Eindhoven University of Technology

MASTER

Verification of ASD multi-component systems in mCRL2

Rol, M.H.

Award date:
2018

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 24. May. 2025

Verification of ASD multi-component systems in mCRL2

M. H. Rol
Eindhoven University of Technology; ASML Netherlands B.V.

Abstract

Analytical Software Design (ASD) assists the creation of software systems. Systems designed in ASD are composed of multiple components in order to divide the complexity of the whole system over them. The verification of system properties and requirements is limited to the scope of single components, disallowing the verification of end-to-end properties. We present an approach for the verification of end-to-end properties on multi-component systems. This provides a higher confidence in the functional correctness and reliability. A system based on a real-life ASD model serves as use-case for the proposed approach. Results show that verification of multi-component systems can be done through mCRL2, but scalability issues are observed as larger systems are verified.

1. Introduction

In the world of today, devices and machines are part of high tech companies. It is expected that these systems operate correctly, thus the verification of systems is important. Verification assists in finding defects early. The earlier defects are detected in the design of a system, the less it costs to fix them.

ASML is focused on the development and production of systems for the semiconductor industry. These systems are responsible for the production of integrated circuits. A part of the software for these systems is designed through Analytical Software Design (ASD) [1], where a system is composed of multiple smaller parts known as components. Each component can individually be verified against correctness properties such as being free from range errors, deadlocks and livelocks. By restricting the input syntax in favor of compositional verification, correctness properties such as global deadlock freedom can be achieved.

System modeling and the verification of system properties and requirement can be done through the toolset mCRL2. An example requirement is that when a button is pressed, the light in the room is turned on. The scope of a property can range from as small as a single component to as large as the whole system. Single component properties are *local properties* and properties that cover multiple components are *multi-component*. When properties cover the whole system they are known as *end-to-end properties*. It is desirable to verify end-to-end properties in order to further increase the confidence in the functional correctness and reliability of the system as a whole. A per component translation from ASD to mCRL2 allows the verification of local properties, but is limited to local properties only. This paper proposes a novel approach for the verification of ASD multi-component systems using mCRL2.

To provide support for multi-component verification in mCRL2, a per component translation from ASD to mCRL2 is used to gather single-component mCRL2 models of a multi-component ASD system. Single-component models are combined together into one multi-component mCRL2 model. The verification of the multi-component model against multi-component properties and requirements is evaluated.

ASD and mCRL2, as well as their use, are summarized in section 2. Section 3 covers the procedure to convert multi-component systems to multi-component mCRL2 models. Section 4 contains the case study of a real-life ASD model, and the multi-component verification and results of this system are contained in section 5. Section 6 covers related work. Finally, section 7 draws the conclusion of the research.

2. Background information

This section contains a brief summary of the tools used.

2.1. Analytical Software Design

Analytical Software Design (ASD), developed by Verum [1], makes it possible to create systems from mathematically verified components. The ASD:Suite, a software design platform, allows one to verify components and generate fully executable source code from

these components. 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 behavior of a component. It forms an abstraction of the component and shows what a component does at a given time but not how. The design model specifies the inner working of a component and how it interacts with other components. The implementation of the design model complies with the specification of the interface model.

- *Foreign component*. A foreign component is not developed using ASD. It consists of only an interface model, used for verification and code generation for other components that use it.

2.1.1. Communication. If a component A uses another component B as a service, then B is a *server* of A and A is a *client* of B. The design model of a component is used to communicate with the interface models of server components, and the design model must comply with the specifications of the interface models of the servers. Visually, servers are positioned below their clients. If a component does not have a component as client, then the client is the outside world. The outside world will be referred to as the *framework*. Since foreign components do not contain design models, foreign components cannot make use of servers.

Components communicate with exactly one client at a time. If a server has multiple clients, then each client makes use of a unique instance of that server. If a server is shared between multiple clients, then that server is a *shared component* and all clients use the same instance of that server.

Components communicate with each other by sending and receiving information. Components are only aware of information that crosses their boundary. The *component boundary* of a component is formed over the models of the component and the interface models of used servers. Figure 1 shows an example system composed of 3 components. In this example, the component boundary of A is limited to the interface models of servers B and C in addition to the interface and design model of component A. Component A is only aware of information sent to the framework and to servers B and C, as well as information received from them.

Information that crosses the component boundary to and from servers and clients are *events*. Components send events as *actions* and receive events as *triggers*. The different types of events are as follow:

- A *call event* sent to a server.

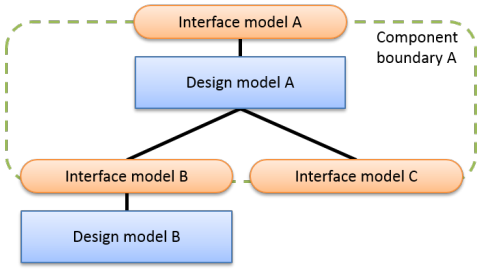

Figure 1. An example system composed of 2 standard components A and B and a foreign component C.

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

Table 1. An example process to deactivate a component A.

- A *reply event* received from a server. This marks the end of the process of a call event that has been sent earlier to the server.

- A *notification event* sent to a client. Notifications that are sent by servers are stored in the *notification queue* of the client. The notification queue is an intermediate for notifications between the server and the client. All standard components make use of a notification queue.

2.1.2. Component process. ASD models are defined in the *Sequence-Based Specification* form. Table 1 provides an example specification of component A, focused on deactivating the component.
A component contains *states*, which define the state of a component. A component can only be in exactly one state at a time. When a component receives a trigger, the trigger is processed by sending actions to servers or clients. Which actions to send is depends on the following:

- The state of the component.

- The trigger received.

- Guards, also known as *state variables*. These are used to further control the flow of process within the state of a component.

Using Table 1, component A processes the actions in case 1 if it receives a *Deactivate* trigger while in the *Active* state and the state variable *Enabled* equals true. If *Enabled* does not equal true, it will proceed by processing case 2 due the *Otherwise* guard. If the action *Illegal* is to be processed, the system will deadlock. This states that component A is not allowed to receive the *Deactivate* call event while *Enabled* does not equal true in the *Active* state.

A single active thread is used to process triggers and actions of a component. When a component receives a call event, this thread is temporarily passed with it to process the call event at this component. Actions, as the result of the trigger, are processed sequentially. If such an action is a call event sent to a server, then the thread is passed along with it. The execution of the calling component is halted until the thread returns via a reply event for the call event.
Replies are of either type *void* or *valued*. In the case of valued, a value is passed along with the reply. This value is used as trigger to determine the next actions to process. In the case of void, no value is sent. While a component awaits a reply event, it blocks any other call or notification event. Using case 1 of Table 1, component A sends a *Deactivate* call event to server component B. Component A now has to wait until a reply event from component B is sent back before A can process the next action.
If a processed action is a notification event, it sends the notification to the notification queue of the client. The framework does not make use of a notification queue, and notifications sent to the framework do not need to be processed. Using case 1 of Table 1, component A will send a *Deactivated* notification to the queue of the client after having received the reply event for the *Deactivate* call event to server B.

Once a component has finished processing all actions of a trigger, excluding reply events, it will update state variables and make a transition to a new state. Using case 1 of Table 1, component A updates *Enabled* to false and transits to the *Inactive* state after having send the *Deactivated* notification.

Finally, the component is about to finish processing a call event by sending a reply event. However, it will first check the notification queue for notifications sent by servers. If a notification is found, the notification is processed and actions, as a result of this notification, are processed as well. This is similar to receiving and processing a call event. This step continues until the notification queue is empty. When the queue is empty, a reply event, for the original call event the component received, is sent back to the client, together with the thread. When a component is capable to fully process all triggers it receives, it preserves *run-to-completion* semantics. All components must preserve run-to-completion.
Using case 1 of Table 1, component A is about to finish processing the *Deactive* call event after the transition to the *Inactive* state. Once the notification queue of component A is found empty, A will send a reply event of type void to the client. A can also process *Pause* or *Resume* call events through case 3 and 4, which would only change *Enabled* to false and true respectively.

It is possible for interface models to send notification events to their clients spontaneously. These represent hidden internal behavior and is captured by the use of *modeling events*. Modeling events are special events triggered by the framework to model internal behavior. In order to preserve run-to-completion, a special *process callback request* is sent by the server to notify clients of notifications in their queue. This allows the thread to be passed on to the clients in order to process the notification events. Process callback requests are internal actions and are not captured by the Sequence-Based Specification. Additionally, they are limited to modeling events only and are not used during the process of a call event.
Modeling events are limited to interface models. During the process of a system, standard components do not make use of modeling events as the internal behavior is already captured by their design models. In this process, only foreign components make use of modeling events.

In the *single-threaded execution model*, there is only one active thread and thus only one component can process at a time. In the *multi-threaded execution model*, each component contains a *deferred procedure call* (DPC) server thread. This thread is responsible for processing notifications in the queue. The single thread in the single-threaded execution model is no longer required to process notifications, but it is still used to process call and reply events. This allows the process of notifications to interleave with the process of call events. In addition this makes the use of process callback requests, as a result of modeling events, redundant.
For the remainder of this paper, models are limited to the single-threaded execution model.

2.1.3. Wrapper components. While shared components have multiple clients, they are unable to communicate with more than one client at a time as ASD is unable to support such multi-client components in the single-threaded execution model. Instead, these components make use of a handwritten *wrapper component*. Figure 2 shows the structure of a wrapper component. For each client, an interface model is provided.

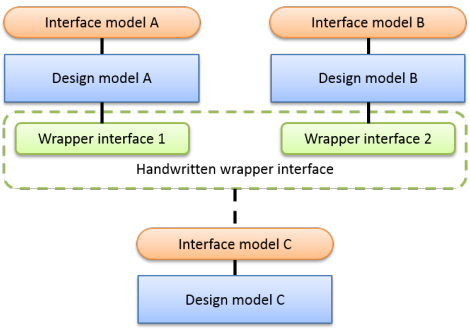

Figure 2. The structure of a wrapper component.

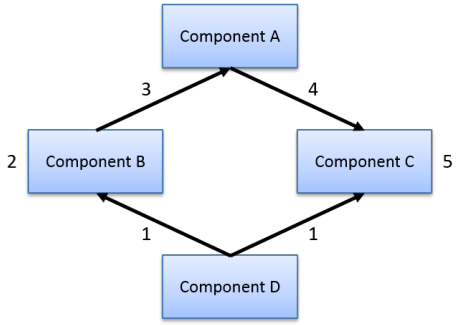

Figure 3. An example system that could end in an undetected deadlock.

The combined behavior of the provided interfaces must be exactly equal to the behavior of the required interface of the multi-client component. Handwritten code is used to code the provided interfaces as the interface model for the multi-client component. Components will be able to make use of this wrapper component as the multi-client component instead.

Wrapper components are to be handled with care. Multi-client events could cause undetected deadlocks to arise as deadlock detection is limited to a single component. A wrapper component is capable to send notifications to all clients as a result of a single modeling event. How to process these notifications through process callback requests is not defined. We assume that the order of notifications processed is chosen arbitrarily such that all cases are captured. However, if as a result of one notification in a client the state of another client alters while the queue holds a notification, then that client could be in a state that the notification is not allowed to be processed. This would results in the occurrence of a deadlock.

Figure 3 shows an example of a possible deadlock, with the steps taken as follow:

1. As a result of a single modeling event, the multi-client component D sends a notification to both component B and C.

2. Since both B and C have a notification in their queue, the process callback request is given to one of them at random. For this case, the callback is sent to B.

3. B processes the notification and as a result sends a notification to component A.

4. As a result of the notification, component A sends a call event to component C. C processes this call event, transits to a new state and replies back to A. No further actions are done.

5. As all the actions from the notification at B have been processed, a callback is now sent to C. However, due step 4, C could now be in a state that the process of the notification is found *Illegal*. The system ends in a deadlock at component C.

```
proc Process_A =
        a . b . (c + d) . Process_A;
```

Listing 1. An example process *Process_A*.

2.2. mCRL2

mCRL2 is a formal specification language [2]. The toolset of mCRL2 [3] allows one to model, validate and verify systems against specifications and properties. Additionally, the toolset supports the generation of state-space models to visualize the behavior of the system, which can be analyzed and explored further.

2.2.1. Process. A system described in mCRL2 is composed of processes. A process contains events of some kind, represented as actions, and a process describes the behavior of a system by the actions that are executed. Listing 1 shows an example process defined in mCRL2.

A *basic action* is a single action that is *atomic*, implying that it cannot be executed with other actions simultaneously. A *multi-action* $a_1|...|a_n$ implies that the actions a_1 to a_n are executed simultaneously. *Process_A* in Listing 1 is composed of basic actions only.

The sequential composition operator . puts two actions in sequence. That is, $a.b$ implies that after the a action, the b action follows.

The composition operator + allows the system to make an arbitrary choice between actions. That is, $a+b$ implies that either the a action or the b action will be executed.

The parallel operator || allows multiple processes to be executed in parallel. That is, $A||B$ implies that the actions of A and B interleave with each other as well as can be executed together as multi-actions. Thus the process $a||b$, with actions a and b, equals $(a.b)+(b.a)+(a|b)$.

Multi-actions are important as they can be used to provide synchronization and communication between parallel processes. The allow operator *allow(v, p)* implies that processes p are only allowed to perform the actions specified in v. By only allowing a multi-action to execute, parallel processes have to wait on each other until they are all capable to execute the allowed multi-action together.
The communication operator *comm(c, p)* implies which multi-actions in processes p are renamed to a single action. c holds the set of renames in the form
$a|b$ -> c, implying that the multi-action $a|b$ communicates together to action c.
The hide operator *hide(c, p)* implies which multi-actions c in processes p are made hidden. Hidden actions are written as *tau* and cannot be observed.

Listing 2 shows an example system defined in mCRL2. In this system, the basic actions a, b, c, d and e, as well as processes *Process_A* and *Process_B* are constructed. *Process_A* and *Process_B* are initialized in parallel and a multi-action is used to synchronize both processes. The allow operator defines that only actions a, b and e are allowed, while the communication operator defines that the multi-action $c|d$ communicates to action e. Since the basic actions c and d are not allowed, both processes would need to synchronize after the a and b action such that the allowed multi-action $c|d$, renamed as e, can be executed.

Finally, actions and processes can carry data parameters. Data parameters are used to help control the flow of process within the processes.

An mCRL2 process specification is used to generate a *Linear Process Specification* (LPS), which acts as a universal intermediate format. Such a specification has the definition of a single process in a basic structure. An LPS is further used to generate a labeled transition system.

2.2.2. Labeled Transition System. A *Labeled Transition System* (LTS) is a system that contains a set of states and transitions, known as the *state space*. The system initializes in one state known as the initial state, and transits to other states by the use of the transitions. The LTS describes all the states a system can be in as well as the transitions it takes to reach each state. An LTS can be used to generate a parameterised boolean equation system.

```
act a, b, c, d, e;

proc Process_A =
        a . c . Process_A;

proc Process_B =
        b . d . Process_B;

init allow({a, b, e},
     comm({c|d -> e},
     Process_A || Process_B));
```

Listing 2. An example system where parallel processes *Process_A* and *Process_B* synchronize with the multi-action $c|d$, renamed as action e

2.2.3. Parameterised Boolean Equation Systems. *Parameterised Boolean Equation Systems* (PBES) are used to encode model checking problems. They are generated from the LPS or LTS, together with a state formula *in modal μ-calculus* ([2] Chapter 6) describing the property to verify. Modal μ-calculus is briefly summarized as follows:

- Let α and β be sets of actions. If α equals true it is the set of all actions and if α equals false it is the empty set. The intersection and union of set of actions are represented by $\alpha \cap \beta$ and $\alpha \cup \beta$ respectively, and the notation $\overline{\alpha}$ represent the complement of the set of actions α with respect to the set of all actions.

- Let α and β be regular formulas that describes sequences of actions. The concatenation of sequences in α to sequences in β is represented by α . β. The union of the sequences in both α and β is represented by $\alpha + \beta$. Finally, α^* represents zero or more repetitions of the sequences in α, and α^+ represent one or more repetitions.

- Let α be a set of actions. The box modality $[\alpha]\phi$ is valid when for any sequence satisfying α, ϕ holds after executing that sequence. The diamond modality $\langle\alpha\rangle\phi$ is valid when a sequence α can be executed such that ϕ holds after executing that sequence. $\mu X.\phi$ is the minimal fixpoint. This is valid for the smallest set of states that can satisfy ϕ. Finally, $\phi \wedge \phi$, $\phi \vee \phi$ and $\phi \Rightarrow \phi$ represent the logical conjunction, disjunction and implication.

- As examples, $[true^*]\langle true\rangle true$ implies that after any sequence of actions, an action can be executed.

This equals deadlock-freedom.
$\mu X.(([true]X \wedge \langle true \rangle true) \vee \phi)$ implies that ϕ will eventually hold after any sequence of actions, excluding sequences ending in a deadlock.

3. Procedure

ASML develops tools and toolchains to specify and analyze large systems of systems. These tools cover various aspects including, but not limited to, semi-automatic model inference (for legacy software) [4], simulation and test artifact generation [5], (timed) system verification [6] and infra-structure binding generation, and (formal) supervisory controller synthesis [7] [8] [9]. Among others, these tools also include a per component translation from ASD to mCRL2 [10].

The conversion of a single component is used as a basis. The approach is to combine single-component mCRL2 models into one model specifying the multi-component model as a whole. This requires modifications to be made to each mCRL2 model individually to prepare them for the combination. This results in *pre-combined mCRL2 models*. Pre-combined mCRL2 models are combined together into one model.
Once all models have been combined, the resulting multi-component model has been constructed and can be verified against end-to-end properties of the system.

3.1. Conversion from ASD to mCRL2

Conversion of a standard ASD component from ASD to mCRL2 is done by taking all models in the component boundary as input. This is converted as a realization into a single mCRL2 model that translates all components into mCRL2 processes. Since the design model is an implementation of the interface model, the specification of the interface model of the component is not included.
Foreign components do not need to be converted as their specifications are part of other components as servers. Additionally, the specification of all servers are treated as foreign components as the design models of the servers are not included.

Figure 4 shows the structure of component A and component B as models in mCRL2, used as example for the remainder of this section to clarify the steps performed in combining them. Model B is a server of A, but both are treated as separate models before combination.

Every state of a component is translated into an mCRL2 process and every event is translated into an mCRL2 action. Since a component can only be in a

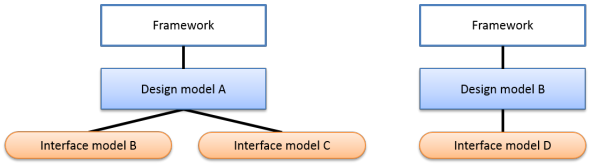

Figure 4. The structure of two separate components A and B modeled in mCRL2. Component B is a server of A.

single state at a time, only a single process per component is enabled. In each process, call and notifications events, translated as actions, can be performed by executing them. This corresponds to triggers in the ASD models. The resulting actions that are processed depends on data parameters and the enabled process of the component, similar to state variables and states in ASD. Reply values and notifications are translated to data parameters. This allows actions to carry them and processes to use them.

The framework is not included in the mCRL2 model as events send to and from the framework are captured by the design model of the component. Additionally, the notification queue is handled by a separate process, responsible for receiving notifications from servers as well as sending notifications to the components. The process of the queue holds the notifications received in a list and notifications are sent to the component in *First In First Out* (FIFO) order. Since the process of the queue holds the information of the queue, it can be used to check whether the queue is empty or not. This is required for run-to-completion as all notifications found in the queue must be processed. Communication, as multi-actions, are used to perform such checks. If the queue is empty, the renamed multi-action *qEmpty* can be executed. If the queue is not empty, the renamed multi-action *qNonEmpty* can be executed.

The use of multi-actions in mCRL2 allows the preservation of the single-threaded execution model. Since only a single component is allowed to process at a time, communication through multi-actions is used to pass thread control between components. Figure 5 provides an overview of the available communications between the framework, a component and a server. Model A, using component A and server B, is used as example, while server C is excluded. Furthermore, *Activate* is used as call event, *Activated* as notification and all reply events are of type void. The communications are marked as follows:

1. A call event from the framework to a component. There is no process used for the framework and thus no synchronization required.

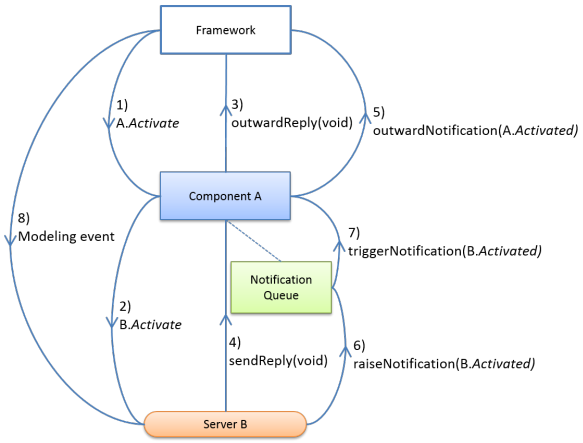

Figure 5. An overview of the communications between the framework, component A and component B.

2. A call event from a component to another component. These call events are identified by *invoke* and *invoked* actions for sending and receiving respectively, renamed together to the call event.

3. A reply event from a component to the framework. No synchronization is required and reply events to the framework are identified by *outwardReply*.

4. A reply event from a component to another component. These reply events are identified by *writeReply* and *readReply* actions for sending and receiving respectively, renamed together to *sendReply*.

5. A notification event from a component to the framework. No synchronization is required and notification events to the framework are identified by *outwardNotification*.

6. A notification event from a component to the queue of another component. These notification events are identified by *pushNotification* and *receiveNotification*actions for sending and receiving respectively, renamed together to *raiseNotification*.

7. A notification event from the queue to the component. These notification events are identified by *sendNotification* and *readNotification* actions for sending and receiving respectively, renamed together to *triggerNotification*.

8. A modeling event from the framework to a foreign component. The foreign component captures the reception of a modeling event, and thus no synchronization is required.

The framework would need to be blocked from sending call or modeling events while an earlier event from the framework is being processed. This would otherwise result in multiple components that are processing simultaneously, which is not allowed. This is solved by introducing a lock on the notification queue. Only when the notification queue is unlocked, and the queue is empty in compliance with the semantics of run-to-completion, the framework is allowed to send call and modeling events. The renamed multi-actions *lockQ* and *unlockQ* are used to lock and unlock the queue respectively.

Components, as data parameter, are used to indicate which component has locked the queue and is in thread control. A special value *NONE* is included, which indicates that the queue is not locked by any component at that time.
When a call event from the framework has been sent (communication 1 in Figure 5), the called component will lock its own queue in synchronization with the call event. After sending the reply event back (communication 3 in Figure 5), the queue is unlocked.
In case of a modeling event (communication 8 in Figure 5), the queue is locked by the server component in synchronization with the modeling event and is unlocked after having processed the modeling event. Since the queue is locked by a server component, notifications sent by the server are not processed until the queue has been unlocked. The process of the queue will always follow after unlocking the queue as the framework is blocked from sending call or modeling events while the queue is not empty. This also makes an explicit action for the process callback request redundant. The queue is locked by the component of the queue when processing notifications (communication 7 in Figure 5) and is unlocked after having processed all notifications.

3.2. Conversion from mCRL2 to pre-combined mCRL2

An mCRL2 model is not ready yet to be used for combination. There is no support for thread control between multiple components and communications between them are not set either. To support these, changes are made on the mCRL2 models to convert them into pre-combined mCRL2 models in preparation for the combination. Changes are made such that the pre-combined mCRL2 model is still functionally correct for the single component.
The choice to change the models before combination is made such that combining pre-combined mCRL2 models only requires the union of their specification. Further changes required after the combination are ex-

plained in section 3.3.

3.2.1. Thread control. In a single component model, the queue lock is used to preserve thread control between the component and servers. In a multi-component model, there are several components, each having their own queue as well as servers. Queue locks alone are no longer sufficient to preserve thread control as the process for a component does not necessarily have to depend on the state of the queue of another component.

In order to solve this, a new process is introduced for the thread of the single-threaded execution model. This thread can be locked and unlocked by a component through the renamed multi-actions *lock_thread* and *unlock_thread* respectively.
The thread is locked during the process of a call or modeling event from the framework, and is unlocked after having processed the call or modeling event from the framework. Locking the thread is done in synchronization with the call or modeling event (communication 1 and 8 in Figure 5). Unlocking of the thread is done as a basic action after sending the reply for the call event (communication 3 in Figure 5) or after processing the modeling event.

If a notification is to be processed (just before communication 7 in Figure 5), the thread is temporarily unlocked. This is done to be consistent to modeling events, where the thread is unlocked after having processed the modeling event. The thread would need to be locked during the process of a notification, hence locking the thread is done in synchronization with processing a notification (communication 7 in Figure 5).
When a notification queue has been fully processed as a result of a modeling event, and no queue contains any notifications, the thread is unlocked.

In a single-component model, the framework is blocked when attempting to send a call or modeling event while the notification queue is not empty. This needs to be expanded to cover all queues in a multi-component model. To do this, a new empty-queue-check multi-action, renamed as *emptyQ*, is added. This multi-action can only be executed if all queues are empty. Only if all queues are empty, the framework is allowed to send a call or modeling event. Thus executing *emptyQ* is done in synchronization with the call or modeling event (communication 1 and 8 in Figure 5).

Next, the queue of a component is unlocked after having sent a reply to the client. However, since the reply is sent to the client before unlocking the queue, the client is allowed to process. This causes multiple components to process at the same time, and is not allowed. To prevent this, sending of a reply is done in synchronization with unlocking the queue. This only counts for replies sent to the framework as servers do not make use of a queue (communication 3 in Figure 5).

Finally, one last issue is present with the process of notifications. If at some point multiple queues contain a notification (excluding the wrapper problem described in section 2.1.3), then there is no indication which queue is allowed to process their notifications. The wrong component could overtake thread control at this point.
To preserve thread control here, a new process for a queue-lock-thread is introduced. A component can only process notifications if the queue-lock-thread is not locked by another component. Locking and unlocking is done using the multi-actions *queue_lock_thread* and *queue_unlock_thread* respectively.
Whenever a component with thread control finds a notification in the queue, it will lock the queue-lock-thread to prevent any other component from processing their queue. Since a notification is to be processed, the thread will be unlocked as well. Unlocking of the thread is done in synchronization with locking the queue-lock-thread.
Once the notification is being processed (through communication 7 in Figure 5), it is safe to unlock the queue-lock-thread as the thread is locked, preventing any other component to overtake thread control at this point. Thus unlocking the queue-lock-thread is done in synchronization with processing a notification.

3.2.2. Communication. With multi-component models, there are more components that can communicate with each other. While thread control helps preserve the flow of process in the system, it is important to keep communication restricted between the proper clients and servers to avoid incorrect communications. For this, changes in events and processes have to be made.

For call events sent to shared servers (communication 2 in Figure 5), the client component is passed along as data parameter, This allows the server to know which component it is communicating with and can reply accordingly. Reply events (communication 4 in Figure 5) are changed similarly, but use two data parameters instead. One to indicate the source component this reply came from and the other to indicate the target client component this reply is sent to. This ensures the reply is restricted between those two components.

Next, if a non-shared server is to be used by multiple clients, then that server would need to be made unique for each client.

Finally, the queue of the component would need to be made unique for the component such that notifications and the process of the notifications are restricted

to that component only.

3.2.3. Overview. To provide an overview of the changes, the communications of Figure 5 have changed as follows:

1. Call events from the framework are done in synchronization with locking the queue (*lockQ*), locking the thread (*lock_thread*) and checking that all queues are empty (*emptyQ*).
2. Call events from a component to a shared server have the client passed along as data parameter.
3. Reply events to the framework are done in synchronization with unlocking the queue of the component (*unlockQ*). Unlocking the thread (*unlock_thread*) follows after the multi-action.
4. Reply events from a shared server have the client and server component passed along as data parameters.

7 Just before processing a notification, the thread is unlocked (*unlock_thread*). If the component processing the notification is in thread control, then this is done in synchronization with locking the queue-lock-thread (*queue_lock_thread*). Processing a notification is done in synchronization with locking the thread (*lock_thread*) and unlocking the queue-lock-thread (*queue_lock_thread*). The queue is automatically locked by the component of the queue when processing a notification.

8 Modeling events are done in synchronization with locking the queue (*lockQ*), locking the thread (*lock_thread*) and checking that all queues are empty (*emptyQ*). After processing a modeling event, before processing notification events, the queue is unlocked (*unlockQ*), followed by unlocking the thread (*unlock_thread*).

3.3. Combining mCRL2

Once pre-combined mCRL2 models have been constructed, they can be combined into a multi-component model. When models are combined, the specification of the resulting multi-component model is the union of them. Additionally, several changes are made to ensure proper multi-component behavior in the single-threaded execution model.

The major change of combining models is the replacement of the interface model of a server component by the design model. If a model is a server, the specification of the interface model would need to be replaced with the specification of the design model. In mCRL2, this is done by replacing the processes. Figure 6 shows the results of the replacement.

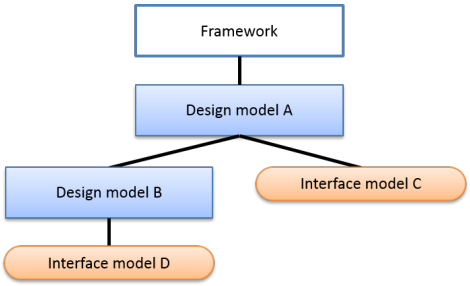

Figure 6. The results of combining the separate models A and B from Figure 4. The specification of the interface model of B in model A is replaced by the specification of the design model of B.

Furthermore there are changes in the process of the server model. As a server, it can no longer receive call or modeling events from the framework. Call events received from the framework are modified into call events received from the client component (communication 1 turned to 2 in Figure 5). This includes the removal of the thread lock *lockQ* and empty-queue-check *emptyQ* actions in the multi-action of the call event. The call event is changed to an *invoked* action to properly communicate with the client model.
Finally, all reply and notifications events to the framework are changed to reply and notifications events to the client component (communication 3 and 5 turned to 4 and 6 respectively in Figure 5). That is, *outwardReply* to *writeReply* and *outwardNotification* to *pushNotification*. Unlocking of the thread process through *unlock_thread*, that originally occurred after sending a reply to the framework, is removed as thread control is now passed back via *sendReply*.

4. Case study

The case study is based on a real-life ASD model found in the TWINSCAN model stack. Components and events are renamed for confidentiality reasons. The structure is preserved such that the verification metrics represent a real-life industrial size case study. After consulting the domain experts, system properties of this use case have been provided and are given in Section 5.

Figure 7 shows an example of the structure of the system. The system consists of two sides, A and B, that are loosely coupled. The client of A and the client of B can independently request to execute an action a and b respectively on their side, but also to cancel the request to execute the actions a and b. Both sides synchronize

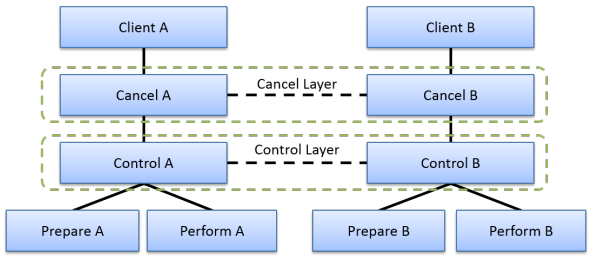

Figure 7. The structure of the case study system.

using wrapper components located on the cancel layer and the control layer. The cancel layer consists of the components responsible for canceling the execution of action a and b and the control layer consist of components responsible for executing actions a and b. The cancel and control layers, including server components, consist of 14 components. The full system consists of 26 components.

4.1. Process

In the initialized state, the clients of A and B can decide whether to execute their action or to cancel the request to execute their actions. When both clients request to execute without a cancel request, then both actions a and b are executed. If at least one client requests a cancel before both request an execution, then a cancel is done. Cancel requests can still be made when actions a and b are being executed, but the process of action a and b are not canceled.
The clients of both sides will raise a notification when an execution or a cancel has been successfully done or not. If an execution has been done, the results of the executions are gathered afterwards. Notifications are raised in response to cancel requests even when actions a and b have been executed.

When the client of a side requests to execute, it initiates that side to prepare the execution through prepare steps. These are initiated even if the other client has not done an execution request yet. After completing the prepare steps, the execution can be done through perform steps. Perform steps can only be done if both action a and b have been requested to execute and both clients have prepared the execution. The prepare steps are canceled if a cancel is to be done.

Once an execution or a cancel has been fully executed and the appropriate notification has been raised at the client of a side, the client of that side is ready to request a new execution or cancel. The system as a whole returns to the initialized state once both clients can request an execution or a cancel again. The period to request and perform an execution or cancel, thus starting in an initialized state and ending in an initialized state after having performed the requested action, is referred to as a *round*. A round consists of performing either an execution of actions a and b or a cancel.

4.2. Procedure case additional changes

There are several additional changes in the procedure while converting and combining models for this system.

4.2.1. Conversion from mCRL2 to pre-combined mCRL2. The system contains a server component responsible for sending asynchronous notifications to clients that have requested one through a call event. A modeling event is used to have this notification sent asynchronously. Clients can also cancel their request through another call event.
Requests received are to be processed in FIFO order. Requests received are shared to all instances of the component, thus the component would need to contain a queue to keep track which clients have made a request. This queue has to be added manually in mCRL2 as this specification is not present in ASD. A client can only be queued once and cannot cancel their request if they are not in the queue. This would otherwise result in *Illegal* actions.

4.2.2. Combining mCRL2. The wrapper component on the cancel layer can communicate with both side A and B simultaneously. This makes it vulnerable for the problem described in section 2.1.3 as the wrapper component can send a notification to both sides as result of a single modeling event. However, step 4 does not take place, preventing the problem to occur and makes the wrapper component safe to be used.

With multiple queues containing notifications as the result of a single modeling event, the processing order of these notifications must be preserved. Only once the notification at one side has been fully processed the notification on the other side is allowed to process. To tackle this in mCRL2, the wrapper component only sends one notification to an arbitrary side. The wrapper component halts and wait for all resulting actions of this notification to be processed before sending the second notification. This allows the process of one notification at a time.
Since the thread is unlocked after processing a notification, the framework must be prevented from overtaking thread control as the wrapper component still needs to send the second notification. For this, a synchronization lock process is added. This lock is to prevent any call or modeling event from the framework to occur while it is locked. The renamed multi-actions *syncLock* and

syncUnlock are used to lock and unlock the synchronization lock process respectively, while the renamed multi-action *syncCheck* can only be executed if the synchronization lock is unlocked.
Once the wrapper component sends the first notification, the synchronization lock is locked, followed by unlocking the thread (required for communication 7 in Figure 5). The wrapper component halts until the first notification is fully processed, which can be detected by checking that all queues are empty. The thread would also need to be locked again when unlocking the synchronization lock. Thus checking that all queues are empty, locking the thread and unlocking the synchronization lock is done in the multi-action *lock_thread*|*emptyQ*|*syncUnlock*.
syncCheck is done in synchronization with a call or modeling event from the framework (communication 1 and 8 in Figure 5) in order to prevent the framework from overtaking thread control while the synchronization lock is locked.

As a shared component with multiple clients, the wrapper component would need to keep track from which side it receives a call event from in order to send the proper reply event back. This information is stored as a data parameter in the processes of the wrapper component.

5. Verification and results

Different systems are constructed from the case study for verification. For all systems, it is assumed that they are always operational. Systems do not need to be initialized before being operational and they cannot terminate. Additionally, systems are either in *good weather behavior* or *bad weather behavior*. Systems in good weather behavior are unable to raise error notifications in foreign components through modeling events, thus systems cannot receive errors spontaneously. Systems in bad weather behavior can raise error notifications. The following systems are constructed:

- *Layers system*. A system focused on the cancel and control layer of the system, as seen in Figure 7. This system is in good weather behavior.
- *Layers no cancel*. The layers system without cancel requests. This system is focused on the execution of action a and b without cancels.
- *Full system GWB*. A system focused on the system as a whole. This includes the cancel and control layers, and is in good weather behavior.
- *Full system BWB*. The system as a whole in bad weather behavior. This system isn't complete and has been implemented to behave similar to good weather behavior. Error notifications do not affect the system. This system has been selected as the error notifications allow for a larger state space.

5.1. Verification

The toolset is run a 56 x 2 GHz and 935 GB memory Linux system. The large amount of memory makes it suitable to perform memory heavy tasks of the toolset on large models.

To perform the verification on a system, the LPS is generated from the mCRL2 model using *mcrl22lps*. The speed to generate the LPS can be increased by using the binary option of *mcrl22lps*.
The LPS is used to generate the LTS in the Aldebaran format using *lps2lts*. Enumeration caching techniques are used to speed up state space generation. Additionally, deadlocks are detected during the generation of the state space. As no reduction in the state space has been performed yet, the Aldebaran format is suitable due to the generation speed, since it only contains information about the transitions. A larger state space makes it more difficult to verify properties, hence the state space is reduced under divergence-preserving branching bisimulation [11] with *ltsconvert*. All non-event actions are hidden for this minimisation. This is done in order to speedup the verifications later.

Table 2 shows the amount of states and transitions of both systems before and after divergence-preserving branching bisimulation minimisation. The layers system is reduced by close to 50% in the size of the state space, while the full system in good weather behavior is reduced with approximately 77.84%. The full system in bad weather behavior is reduced with approximately 93.03%.

With the state space set, properties, written in modal μ-calculus, are used to generate PBESs through *lts2pbes*. Since the Aldebaran format only contains information about the transitions used, the original LPS has to be included to provide the required data and action specifications. The resulting PBES contains a number of equations that are to be solved. If the solution for the initial state is true, the property holds on the system. Otherwise it does not hold.
Constant parameters in the equations can be removed using *pbesconstelm*. This is also used to remove redundant equations, lowering the amount of equations to be solved and speeding the verification process. Furthermore, *pbesparelm* is used to apply parameter elimination. The PBES is solved using *pbes2bool*. The compiled jitty rewriting is used as rewriting strategy for the solver.

System	# States		% Reduction	# Transitions		% Reduction
	Before	After		Before	After	
Layers System	109,603	55,361	49.49	114,302	58,338	48.96
Layers no cancel	9,107	7,085	22.20	9,472	7,422	21.64
Full System GWB	17,093,409	3,787,974	77.84	19,002,987	4,298,103	77.38
Full System BWB	178,603,107	12,451,325	93.03	196,784,882	14,879,416	92.44

Table 2. Number of states and transitions for the systems before and after divergence-preserving branching bisimulation minimisation.

5.2. Layers

The layers system consists of both the cancel layer and control layer. The layers are responsible for correctly handling process and cancel requests on both sides.

5.2.1. Process. The process resembles the process described in section 4.1. The system decides to perform an execution or a cancel depending on the requests made on the client of both sides. In the event of an execution, the control layer ensures synchronization between both sides during their prepare and perform steps. Only once both sides have completed their prepare steps, the execution of a and b can continue in the perform steps.
Similarly, the cancel layer ensures synchronization between both sides in the event of a cancel. When a cancel is to be done, prepare steps as well as execution requests are canceled.

It is not allowed to make requests for actions that cannot be done at a given time. For instance, the client of a side is not allowed to make execution requests while a cancel is being done. These requests are blocked through the use of *Illegal* actions.

5.2.2. Properties. Table 3 contains simplified modal μ-calclus formulas of properties of the layers system. The initialized state, as explained in section 4.1, can be identified using property 0 from table 3. This is used by subsequent properties that must hold from the start of a round. The given properties are for side A only. Properties for side B have the same structure, but *A* and *B* are swapped. The following properties are derived and verified using mCRL2:

- An execution cannot be done if both clients have not made an execution request in the current round (Property 1).
- A cancel notification cannot be raised if both clients have not made a cancel request in the current round. The structure of this property is similar to property 1.
- A client cannot make an execution request if a cancel request has already been made by the client in the current round (Property 2).
- An execution is done after both clients have made an execution request (Property 3).
- A cancel cannot be performed if both clients have made an execution request. The structure is similar to property 3, except that after both execution requests a cancel cannot be done until the execution has finished.
- After two cancel requests in the same round, a cancel notification is raised (Property 4).
- When a cancel request has been made without two execution requests, a cancel will be done (Property 5).
- A client of a side cannot make multiple execution requests in a single round (Property 6).
- A client of a side cannot make multiple cancel requests in a single round. The structure of this property is similar to property 6.

Verifying all of the above properties results in *true*, implying that all properties hold on the system.
Additionally, the following properties are for checking the system to be free of errors. This is only required for the execution of a process and cancel requests are omitted. During the generation of the LTS of the layers system without cancel requests, only synchronization errors can occur. This helps simplify error checking by limiting to these errors only.

- An error cannot occur during an execution at a side until the client of that side can request a new execution (Property 7).
- A synchronization error will be raised at a side if the client of that side requests a new execution while the other side still needs to gather the results of the previous execution (Property 8). This error

ID	Property	Formula
0	Initialized state	$(\langle A\_Request\_Execute\rangle true \wedge \langle A\_Request\_Cancel\rangle true \wedge \langle B\_Request\_Execute\rangle true \wedge \langle B\_Request\_Cancel\rangle true)$
1	Cannot execute without two execute requests.	$[true^*](\text{Initialized state} \Rightarrow [((\overline{(A\_Request\_Execute)})^* + \overline{(B\_Request\_Execute)}^*).A\_Execute]false)$
2	Cannot request execution after a cancel request	$[true^*.A\_Request\_Cancel.\overline{(outwardNotification(A\_Canceled))}^* .A\_Request\_Execute]false$
3	Perform execution after two execute requests	$[true^*](\text{Initialized state} \Rightarrow [A\_Request\_Execute.\overline{(A\_Request\_Cancel \cup B\_Request\_Cancel)}^*.B\_Request\_Execute]\mu X.([\overline{A\_Execute}]X \wedge \langle true\rangle true))$
4	Raise cancel notification after two cancel requests	$[true^*.A\_Request\_Cancel].\overline{(outwardNotification(A\_Canceled))}^* .B\_Request\_Cancel]\mu X.([\overline{outwardNotification(A\_Canceled)}]X \wedge \langle true\rangle true)$
5	Perform cancel after cancel request	$[true^*](\text{Initialized state} \Rightarrow [A\_Requet\_Cancel + B\_Request\_Cancel] \mu X.([\overline{A\_Cancel}]X \wedge \langle true\rangle true))$
6	Cannot make multiple execute requests in a round	$[true^*.A\_Request\_Execute.\overline{(outwardNotification(A\_Executed))}^* .A\_Request\_Execute]false$
7	No synchronization error during an execution	$[true^*](\text{Initialized state} \Rightarrow [A\_Request\_Execute.\overline{(B\_Request\_Execute)}^* .B\_Request\_Execute.\overline{(A\_Request\_Execute \cup B\_Request\_Execute)}^* .Sync\_Error(A)]false)$
8	Synchronization error after requesting an execute too soon	$[true^*]((\langle A\_Request\_Execute\rangle true \wedge \langle B\_Get\_Results\rangle true) \rightarrow [\overline{(B\_Get\_Results)}^*.A\_Request\_Execute] \mu X.([\overline{Sync\_Error(A)}]X \wedge \langle true\rangle true))$

Table 3. Properties of the layers system written in modal μ-calculus

holds that one side is ready for a new round but both sides would need to synchronize first before a new execution request can be done.

Verifying the above properties results in *true* for the layers system without cancel requests.

5.3. Full system

The full system describes the system as a whole and includes not only the cancel and control layers, but also the prepare and perform steps. Additionally, this covers both the full system in good weather behavior and the full system in bad weather behavior.

5.3.1. Process. The process closely resembles the process described in section 5.2.1. Unlike the layers system, when a request for an execution or cancel has been made, more requests for the same action can be done in the current round. If the request is not supposed to be made, it will result in a protocol violation error raised instead of *Illegal*.
Additionally, prepare and perform steps are done. Prepare steps must be completed before the perform steps can be done, and the perform steps must be completed for an execution to be successfully executed.
Finally, error notifications raised in bad weather behavior are treated similarly to notifications raised in good weather behavior. This causes little differences in the properties between both full systems.

5.3.2. Properties. Table 4 contains simplified modal μ-calculus formulas of properties of the full system. The initialized state is identified differently than in the layers system as execution and cancel requests can be made at any time. Instead, the property states that when performing an execution or cancel request (as call event), a protocol violation error cannot occur for that request. Property 0 shows the structure of the formula, limited in showing the execution request at side A only for visualization. Similar to the layers system, the ini-

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

Table 4. Properties of the full system written in modal μ-calculus

tialized state is used in subsequent properties and all properties shown are for side A only. The following properties are derived and verified using mCRL2:

- An execution is done after both clients have made an execution request (Property 1).
- All prepare steps at a side have been done before an execution is being performed at that side (Property 2). Side B intentionally fails as side B has multiple prepare step, but continues to the perform step after only done one of them.
- All perform steps at a side have been done before the notification for a successful execution at that side has been raised (Property 3).

Verifying the above properties results in *true*, with the exception of the prepare steps done at side B. Those results in *false* as expected.

5.4. Verification statistics

During the execution of the toolset, the time required for each task has been tracked. Table 5 contains the average time for the toolset spent on each task for all properties of the respective system.
From Table 5, there is a significant growth in time compared to the states in Table 2. A single property for the full system in bad weather behavior takes over one and a half day to verify. Comparing the layers system and the full system in bad weather behavior, a growth of approximately 225% in state space size caused a growth of over 2,142% in time. This is mainly caused by *pbesconstelm*. *pbesconstelm* covers only 24.83% of the total time in the layers system, but 61.09% in the full system in bad weather behavior.
Verification on the full system in bad weather behavior also shows difficulties in memory usage. Verifying a property of the full system in bad weather behavior requires between 100 and 300 GB in RAM memory.

6. Related work

mCRL2 has been used for the verification of real life embedded systems, such as pacemakers [12]. The behavior of the system, composed out of components, can be modeled in order make the communications between components explicit. This is the case for the research of this paper, in which events, translated as actions, allow explicit communications between components. The approach to model embedded systems is done different in order to capture the expressiveness of the single-threaded execution model of ASD.
Another mCRL2 use case is the formal verification of the Wheel subsystem, a component of the Resistive Plate Chamber subdetector of the Compact Muon Solenoid experiment [13]. While the verification is limited to a single component, the component itself is composed out of several finite state machines (FSM). Each FSM is translated to a single mCRL2 process, and the composition of several FMSs has been verified. Similar to the research of this paper, scalability forms a difficulty.

	Time (s)	% of total
Layers System		
lts2pbes	34.09	55.44
pbesconstelm	15.27	24.83
pbesparelm	5.66	9.21
pbes2bool	6.47	10.52
Total	61.49	
Layers no cancel		
lts2pbes	1.64	34.75
pbesconstelm	1.13	23.94
pbesparelm	0.79	16.74
pbes2bool	1.16	24.57
Total	4.72	
Full System GWB		
lts2pbes	7,863.64	39.78
pbesconstelm	7,762.68	39.28
pbesparelm	2,240.30	11.33
pbes2bool	1,899.39	9.61
Total	$19.77*10^3$	
Full System BWB		
lts2pbes	$39.67*10^3$	30.11
pbesconstelm	$80.47*10^3$	61.09
pbesparelm	5,566.49	4.23
pbes2bool	6,014.61	4.57
Total	$13.17*10^4$	

Table 5. Average time spent for each task on each system.

Next, CADP (Construction and Analysis of Distributed Processes) [14] is a toolbox of asynchronous concurrent systems. Similar to mCRL2, it supports model checking for temporal logic and μ-calculus. A fault-tolerant routing algorithm in Network-on-Chip architectures has been verified using CADP [15]. The algorithm is composed out of components. Instead of composing the system as a single model beforehand, LTSs for each component are generated first. These LTSs are incrementally composed and minimized into a single LTS of the whole system.
Another CADP use case is the formal verification of a reconfigurable monitoring application [16]. Reconfigurable interfaces of this application lead to a combinatorial explosion of the state space. Similar to the research of this paper, irrelevant actions are made hidden in order to reduce the state space through minimisation. The minimisation process has not always been successful due memory issues caused by the size of the state space.

Furthermore, Dezyne[1], a model-driven software engineering tool developed by Verum, is a successor of ASD. Similar to ASD, it translation from a Dezyne model to an mCRL2 model [17] is provided. This translation is also limited to the scope of a single component, including interfaces found within the component boundary.

7. Conclusion

This paper presented an approach in verifying end-to-end properties on multi-component systems designed in ASD. The case study of a system, in which two sides synchronize the processing and cancellation of their actions, has been evaluated. The verification of the whole system, composed of 26 components, was successful as all properties resulted in the expected outcome.

Scalability forms a difficulty. Depending on the evaluated system, the state space size ranges between 10 thousand and 13 million states. The size of the state space grows exponentially in relation to the amount of components and events. Consequently, this results in an increase on the time and memory requirement of the verification. The largest system requires over one and a half days in time and over 200 GB in RAM memory. This is problematic for running the verification on conventional machines.

For future work, the presented approach could be optimized in order to reduce the time and memory required for verification. Next, the presented approach is limited to systems in the the single-threaded execution model. This can be expanded to support systems in the multi-threaded execution model as well. Finally, due the similarities in ASD and Dezyne, the presented approach grants an opportunity to expand the translation from Dezyne to mCRL2 to cover multi-component Dezyne systems. This allows multi-component and end-to-end verification on systems designed in Dezyne.

References

[1] G.H. Broadfoot, P.J. Hopcroft. *Analytical Software Design*. Verum Consultants B.V. 2003.

[2] J.F. Groote, M.R. Mousavi. *Modeling and analysis of communicating systems*. The MIT press. 2014.

[3] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, J.W. Wesselink, T.A.C. Willemse. *An Overview of the mCRL2 Toolset and Its Recent Advances*. TACAS 2013, LNCS, Vol 7795, pages 199-213.

[4] K. Aslam, N. Yang, Y. Luo, R.R.H. Schiffelers, A. Serebrenik, M.G,J. van den Brand. *Model inference for soft-*

[1] See `https://www.verum.com`; accessed 24 June 2018.

ware: Combining active and passive learning results. ICT.OPEN. 2018.

[5] E. Rontogiannis. *Protocol-compliant simulators: generating protocol-compliant simulators for and from DCA's control interfaces*. Technische Universiteit Eindhoven. PdEng Thesis. 2017.

[6] E. R. Sahagun. *Translation of ASD Single-threaded Execution Model to EFSM*. Eindhoven University of Technology. Master Thesis. 2014.

[7] F.J.W. Zeelen. *Supervisory machine control of ASML wafer scanners*. Eindhoven University of Technology. Master Thesis. 2007.

[8] M. J. Starke. *Supervisory control using extended finite automata for ASML wafer scanners*. Eindhoven University of Technology. Master Thesis. 2013.

[9] R. Loose. *Component-wise Supervisory Controller Synthesis using existing plant models in a client/server structure*. Eindhoven University of Technology. Master Thesis. 2017

[10] R. Jonk. *The semantics of ALIAS defined in mCRL2*. Eindhoven University of Technology. Master Thesis. 2016.

[11] J.F. Groote, D.N. Jansen, J.J.A. Keiren, A.J. Wijs. *An O(m log n) algorithm for computing stuttering equivalence and branching bisimulation*. ACM Transactions on Computational Logic. 2017, Vol. 18, No. 2.

[12] J.E. Wiggelinkhuizen. *Feasibility of formal model checking in the Vitatron environment*. Eindhoven University of Technology. Master Thesis. 2007.

[13] Y.L. Hwong, J.J.A. Keiren, V.J.J. Kusters, S.J.J. Leemans, T.A.C. Willemse. *Formalising and analysing the control software of the Compact Muon Solenoid Experiment at the Large Hadron Collider*. Science of Computer Programming, 2013, Vol. 78, No. 12, pages 2435-2452.

[14] H. Garavel, F. Lang, R. Mateescu, W. Serwe. *CADP 2011: a toolbox for the construction and analysis of distributed processes*. STTT, 2013, Vol. 15, No. 2, pages 89-107.

[15] Z. Zhang. *Fault-Tolerant Routing Algorithm for a Network-on-Chip*. The University of Utah. PhD Thesis. 2016.

[16] N. Gaspar, L. Henrio, E. Madelaine. *Formally Reasoning on a Reconfigurable Component-Based System — A Case Study for the Industrial World*. FACS 2013, LNCS, Vol 8348, pages 137-156.

[17] R. van Beusekom, J.F. Groote, P. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, T.A.C. Willemse. *Formalising the Dezyne Modelling Language in mCRL2*. FMICS-AVoCS 2017, LNCS, Vol 10471, pages 217-233.