] Sources are available at https://github.com/mCRL2org/mCRL2

Table 3. Time required for state-space generation and number of states and transitions for the systems before and after divergence-preserving branching bisimulation minimisation.

					After bisim. red.		
System	time (s)	mem	#states	#transitions	#states	#transitions	%red
Layers no *Cancel*	7	28MB	9,107	9,472	7,085	7,422	22.2
Layers subsystem	46	63MB	109,608	114,310	55,361	58,338	49.5
Complete system GWB	14,713	1.7GB	17,179,798	19,098,495	3,787,974	4,298,103	75.0
Complete system BWB	154,397	14GB	178,603,107	196,784,882	12,451,325	14,879,416	93.0

Table 4. Average time spent to verify a single property on each of the four (sub)systems.

	time (s)			
	Layers no cancel	Layers System	Full System GWB	Full System BWB
`lts2pbes`	2.06	9.82	2,317	14,126
`pbessolve`	0.44	3.74	890	4,150
Total	2.50	13.56	3,207	18,276

on the bad-weather behaviour system, since that has not been fully implemented. The time required for the full system is significant: almost one hour for the good weather version and close to five hours per property for the bad weather system.

While running these large experiments, we observed that a lot of time and memory is spent on storing and loading intermediate files from disk. For a typical property of the BWB system, `lts2pbes` spends more than three quarters of the time on storing the PBES on disk. This problem could be by-passed by implementing an integrated tool that combines the functionality of `lts2pbes` and `pbessolve`. The amount of memory required to verify a property of the BWB system is roughly 180GB; this peak is also reached while writing the output in `lts2pbes`.

6 Related work

The successor of ASD is Dezyne[2], also developed by Verum. Similar to the ASD-mCRL2 translation, there is also a translation from Dezyne to mCRL2 [2]. This translation is also limited to single components, so it does not support verification of end-to-end properties.

mCRL2 has also been used in other studies to analyse systems with a very large state space. For example, in [1], the train control system ERTMS Hybrid Level 3 is analysed with mCRL2. They apply the same workflow as we do: minimise the transition system with bisimulation before checking any property. The largest state space they verified contains close to 34 million states.

Remenska *et al.* [15] also work with an automated translation. They convert the behaviour captured in UML2.0 sequence diagrams to mCRL2 specifications. Their technique is applied on DIRAC, the computing grid framework of CERN's LHCb experiment. Although the state-space is too large to generate completely, they do find a counter-example to the desired property with depth-first search.

The idea of generating code from formal models that have been checked with model checking is also applied in [7]. The authors present a tool called DLC, Distributed LNT Compiler, which can produce C code from an LNT specification. The generated code is suited for running om multiple machines concurrently, synchronization between the machines is achieved with a rendezvous protocol. LNT specifications can be analysed with the existing tools from the CADP toolset [8].

[2] See `https://www.verum.com/`, accessed 13-05-2019

7 Conclusion

We showed how to translate a component system implemented in ASD to mCRL2. This enables checking of end-to-end properties on ASD, which is important for mission-critical software. Furthermore, we demonstrated with a case-study that this approach is applicable to an actual industrial system. For the two variants of the layers subsystem, the time and space required to run the model checker is sufficiently small to enable interactive verification during development. This is due to the semantics of ASD that avoids a major state-space explosion. The results exceeded our own expectations and give us hope that model checking of complete systems can be applied more often in industrial settings. Based on these results, we aim to develop an environment in which all industrial controllers, newly developed in ASD, can be completely verified during their development process.

One of the challenges that needs to be tackled before wider adoption of this approach is possible, is the complexity of modal µ-calculus. Currently, specifying properties with µ-calculus requires expertise, and it is not uncommon for formulas to contain mistakes. A possible solution is to supply developers with natural-language templates in which they enter the correct action names. The corresponding formula will then be generated from the template [16].

References

1. Bartholomeus, M., Luttik, B., Willemse, T.A.C.: Modelling and Analysing ERTMS Hybrid Level 3 with the mCRL2 Toolset. In: FMICS 2018. LNCS, vol. 11119, pp. 98–114 (2018). https://doi.org/10.1007/978-3-642-15898-8
2. van Beusekom, R., Groote, J.F., Hoogendijk, P., Howe, R., Wesselink, W., Wieringa, R., Willemse, T.A.C.: Formalising the Dezyne Modelling Language in mCRL2. In: FMICS/AVoCS 2017. LNCS, vol. 10471, pp. 217–233. Springer (2017). https://doi.org/10.1007/978-3-319-67113-0_14
3. Broadfoot, G.H.: ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In: FM 2005. LNCS, vol. 3582, pp. 548–551 (2005). https://doi.org/10.1007/11526841_39
4. Broadfoot, G.H., Hopcroft, P.J.: Analytical software design. Tech. rep., Verum Consultants B.V. (2003)
5. Bunte, O., Groote, J.F., Keiren, J.J., Laveaux, M., Neele, T., de Vink, E.P., Wesselink, W., Wijs, A., Willemse, T.A.: The mCRL2 Toolset for Analysing Concurrent Systems: Improvements in Expressivity and Usability. In: TACAS 2019. LNCS, vol. 11428, pp. 21–39 (2019). https://doi.org/10.1007/978-3-030-17465-1_2
6. Cranen, S., Groote, J.F., Reniers, M.A.: A linear translation from CTL* to the first-order modal μ-calculus. Theor. Comput. Sci. **412**(28), 3129–3139 (2011). https://doi.org/10.1016/j.tcs.2011.02.034
7. Evrard, H., Lang, F.: Automatic distributed code generation from formal models of asynchronous processes interacting by multiway rendezvous. Journal of Logical and Algebraic Methods in Programming **88**, 121–153 (2017). https://doi.org/10.1016/j.jlamp.2016.09.002

8. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. STTT **15**(2), 89–107 (2013). https://doi.org/10.1007/978-3-540-73368-3_18
9. Groote, J.F., Jansen, D.N., Keiren, J.J.A., Wijs, A.J.: An O(m log n) Algorithm for Computing Stuttering Equivalence and Branching Bisimulation. ACM Trans. on Comput. Logic **18**(2) (2017). https://doi.org/10.1007/978-3-662-49674-9_40
10. Groote, J.F., Kouters, T.W.D.M., Osaiweran, A.: Specification guidelines to avoid the state space explosion problem. Softw. Test., Verif. Reliab. **25**(1), 4–33 (2015). https://doi.org/10.1002/stvr.1536
11. Groote, J.F., Mousavi, M.R.: Modeling and Analysis of Communicating Systems. MIT Press (2014)
12. Jonk, R.: The semantics of ALIAS defined in mCRL2. Master's thesis, Eindhoven University of Technology (2016)
13. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3), 333–354 (1982). https://doi.org/10.1007/BFb0012782
14. Osaiweran, A., Schuts, M., Hooman, J., Groote, J.F., van Rijnsoever, B.J.: Evaluating the effect of a lightweight formal technique in industry. STTT **18**(1), 93–108 (2016). https://doi.org/10.1007/s10009-015-0374-1
15. Remenska, D., Templon, J., Willemse, T.A.C., Homburg, P., Verstoep, K., Casajus, A., Bal, H.: From UML to process algebra and back: An automated approach to model-checking software design artifacts of concurrent systems. In: NFM 2013. LNCS, vol. 7871, pp. 244–260 (2013). https://doi.org/10.1007/978-3-642-38088-4_17
16. Remenska, D., Willemse, T.A.C., Templon, J., Verstoep, K., Bal, H.E.: Property specification made easy: Harnessing the power of model checking in UML designs. In: FORTE 2014. LNCS, vol. 8461, pp. 17–32 (2014). https://doi.org/10.1007/978-3-662-43613-4_2
17. Roscoe, A.W.: On the expressive power of CSP refinement. Formal Aspects of Computing **17**(2), 93–112 (2005). https://doi.org/10.1007/s00165-005-0065-x