FORMAL ANALYSIS AND VERIFICATION OF INTERACTIONS IN THE O2N MIDDLEWARE FRAMEWORK

MASTER THESIS

L.C. (Lukas) Bos
s1583603
14/08/2020

Faculty of EEMCS

Supervisors University of Twente
dr. ir. A.B.J. Kokkeler
dr. ir. R. Langerak
ir. E. Molenkamp

Supervisors Thales Nederland B.V.
ir. G. I. S. Hoekstra
ir. M. Horsthuis

THALES

UNIVERSITY OF TWENTE.

ABSTRACT

The research described in this thesis aims to determine the extent to which formal methods and tools can be applied to analyze distributed systems that communicate internally using O2N, which is a middleware framework created by Thales. An analysis has been made of the characteristics of communication patterns in O2N, and it is explored which formal tools can be applied. Using one of these tools, it has been determined to what extent formal methods can be used to verify and analyze interactions between components in O2N-based systems. The main characteristics of interactions in O2N is that components are structured using a reactive architecture. Communication is often performed asynchronously, where complex message types are sent using a protocol called 'command-state'. To provide this protocol, a publish-subscribe pattern is used. An abstraction on the behaviour of components, based on their interactions, can be described using ComMA specifications. Based on these characteristics, SPIN is chosen as formal tool to perform the analysis. The ComMA specifications can be used to create the Promela models which SPIN needs. After verification, it appears that interactions in O2N can be at least verified for deadlocks, livelocks, invariants (global variable bounds), and some liveness properties. Since the limits of what SPIN can verify are being reached, it is possible to try other tools in future research to further determine the extent to which formal methods can verify and analyze O2N-based systems.

FOREWORD

What you are now reading is the result of my graduation project, on which I have been working for almost 8 months. I would like to thank André, Bert and Rom from the University of Twente for their support and advice. I also like to thank Mark and Idzard from Thales for all the help and for investing their time in my work, even after the Corona lockdown started. Although we could not meet in person, the supervisor meetings always have been of great help.

This thesis also marks the end of six years as a student at the University of Twente. First of all, I want to thank my parents for their endless support. I want to thank Ruben, Rutger, Martijn, Dennis, Bart and many more from C.S.V. Alpha for all the fun and good talks, but also for being there during more difficult times. I want to thank Rolf, Cas and Thijs and all other RoboTeam Twente members, for being part of one of the most educative and fun experiences I ever had. I want to thank Mirjam & Michiel for their support, albeit from a distance. Finally, I would like to thank Mirjam for making me believe in myself.

”Simplicity is prerequisite for reliability” - E.W. Dijkstra

Contents

1 INTRODUCTION

1.1 Motivation

Thales builds radar installations for maritime and military purposes. Their radar installations are used to, for example, detect ballistic missiles and (stealth) planes at large distances at a range of more than 1000 kilometers, but are also used for close range surveillance. Such an advanced system is divided into a lot of different subsystems distributed over multiple devices. In the software running on these devices these subsystems are divided into *components*. These components are all executing their processes parallel to each other, and interact with each other using asynchronous message passing. This classifies the whole system as a distributed system. Structuring such a large system into multiple loosely coupled components has several advantages, such as modularity, flexibility, and maintainability. However, the heavy use of concurrency introduces various risks, such as deadlocks, livelocks, and race conditions. Currently, to provide the communication between all components Thales has created the middleware framework *O2N*. O2N is already being deployed and it has shown itself to be functional and usable. However, Thales is interested in having more insight in the way components interact with each other using formal methods and tools. Formal methods and tools can be used to analyze complex behaviour of concurrent processes in distributed systems, and can therefore be used to help reduce the risks often found in such systems. However, it can be difficult to properly apply such a tool, since systems can have characteristics that make analysis difficult or even impossible. This research is aimed at the application of formal methods and tools to interactions between components in an O2N-based environment.

The Thales Smart-L radar on top of a defence frigate.

1.2 Research objective

The goal of this research is to determine to what extent formal tools can be applied to O2N components such that the behaviour of these components can be analyzed for potential problems. By doing this it may be possible to find problems that occur in interactions between components that would otherwise not have been found, or would only have been found later in the development process. This is formulated into the following primary research question:

To what extent does the O2N framework need to be extended such that the interaction between a given set of O2N components can be modelled and formally verified?

The primary research question is divided into multiple sub-questions:

1. *What are characteristics of interactions between components in the O2N framework?*
2. *What tools are capable of verifying, and give insight in, the interaction between components in the O2N framework?*
3. *To what extent can a single O2N component be modelled that it suits both formal verification and code generation?*
4. *How can the interaction between a set of O2N components be modelled and formally verified?*

The first research question aims to determine the characteristics of O2N that may or may not be relevant when attempting to formally analyze this system. When these characteristics are known, they can be used for the second research question, which aims at finding the right formal method or tool that fits these characteristics. A literature study will be conducted on existing formal verification tools and methods and how these can be applied in the context of O2N. Given the potential complexity of an O2N component, it is expected that there are challenges to overcome and that some tools may be more useful than others. The third research question deals with the question on how an O2N component can be translated into a formal model, and if that process can be done automatically. It will be researched whether there are characteristics of O2N that currently cannot be specified formally. The fourth research question deals with modelling the interactions between components. If a method has been found to generate a formal model for a single component, then how can these models be used for verification purposes? Possible challenges that may arise at this point are: handling unused inputs for components, state-space explosions, and dealing with complex messages that are sent between components.

1.3 Report structure

In chapter 2 a literature study is conducted. This literature review aims to answer the first two research questions. It shows an overview of the characteristics of O2N, an overview of relevant research on O2N that has been conducted previously, as well as different formal methods and tools that have potential to describe and verify concurrent distributed systems like O2N. In chapter 3 an approach is shown on how the remaining research questions will be answered. In chapter 4 the process of selecting SPIN as suitable formal tool is described in detail. Then, chapter 5 shows how the formal models are being created to be analyzed with SPIN. In the results, chapter 6, several attempts at verifying different properties of the models are being described. The discussion chapter, chapter 7, displays some reflection on the research. The conclusion of this research can be found in chapter 8. Further reference to support this research, such as a list of terms, large tables, and source code, can be found in the Appendices.

2 BACKGROUND

In this chapter a background research is conducted, which aims to answer the first two research questions. In section 2.1 an overview is given of the characteristics of communication patterns in O2N. In section 2.2 the existing research on O2N is described. In 2.3 the tool 'ComMA' is described, which is a tool that plays an important role in this research. Section 2.4 shows an overview of multiple formal methods that have potential to formally describe O2N, for example Petri nets and process algebra. Section 2.5 gives an overview of useful formal concepts that are used throughout this thesis.

2.1 The O2N middleware framework

O2N is a middleware framework, and it is used to provide communication in distributed systems created by Thales. A middleware framework can be seen as the connecting element between multiple components in a distributed system. In a distributed system, different components can run on different devices, and each device runs it's own operating system (OS). Middleware acts as an additional layer on top of this OS to provide the same interface to each component [1] throughout the distributed system. A component within such a system is simply a separate software application using the interface of the middleware. A visualization of the middleware layer is given in Figure 2.1.

A major advantage of using middleware is that it hides the networking complexity from the components. Therefore when a component is connected to this framework, the component can receive inputs from the middleware and can transmit outputs to this middleware as well. The middleware is then responsible for the communication between different components, regardless of the machine on which they run. It is possible that a component only has inputs to O2N (such as a component controlling engine hardware) or only outputs to O2N (such as a com-

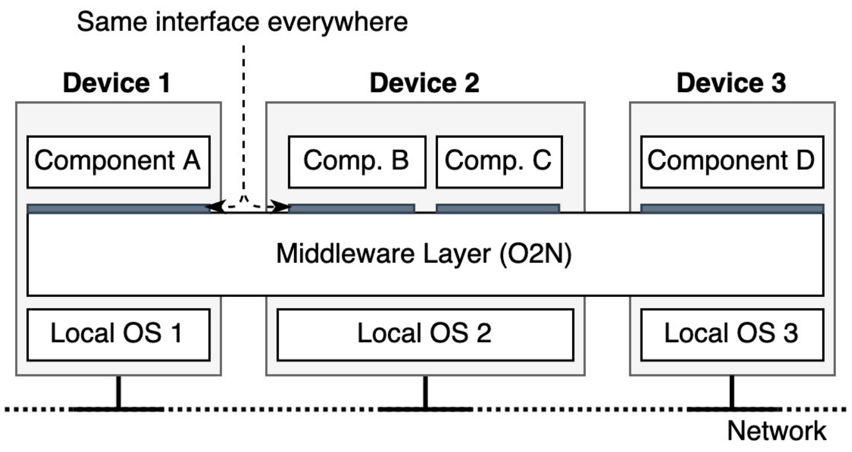


Figure 2.1: Visualization of the middleware layer.

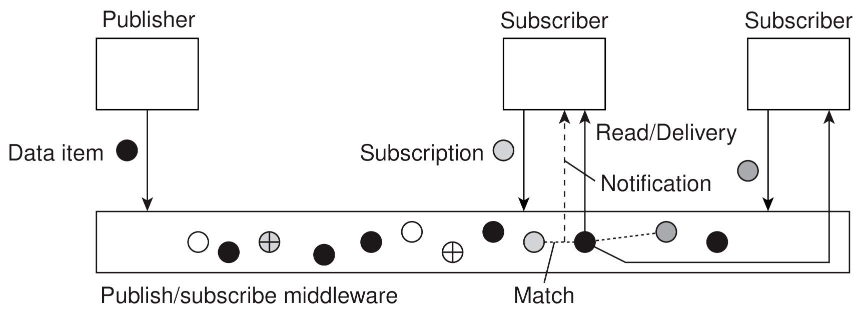


Figure 2.2: Visualization of the Publish-subscribe pattern [1]. Data is represented by white, black, gray and crossed circles. A publisher publishes data represented by either a black, white or crossed circle. The publisher in the figure publishes a black circle. The subscribers publish a subscription, in the image represented by the grey circles. When a subscription matches with some published data, the subscriber will be notified with that data.

ponent that sends input from a sensor). Hiding network complexity from the components also implies that developers can develop these components without having to worry about details of the data transfer. Data transfer is usually done using the *publish-subscribe pattern*. There are also some other (synchronous) methods of communication a component can have, but these are not considered relevant for this research, because the dialogues between components always use the publish-subscribe pattern. Within the boundaries of the publish-subscribe pattern, O2N gives developers freedom to specify what kind of communication to use. For example, when it is known that two components run on the same device, the developer can indicate that they communicate using shared memory (instead of over the network) to reduce both message latency and network traffic.

2.1.1 The publish-subscribe pattern

The publish-subscribe pattern (often abbreviated to publish-subscribe or pubsub) is a communication pattern used by O2N that allows for a loose coupling between components, thus making the system more scalable. It is the base of communication between components. When a component that uses publish-subscribe has an output to send over the network, it *publishes* this output on the network without knowing whether other processes will receive it. To differentiate between multiple kinds of messages, a message is published with a *topic*. When a component needs inputs from the network, it *subscribes* to a *topic*, which represents messages of a given type and/or origin. Using publish-subscribe one component can publish a message, and multiple subscribers can subscribe to the corresponding topic to receive the message. There are multiple variants of publish-subscribe, such as shared-memory publish-subscribe, which works with a shared memory and therefore has a 'broker' that stores the message before a subscriber receives it. Another variant is 'brokerless' publish-subscribe using an event bus, which does not store messages 'in between'. Therefore, if a message is sent and there are at that point no subscribers, the message gets lost. A visualization of the publish-subscribe architecture is given in Figure 2.2.

It is shown that a publisher pushes a data item to the middleware, and that the subscriber can subscribe to such a data type through a *topic*. If the topic of the subscription matches the topic of the data then the subscriber will retrieve the data and process it. Using the publish-subscribe protocol a virtual link emerges between nodes. While in fact all nodes are connected to the same network, virtually they are more separated, only talking to nodes with a corresponding topic (Figure 2.3). The publish-subscribe pattern does not specify anything about the content of the messages. However, a topic is always related to a *message type*. This means that the

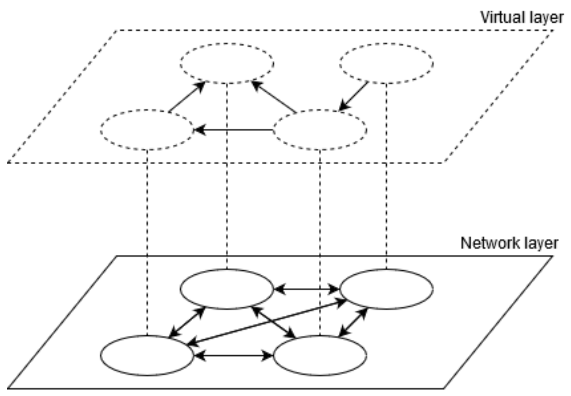


Figure 2.3: Visualization of node abstraction: While all nodes are in the same network they can, from a functional point of view, be decoupled. This means that while in some sense every component in an O2N-based system may communicate with every component, it is beforehand specified to which other components a component communicates. When in this research an interaction between components is described this is always an interaction based on the virtual layer.

structure of the messages sent on a topic is always the same. A component can subscribe to multiple channels at a time to receive different types of messages. Or, similarly it can publish to different channels to publish different types of messages. How these messages are structured is up to the designer of the interactions of the component: a component may subscribe to only a few topics that contains more complex messages, or it may subscribe to many topics with less complex message types.

2.1.2 Reactive architecture

O2N components act on messages and events, indicating a reactive architecture. This means that an O2N component executes only when a message is received, or when an event is triggered (for example when receiving sensor input or a when using a timer). By using this reactive architecture Thales gets more control on concurrent behaviour between O2N components. Using this architecture it can be reasoned that some components would never execute code when no messages are coming in. In such a case another process has to execute (send a message) before that component can execute (after receiving that message), thus having ensured that the two components always execute sequentially. This is only the case if the component does not execute on internal events. This is something that needs to be considered when designing an O2N component. For this research it cannot be assumed that a component is purely reactive. If not the full implementation of a component is known it must be assumed that a component can as well trigger events without having received an input from O2N. Still the reactive properties can be useful for verifying the components, as it is known that the component has to handle it's inputs and must exhibit specified behaviour on O2N inputs.

2.1.3 O2N component specifications

Up to this point specific properties of O2N and how O2N components work together have been discussed. Now it is shown what such a component looks like. An example is given in Listing 2.1. When a component is being created for O2N, the first step is to create a specification of the interface. This specification contains settings for O2N, for example which protocol to use for sending and receiving messages, for example ZMQ, TCP, UDP or shared memory. Also, the specification describes the inputs and outputs of the component, and what types of messages are expected at these inputs and outputs. In the example it can be seen how the publish-subscribe pattern is used: messages of type `PingPong` are published to the topic called

```
component DummyComponent {
    publish ”ping” as ping {
        messageInterface = PingPong [1,1.1)
    }
    subscribe ”ping” {
        messageInterface = PingPong [1,2)
    }
    timer pingCheck {
        initialSeconds = 1
        intervalSeconds = 1
        realtimePriority = 5
        schedulingPolicy = SCHED_RR
        autostart = true
    }
}
```

Listing 2.1: Component specification example. This component can send and receive message of type `PingPong` on the ”`ping`” topic. It also specifies a timer with an interval of 1 second and some other properties for the scheduler, influencing the reliability and scheduling behaviour of the timer. This specification can be converted to either a C or a C++ program.

”`ping`”, and the component subscribes to the ”`ping`” topic as well using the same message types. It may be clear that this dummy component is therefore not very useful: is uses O2N to talk to itself. Knowing the format of the messages that are transmitted between components is interesting for this research, since from component specifications like the one in Listing 2.1 it can be determined which components interact with each other by the topics to which they publish and subscribe. This specification therefore gives the component some context, which is useful when analyzing and verifying the interactions of the component. After creating such a component description, a component can be auto-generated, providing the necessary functions for the developer to provide the communication between components. In Listing 2.2 an example is shown of a header file, which is generated from the specification in Listing 2.1. O2N has generated a base-class that provides functions for sending data, and provides 'hooks' for retrieving data, e.g. it defines a function that is called when a message is received. The programmer must override this function with his own implementation to properly handle the message. Also a function is available to transmit messages, which the developer must call in the implementation.

```
class DummyComponent : public DummyComponentBase {
public:
    DummyComponent();
    int init() noexcept override;
    int start() noexcept override;
    int stop() noexcept override;
    int deinit() noexcept override;

    // override that is called by the timer
    void pingCheck_timeout() noexcept override;

    // function to call when a ping is received
    int ping_receivePing(const examples::Ping &ping,
             pubsub::ISubscriberCallbacks &callbacks) noexcept override;
};
```

Listing 2.2: C++ header file generated from the O2N component specification. Besides the constructor, there are some functions that give a developer control on the *lifecycle* of the component (e.g what to do when the component starts, stops, etc). There is also a `pingcheck_timeout()` function which is generated from the timer in the specification. As specified, O2N makes sure this function is called every second. The last function is a function that is called when a message is received on the ”`ping`” channel. For the publishing of messages no function is shown, but it does exist in the base class. The `publishPing(...)` function must be used by the developer to publish messages.

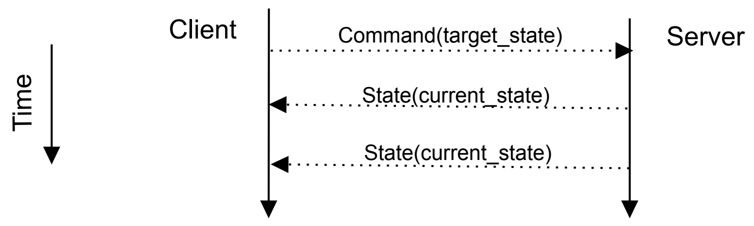


Figure 2.4: Visualization of the command-state pattern. The client sends a command to the server. As an acknowledgement, the server will reply with it's current state. This state can be a transition state which make sure the target state will eventually be reached (for example, an engine accelerating to a target velocity). When the target state is reached, it sends another state message.

2.1.4 Usage of O2N

O2N uses the publish-subscribe pattern as an abstraction for communication, but there is still freedom on designing interactions within the technical boundaries of the publish-subscribe pattern. To give some examples: the publish-subscribe pattern allows components that don't communicate at all, or that there is a monologue (1 component sends and another one receives, but does not send back), or that there is a dialogue where both components send and receive messages to each other. These monologues and dialogues can exist in many forms as well. In reality all these variations can be present in a system. However, when developing components, the developers at Thales use 'best practices' on how to specify these communications. Having such best practices can be beneficial to achieve consistency throughout a distributed system.

One of the best practices used by Thales is the 'command-state' pattern, which is a pattern used on top of the publish-subscribe pattern. It is a pattern that describes a dialogue between components. The pattern describes that if a client sends a command message to a server, this message will contain the desired state of (a part of) the server. The server will then reply to this message with a state message. This can be an acknowledgement, or a rejection. In both cases it will reply with it's new state. This new state can be a temporary state, for example accelerating an engine. An important aspect to the command-state pattern is that whenever the internal state of a component changes, it must notify this to the other components that are subscribed to the state messages of this component. For example, after the command message to rotate an engine with a given speed, the component controlling the engine may first reply with a message indicating that it is in the 'accelerating' state, and when the desired velocity is reached and it will move out of the 'accelerating' state, then it has to send a new message indicating the velocity is reached and the state changed. The command-state pattern is visualized in Figure 2.4. In the command-state pattern, the component that sends commands is in the dialogue regarded as the client, while the receiver of the commands is called the server. These terms are only valid for the dialogue under consideration, since the client can also participate in another dialogues where it can be the server instead.

2.1.5 Summary of characteristics of O2N based systems

O2N is a middleware framework that uses asynchronous message passing to communicate between components. The components themselves are generally built using a reactive architecture, which means that the components respond to either internal events or messages from other components. Components handle these messages sequentially. Different communication patterns can be used while creating a specification, however, this research focuses on the publish-subscribe and command-state patterns. A component is created from a specification which describes the inputs and outputs of a component. This specification can be used to generate C++ placeholder files that do most of the O2N integration for the developer.

2.2 Existing research on O2N

2.2.1 Continuous Evolution of software systems

Another study at Thales, which shares a common ground with the one described in this thesis, is currently in progress. In collaboration with the Dynamics research group from TNO[1], Thales conducts research on 'continuous evolution' of software systems. For some of their applications, which need to last for multiple decades, sometimes software updates are created in a much higher rate than when these updates can actually be applied to the systems. This means that although the software systems are updated regularly, in practice the updates that are applied are infrequent, big updates that have a high impact on the system. Therefore it is attempted to use a formal model, open (Petri) nets [2], to simulate the interconnection of old and new components. If a component has been updated, it can be simulated if it would still work the same way the previous component would do. Then it can be determined that either the collaboration of components works, or that a problem occurs, for example a deadlock. Therefore, for every updated component it can be known if it either does or does not work within a larger system. While this research certainly has similarities with the research conducted in this thesis, there are differences. The key difference is that 'continuous evolution' aims to verify interfaces that replace older interfaces, while this research aims to verify any kind of interface, including new ones. Due to the different approaches the 'continuous evolution'-research uses open nets, while the research described in this thesis has yet to determine which formal method will be used. Currently one paper of the 'continuous evolution'-research has been published [3]. Because of the ongoing research for 'continuous evolution', Thales already aims to use the tool *'ComMA'* (Further described in section 2.3) for their component specifications.

2.2.2 Verum Dezyne

In the past, Thales has worked with the formal tool Dezyne. Dezyne [4, 5] is a tool created by Verum B.V. that provides formal tooling in an industrial setting. This is done by using mCRL2 as the verification engine [5, 6], which is further described in section 4.2. Dezyne provides developers with a graphical user interface to define state machines for their models. Such a state machine is used to provide the model checking and to generate an outline of the desired component, e.g, the signature and behavioural interface are generated from specification while the developer can work on the details of the implementation. With Dezyne a developer can specify the signature and behavioural interface using special .dzn models, which can be formally verified. From the specification it is also possible to generate C++ code that satisfies this specification. Dezyne is already an end-product. For Thales it might not be of interest to have their whole architecture migrated towards Dezyne, as this would make Thales quite dependent on this tool. Besides, Thales is considering to use the ComMA specification language (section 2.3) as their underlying framework. If ComMA is used, then using Dezyne may result in unnecessary conversions between languages. Internally, Dezyne converts .dzn models to mCRL2 models. When using both ComMA and Dezyne a model would be converted from ComMA to .dzn to mCRL2. To prevent the risk of errors and/or incompatibilities occurring during these conversions, it may be safer to directly convert form ComMA to mCRL2 instead.

[1]Netherlands Organisation for Applied Scientific Research

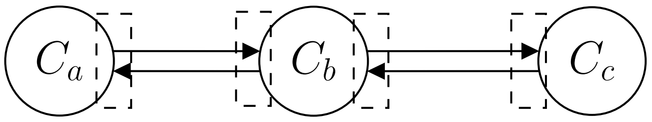


Figure 2.5: The locations of interfaces in components, marked using striped rectangles. An interface is an entry- or exit-point of a component. A component can have multiple interfaces. An interface belongs to one component, (but one interface can be used to communicate with many other components).

2.3 ComMA

ComMA stands for 'Component Modelling and Analysis'. It is a tool initially designed for use in the medical domain by Philips [7, 8]. ComMA provides a language to specify interfaces such that it can perform runtime verification on the behaviour of components. Thales is looking to change their component descriptions to ComMA specifications. There are, however, still some challenges to overcome and the ComMA language will properly be somewhat different from what it is now if/when Thales adopts the tool definitively. The specification language used in ComMA makes a distanction between a *'signature'* and a *'behavioural'* interface. It is intentional that both the signature and behaviour are annotated with the word 'interface', as the behavioural interface does not fully describe the behaviour of the component. To clarify this further, it is first necessary to have a solid definition of what an interface exactly is in the context of ComMA and O2N.

2.3.1 Redefining 'interface'

A ComMA specification defines the *signature interface* and the *behavioural interface* of a component. For O2N components, the combination of the signature interface and behavioural interface is simply called the *interface* of the component. Figure 2.5 shows a possible interaction between three components. The start and endpoints of arrows can be seen as interfaces (both signature and behavioural) and are marked with striped rectangles. In terms of ownership, an interface belongs to one component. Other components know about this interface to communicate with the component. A component can have multiple interfaces. Because of this structure it is possible that interface definitions can be reused. Therefore different components can still use the same interface if a part of their behaviour is similar. For example, two completely different engines can both have an instance of the same 'rotation' interface, although the rotation messages may be interpreted differently.

Signature interface A signature interface describes how a component communicates with the outside world. It can also say something about the internal structure of the interface itself. The signature is similar to the definition of an interface in (object-oriented) programming, where it is defined as an abstract class to define a specific type of classes that belong together to for example enable polymorphism. The signature consists of groups of synchronous and asynchronous calls and asynchronous notifications.

Behavioural interface A behavioural interface can be explained with a state machine. As the name implies this kind of interface describes the behaviour of a component. It describes specific states of a component, and it describes which transitions can be taken to take a transition into another state. For a given state, the availability of inputs and outputs (which are indicated by the signature interface) of the component may change. The signature interface itself does not change when a component changes state, since it is a definition. The state can however alter the values that being transmitted (or ensure that no values are transmitted at all). These values, ComMA

```
transition trigger ReceivedCommandMsg(
    CommandMsg message)
    guard (message.onOffState == OFF)
    do: dummyValue = 0;
    next state: OFF;



...
```

C++

```
void receivedMessage(CommandMsg message) {
    if (message.onOffState == OFF) {
        stopDoingTask();
        dummyValue = 0;
        internalValue = 1;
        delete previousCommandPtr;
        this->state = STATE::OFF;
    }
}
```

Figure 2.6: Example of abstractions using a fictional implementation. This example illustrates the differences of visible behaviour in the ComMA specification and real behaviour in the C++ implementation. Although the guard, the variable `dummyValue` and the next state are all present in the C++ implementation, the C++ implementation actually does more, such as cleaning up memory, handling some local variables, or interfacing with hardware. However, if the specification is correct these C++ implementation details should not affect the global behaviour of the component, such that these details can be ignored.

when put into another component, can alter the state of that component. In the behavioural interface it is also possible to give constraints on data and time, such as allowed response time, periodicity of notifications and data relations between parameters of subsequent calls [7]. The behavioural interface is not to be confused with the actual behaviour of a component, which may be different. The behavioural interface can also display a higher-order behaviour of the component while lower-order behaviour may remain hidden. When implemented correctly, the behavioural interface can thus be regarded as the *visible behaviour* of a component. This is illustrated in an example in Figure 2.6.

A complete interface, containing both a signature and behavioural interface, can therefore show inputs and outputs of a component together with the behaviour of the component, albeit using an abstraction of the real behaviour of the component. This information is useful when analyzing and verifying the possible interactions this interface can provide.

2.3.2 Using ComMA

In Figure 2.7 an overview is given on how ComMA will be used by Thales. The ComMA specifications will replace the component specifications (described in subsection 2.1.3). It is important to remember the relationship of a *component* with an *interface*. A ComMA specification describes an interface of a component. But a component can have multiple interfaces, such that interfaces (and therefore also ComMA specifications) can be reused between different components.

Because ComMA may eventually replace the component specifications, Thales components may be created using C/C++ placeholders that are generated from ComMA specifications. The ComMA language can easily be extended to support generation to new languages. ComMA is created using XText [9], and in combination with the XTend plugin [10] it is possible to develop code generators to convert a ComMA specification to other languages. This functionality makes it easy to create the C/C++ placeholders. But creating code placeholders is not the only feature ComMA has to offer. What makes ComMA different from the previous component specifications is the *runtime monitoring* functionality. From ComMA a 'monitor' application can be generated that determines during runtime whether the component is behaving as specified. Besides specifying behaviour, ComMA also allows to set data constraints on variables, providing runtime assertions when these variables have illegal values. Also timing constraints can be given in ComMA, specifying maximum delay in communication between components.

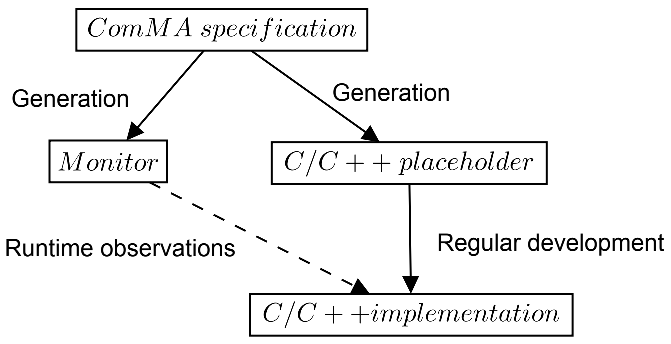


Figure 2.7: Schematic on how ComMA is used. From the specification a monitor and a C/C++ placeholder application is generated. The monitor observes if the implementation behaves as specified.

2.4 Formal methods

In this section some common verification methods are discussed. These methods provide some fundamental approaches to verifying a system. There are many formal methods and tools available. Some of these methods and tools are more applicable to O2N than others. In this chapter some general formal approaches are discussed that are designed for distributed systems.

2.4.1 The actor model

The actor model is a model that splits every process into a so-called *actor*. An actor is a primitive unit of computation [11], which means it can receive a message and do some kind of computation based on that message. Actors are completely isolated from each other, and they will never share memory. It is possible that an actor has a private state which can never be changed directly by another actor. Although multiple actors can run concurrently, an actor in itself computes sequentially. Therefore if you send 3 messages to one actor, these messages are processed sequentially. This means that if one wishes to process 3 messages concurrently, it is necessary to create 3 actors. According to the actor model, an actor can do three things when it receives a message: create new actors, change it's behaviour, or send messages to other actors. A behaviour can be seen as the private state of the actor. A major advantage of the actor model is that it makes large distributed systems more scalable and transparent. Also, it makes the separate entities more consistent with each other. Because of this it is widely used in the software industry. What the actor model cannot do, is verifying properties of the system, such as preventing deadlocks, livelocks, etc. The actor model has inspired programming languages, such as Erlang and Scala [12]. Many other languages have libraries that implement the actor model, for example CAF for C++ [13] and Libactor for C [14]. However, it also has influenced concurrency theory [15], as it has inspired the process calculi 'Communicating Sequential Processes' (CSP) and the 'Calculus of Communicating Systems' (CCS), which are further described in subsection 2.4.3.

What is interesting however about the actor model is the similarity of actors with O2N components. Like the actor model, an O2N component can initialize other components, change private state or send messages to other components. Also the components execute in parallel, and have messages from other actors as stimulus. It is unknown if O2N is designed according to the actor model, but the resemblance shows that the structure of O2N is not uncommon. Some tools for formalization of system behaviour using the actor model do exist. One of these tools is $\mathcal{R}$ebeca [16] which is designed to apply formal methods on real-world applications using the actor model.

2.4.2 Petri nets

Another method of formalizing system behaviour is by using Petri nets. Since the introduction of Petri nets in 1962 [17], a large amount of variations on Petri nets have been created, and many tools have been developed to work with Petri nets [18]. A Petri net is a graphical and mathematical modeling tool that can describe concurrent processes. Because Petri nets have a clear graphical notation, they can also be used to provide visual support when describing such processes. A Petri net is structured in *transitions* and *places*. Throughout a Petri net there exists a distribution of *tokens*. A distribution of these tokens in a Petri net is called a *Marking*, which represents the state of the Petri net. Thales already uses Petri nets (more specifically: Open Petri nets) for the 'continuous evolution' research. Figure 2.8 shows two simple examples of the same Petri net with different marking. The transition t can only *fire* when all places leading to t contain a token. The respective amount of tokens is then *consumed*, while new tokens are being *produced*. A visualization of the Petri net firing rule is given in Figure 2.8.

A disadvantage of Petri nets is that when they become large, they may become difficult to use [19]. Although their visualizations are very easy to grasp, even for people that have little knowledge in the field, it does not provide an efficient approach to formalize systems. For describing O2N components, Petri nets can become quite large, even when only simple state machines are involved. This is due to the fact that every system property needs to be translated into a system of places and transitions. A simple system can already result in a large amount of places and transitions. This also causes it to be difficult to see what is going on in a large Petri net. As it is expected that O2N models can become quite complex, especially when the interaction between multiple O2N components is modelled. Therefore, using Petri nets may not be the ideal solution for modelling O2N.

To reduce some of these complexities and to add other functionality, some other forms of Petri nets have been developed. Multiple variations on Petri Nets exist, of which some are described below. These variations all take a different approach with Petri nets to solve different kinds of problems. An open (Petri) net, for example, can be seen as an incomplete Petri net, that requires at least one other open net to become a complete Petri net. Open nets can be very well used to model different concurrent processes. An open net is modelled as a Petri net with interfaces to which one or more additional open nets can connect. With open nets it is possible to, for example, create an open net for a client and an open net for a server. When these are then connected they form one closed Petri net that describes the behaviour of both systems together [20] [2]. Other notable variations of Petri nets are timed Petri nets, which add timing constraints to the Petri net by using invariants on transitions and places to indicate timing, and coloured Petri nets, which uses 'colours' to represent different types of data.

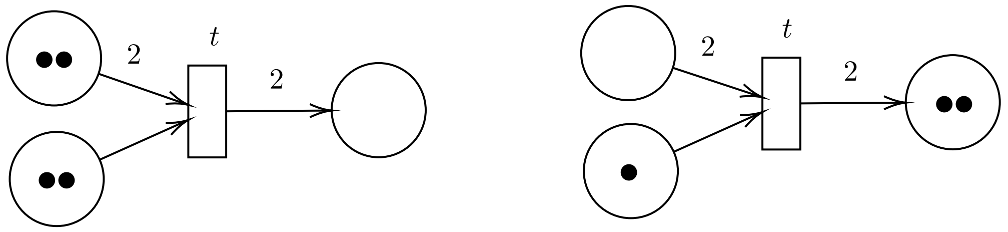


Figure 2.8: Visualization of the Petri net firing rule. There are two places connected to transition t, and one output place. t can only fire when there are respectively 2 and 1 tokens in the input places. When this requirement is satisfied, the transition t will occur and 2 tokens will be generated at the output of the transition. The top image shows the initial marking, which is in this case the state before the transition. The bottom image shows the next marking which is the state after the transition. After this transition the inputs of t have been exhausted and another transition is not possible.

2.4.3 Process algebra

Another form of analysing the behaviour of a system is process algebra. As the name implies, process algebra provides algebraic descriptions for concurrent processes. Process algebra refers to behaviour as *processes*. It regards a system as any entity that shows behaviour, and the behaviour of a system is the collection of events or actions that a system can perform [21, 22, 23]. Process algebra is described using *process expressions* and can be visualized using *labeled transition systems*, or *LTS* in short. Using such an LTS as visualization can show clearly the behavior of the system and can be useful when having to explain this behaviour. Another advantage is that it is relatively easy for computers to use an LTS representation to run verifications on it. The word 'algebra' indicates a mathematical syntax. Most process algebras have a very concise and simple syntax. A syntax that is common between most process calculi is the notation for choice. As an example is shown in Equation 2.1. A visual representation of the automaton described by the equation is shown in Figure 2.9.

$$P \overset{\text{def}}{=} a.X + b.Y.c.Z \tag{2.1}$$

This indicates an automaton where from a state P a choice (indicated by the $+$-operator) is present to either do a transition a to state X, or do transition b to state Y. Sequential operations are denoted by the .-operator, which is shown by the transition from Y to Z using action c. There are a lot of variants of process algebra which mostly have small differences, and thus have differences in syntax. Some popular variations are the Calculus of communicating systems (CCS) [24], the Algebra of communicating systems (ACP) [25], π-calculus [22], and Communicating Sequential Processes (CSP) [26, 27].

2.4.4 Timed automata

Timed automata [28] have similarities with process algebra, since process algebra can be described using automata. Timed automata obviously also have a notion of time by using invariants on both the states and transitions. Although timed automata do not (necessarily) have a mathematical representation like process algebra, it does have a graphical notation using labelled transition systems. Timed automata are formalized by first converting them to regular finite automata. This is done by extracting states from time regions. A challenge for timed automata are state space explosions, since converting timed automata to regular automata can result in very large automata to verify. For O2N it is not necessary to model strict timing, there is no absolute or relative unit that needs to be used to describe time to verify dialogues. What does matter for O2N is the order in which actions can be performed. Therefore, using timed automata provides not much difference as compared to process calculus. Timed automata can still be used for O2N, since time can simply be ignored and a regular automaton can be modeled instead.

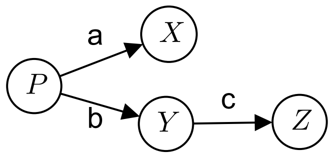


Figure 2.9: Visualization of the automaton described by Equation 2.1.

2.4.5 Overview of formal methods

In this section multiple formal methods have been described that can be used for formalizing distributed systems. Many of these methods serve as a background which is implemented in one or more tools. The actor model shows similarities with O2N-based systems, as an O2N component can be viewed as an actor. However, it is more a programming paradigm than a formal method. There are tools available that allow formal methods to be used on systems that use the actor model, such as $\mathcal{R}$ebeca. However, using the actor model does not say much about the structure of an interaction between multiple components, and by using the actor model there is no guarantee of the absence of deadlocks and/or livelocks. Therefore, it is expected that other approaches provide better value for this purpose.

The concept of Petri nets is easy to comprehend, but this simplicity has drawbacks when modelling complex systems. The simplicity of Petri nets can cause complex systems to look even more complicated. For complex components the visualization method of Petri nets may not suffice in giving insight in the system, which is a large disadvantage when analyzing these components. A possible solution for this is to use a more abstract variation, such as coloured Petri nets or open nets. However, when describing state machines it might be more suitable to stick with automata theory, which leaves process algebra and timed automata as more suitable background options for formalizing O2N components.

Process algebra is a promising approach to modelling O2N components. An advantage of process algebra is that they are mathematically and graphically expressive. Timed automata have the same advantages of process algebra in terms of graphical notation and computability by computers. They can also be nicely visualized using labelled transition systems. Therefore it appears that, although every model would be possible, both process algebra and timed automata provide the most solid basis to work from to formalize component interactions in O2N.

2.5 Formal background

This section describes some background details for model checking and temporal logic. Most of the concepts explained in this section are being used extensively throughout this thesis.

2.5.1 State-space

The set of all possible states of a system is called the *state-space*. The state-space is an important term for formal verification, since the size of the state space plays a large factor in the extent to which a system can be verified. When the state-space of a system is large it is required to use more computer memory and time for verification as compared to less complex systems. The state-space usually grows exponentially, since for example adding a simple boolean variable to any system already may double the size of the state-space. When a state-space increases too much then this is called a *state-space explosion*. The term is also used for a state-space that is simply too large to be verified. However, there are methods to reduce the size of the state-space, such as partial order reduction or by reducing the system to a smaller system that can replicate the same behaviour. Optimizations like these come with a performance cost, so using them is not always preferable. Algorithms for the optimization and analysis of the state-space of a system can be complicated and will not be described in this thesis.

2.5.2 Model checking

Model checking is a method of determining whether a (finite) abstract model of a system holds to a specification. A system that performs model checking is called a *model-checker* or a *verifier*. It can for example be used to determine whether a condition holds in every possible state of the system. How these conditions work varies for different tools, but common methods are Linear Temporal Logic (LTL) [29], Computation Tree Logic (CTL) [30], and modal μ-calculus which are all described later in this section. When abstracting a system to a formal model, there is a challenge to find the proper abstraction. If the abstraction has too much detail the state-space becomes too large. However if the abstraction has not enough detail, then the model checker may not find problems in the system since these problems are abstracted away.

2.5.3 Temporal logic

Temporal logic can be applied to a formal model of a system. Temporal logic is used to reason about properties that are expected to hold for the formal model, such that a model checker can verify whether these properties hold. Using temporal logic one can reason about the order in which certain events must occur, or to state that a property eventually must be satisfied. A formal model can be described by a transition system, in which every possible execution results in an *execution trace*. The model checker can analyze these traces for the given properties. There are two variants of temporal logic that are widely used: LTL and CTL, which are discussed below.

2.5.4 LTL and CTL

LTL formula's are propositional logic formulas, combined with connectives G (globally), F (Future/eventually), X (next), U (Until), and W (Weak until). LTL properties are properties that have to hold for the whole state-space.

- Propositional logic contains for example the $\wedge$ (AND), $\vee$ (OR), $\neg$ (negation) and $\rightarrow$ (implication) operators.
- G ϕ means that ϕ holds in every reachable state.

- F ϕ means that there always exists a reachable state where ϕ holds.
- X ϕ means that ϕ holds in the next state.
- ϕ U ψ means that eventually ψ will hold, but until that point ϕ must hold.
- ϕ W ψ means that ϕ will hold at least until ψ. If ψ never becomes true ϕ must remain true forever.

An alternative to LTL is CTL [30], which allows to specify properties that do not have to hold for all possible program executions, but only for one possible execution path. To do this, a distinction is made between path operators and state operators. Path operators in CTL are:

- A: a property has to hold for all paths starting in a particular state.
- E: a property has to hold for at least one path starting in a particular state.

By using path operators CTL is more expressive than LTL. However, not all combinations of path and state operators are possible, so there are still LTL formula's that cannot be expressed in CTL. Therefore it is hard to compare LTL and CTL. There is also a variant which combines the properties of both LTL and CTL (e.g. the state and path operators may be mixed in any order), which is called CTL* [31]. The properties that have to hold are then the same as for usual LTL formulas. However, if these have arguments then these must start with a path operator.

2.5.5 Modal logic

Modal logic has a lot of similarities with temporal logic. It is an extension of propositional logic using the box operator ($\Box$) and the diamond operator ($\Diamond$). $\Box\phi$ means that if from a state s a transition is made, the statement ϕ must hold for all states s' where $s \rightarrow s'$. $\Diamond\phi$ means that if from a state s a transition is made, the statement ϕ must hold for at least one state s' where $s \rightarrow s'$. The box and diamond operator are not the same as the Future or Global operator, as they only restrict the states to which a transition is possible from the state under verification. Temporal logic restricts the rest of the execution, without knowing the length of the trace. However, there is an extension of modal logic that does allow this, which is modal μ calculus.

Modal μ-calculus

In contrast to modal logic, modal μ calculus allows specifying that properties must hold in the next state. However, it also passes on the requirement of this property to the next state. This shows some recursive behaviour in which properties have to hold for an unknown time for follow-up states. There are two types: least fixed point and greatest fixed point. For least fixed point the recursion has to terminate at some point, while for greatest fixed point this never needs to happen. Because of this, the modal μ-calculus is very expressive and all logics like LTL, CTL and CTL* can be written as a modal μ-calculus expression.

2.5.6 Model properties

Safety

A safety property expresses that the system operates in a safe way, and will never reach a bad state. Therefore safety properties are often characterised by the phrase *"nothing bad will happen"*. If a safety property is violated, then a finite counter example can be given that shows the trace of the simulation that led to the violation.

Assertions are a form of safety properties. Assertions are constraints to variables that are placed within a formal model. They can be used to give constraints to a variable in a given location in the model. Usually, assertions can also be used in a real system. The difference between assertions in a real component as compared to a formal model of that component, is that the assertion will be verified for every state in the state-space, while in the real component these states may only occur by a given probability, and an incorrect scenario may thus be missed.

Another example of a safety violation is a *deadlock*, which is also called an *"invalid end-state"*. A deadlock occurs when the system is not able to do any transitions, while it is not in an end-state. This usually means that the processes are somehow waiting on each other to perform an action. If the system has no transitions while it is in an endstate, then this is not a problem and it is not considered a deadlock.

Liveness

In contrast to safety, liveness properties are characterized by the phrase *"eventually something good will happen"*. Therefore liveness properties check if a given state is eventually reached. this has the implication that, if such a property is violated, the counterexample is infinite. Liveness properties can be used to verify that an engine will eventually be turned on, or, if it is on, it will eventually be turned off.

Invariants are a form of liveness properties. When they are implemented using LTL they can be observed as global constraints to variables. Examples in LTL are `([] (x < 10) && (x > 0))`, which specifies that the variable `x` always must have a value between 0 and 10, Using an invariant, or `([] (y > z))`, which specifies that `y` always must be greater than `z`. It can be specified that a given variable should always have a value between the given boundaries for every possible state.

A *non-progress cycle* is an infinite cycle where continuously enters the exact same state. Not all non-progress cycles are necessarily incorrect behaviour. A livelock however is a non-progress cycle that is undesired. As an example, a system that returns to it's original state after it has performed a given task may have no progress, but still represents normal system behaviour which is not a problem. A system where a message the same message is repeated over and over again while it is ignored by the receiver is a non-progress cycle that could be a problem and then is a livelock.

Fairness

It is possible that a liveness property in a formal model is violated on a path that does not happen in reality, because the transition is enabled but never executed. In other words, the formal model allows something to happen infinitely often, but it never actually happens. When such a situation occurs the model checker may find issues in the formal model that never occur in real life. To prevent such a situation the notion of *fairness* is introduced. The word *fair* indicates that the formal model has to be fair to the real world. There are two notions of fairness. *weak fairness* means that if a transition is enabled for an unbounded time, the transition must eventually be taken. *strong fairness* means that if a transition is enabled *infinitely often*, then it eventually must be taken. The transition does not need to be enabled continuously, it just needs to be enabled repetitively. Fairness can be expressed as an LTL property:

$$F\ G\ enabled_t \rightarrow G\ F\ taken_t$$

Meaning that if eventually for all states a transition is enabled, then for all states eventually the transition must be taken. For strong fairness the LTL property is different:

$$G\ F\ enabled_t \rightarrow G\ F\ taken_t$$

This means that if, for all states eventually a transition is enabled, the transition must eventually be taken. The notion of fairness can for example be used to determine if a message is sent to a process that the message is eventually read. The model can therefore be verified that it does not ignore messages that are being sent. Fairness can be used in two ways. It can be used in LTL properties to ensure that liveness properties hold on all *fair* paths. This is very useful when performing liveness verification. It can also be used to determine whether a model *is* fair.

2.5.7 Formal tools

There are many tools available for formal verification. Some examples are UMC, SPIN, NuSMV/nuXMV, CADP, UPPAAL, TLA+, ProB, mCRL2, FDR4 and CPN. The existence of many different tools indicates that there are different appliances of formal tools. In this chapter it will not be elaborated on all the existing tools. The next two chapters describe the process of selecting a tool that will be used for this research.

3 METHOD

This chapter shows how this research will build upon the background by forming an approach on using formal tools to analyze and verify O2N based systems. First in section 3.1 an outline of the research is given. In section 3.2 an approach on tool selection is described. Then, in section 3.3 the approach on modelling one model is described, and in section 3.4 the approach to the modelling of interactions is shown.

3.1 General outline

To model behaviour of a component it is necessary to specify this behaviour first. The tool 'ComMA' provides this functionality through it's specification language. It is likely that Thales will use a modified version of ComMA if they decide to use this tool. However, since it is not known what changes will be made, the current version of ComMA is assumed for this research. To do this, Thales has provided this research with two ComMA specifications of components that are, according to Thales, representative for O2N components. These components will be discussed in detail in section 3.3.

To illustrate the next steps of this research the visualization of the ComMA use-cases (Figure 2.7) is extended and shown in Figure 3.1. It shows that instead of performing runtime observations, the ComMA specification will be converted to a formal model such that every possible system state can be analyzed. It is likely that the observations in the formal model contain less details than the runtime observations ComMA usually performs, since there may be variables that have unknown behaviour and cannot be observed in the formal model. On the other hand, the formal model can perform exhaustive verification, meaning that the properties that it can verify will be verified more thoroughly and it could potentially find problems that do not often occur on runtime.

When a formal model is generated from the ComMA model, it will be analyzed what properties of this model can be verified. The result of this research will contain: an overview of a set of tools that may be applied to verify O2N-based systems, a guide on how to model O2N-based components in the language of the most suitable tool, and an overview of verification possibilities in that tool. The overview of verification possibilities will be created in a process where each of these properties is simply tried to be implemented and verified. If no errors are found a simple error may be introduced in the formal model to demonstrate that the tool can find the error.

3.1.1 Some remarks on ComMA

The fact that a component can have multiple interfaces, and therefore be specified using multiple ComMA specifications, will be ignored throughout this research. There are two reasons for this. The first reason is that Thales has not yet integrated ComMA to a large extent, so not many real ComMA specifications are available. The second reason is that a single interface has to provide enough information for an interaction to be verified. This is because the behaviour of

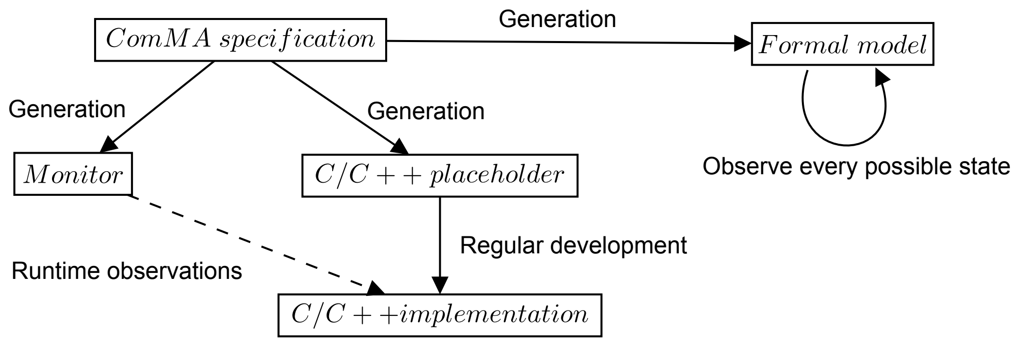


Figure 3.1: ComMA will be used for an extra use-case. Besides generating a C/C++ placeholder, also a formal model will be generated to verify the specified behaviour. If there is a problem with the specification, it can be found much earlier in the development process by analyzing the formal model.

the interface must be specified in the specification, and if there is external behaviour, this needs to be specified as well. Therefore, by limiting the amount of interfaces a component can have it is not expected that this will have a major influence on the verification of interactions.

3.2 Approach on selecting a suitable formal tool

A tool needs to be found that is most suitable for formal verification of an O2N component. In the background a list of existing tools is given, but more research is needed before a tool can be chosen. Therefore a selection of tools will be investigated further. Based on short descriptions of the tool, an overview of the features and by determining the purpose of the tool, three tools will be selected. The minimum criteria for the tools that will be selected are:

- **The tool supports multiple parallel processes that can communicate.** Communicating parallel processes are a significant characteristic of O2N-based systems. Since this research aims at the communication between such processes, the tool must allow the specification of models that show this behaviour.
- **The tool is based on Process Algebra or Timed Automata.** From the background research[1] it appeared that either process algebra or Timed Automata would be most suitable for modelling O2N.
- **The tool has visual support for the state machine.** Having a visual view of the state machine may help in understanding the behaviour of a system. This is useful for creating models and for finding problems in the model.
- **The tool is at least able to find deadlocks** Finding deadlocks is the minimum a tool should be able to find. Finding deadlocks is a key property that Thales has asked for. Having more options for verification is preferable.
- **The tool has a simulator to navigate through the state-space.** When analyzing the behaviour of a system it is very useful to view the behaviour in a step-by-step analysis. When a verifier finds an issue, it should be able to view the trace in the simulator. In such a case a problem can be more easily found and understood.
- **The tool is actively maintained.** A tool that is actively maintained has some advantages over tools that are deprecated. A major advantage is that, when a problem with the tool occurs, the developers can be contacted. It is also more likely that the tool will support

[1]subsection 2.4.5

newer operating systems and adopt new technologies. A tool that is actively maintained also suggests that the tool is actively used (why would it otherwise be maintained?), and when a tool is actively used it is more likely that problems will be found by other users of the tool as well. A disadvantage of an actively maintained system is that newer versions may introduce new bugs.

- **The tool supports the Windows, Linux and macOS operating systems.** Thales uses both the Windows and Linux operating systems. However, this research will be conducted on the macOS operating system. Therefore, all three operating systems should be supported.

3.2.1 Tool testing criteria

The three tools under consideration will be structurally tested for several properties, which are listed below. To check for these properties a simplified component will be specified. The simplifications are to prevent running into problems with specific O2N details, while staying true to the original challenges of formally specifying O2N components. For this research the focus lies on interactions, therefore a simplified component is designed that has the same communication principles (i.e. the command-state pattern) as a real O2N component. This component can change state on inputs and has to return messages back. However, for now the possible presence of infinite numeric values and strings are ignored. The simplified model that will be created is a model of an engine and an engine controller. The example of an engine is chosen because later an O2N component will be modelled with a similar but more complex structure. Figure 3.2 shows a state diagram of this simple engine. When modelling the engine example some properties will be given extra attention.

- Passing values between two processes.
- Publishing state after a command message (command state pattern).
- Publishing state after an internal change (command state pattern).
- Support complex message types.
- Visualizing the state-machine.
- Simulating and visualizing verification runs.

The most important feature is modelling the command-state pattern, since that pattern serves as a basis for the interactions between components. Supporting complex message types probably helps a lot with applying this pattern. The ease of use of the tool is also taken into account by reviewing the visualizations of the state-machine and simulations. With the simple engine the verification possibilities are not taken into account, as these are already well known from the background research.

After the engine example has been modelled in all three tools it can be reflected on how each tool satisfies the properties above. The tool that appear to be the most promising tool will be used for the remainder of this research: modelling real O2N components, and then modelling and validating the possible interactions for these components.

3.3 Approach on modelling a single component

3.3.1 ComMA specifications used for this research

Thales has provided two O2N interface specifications that will be used for this research. These interfaces are called *PeriodicTask* and *DriveControl*. According to Thales these two examples may in some cases display problematic behaviour that is also present in other (confidential)

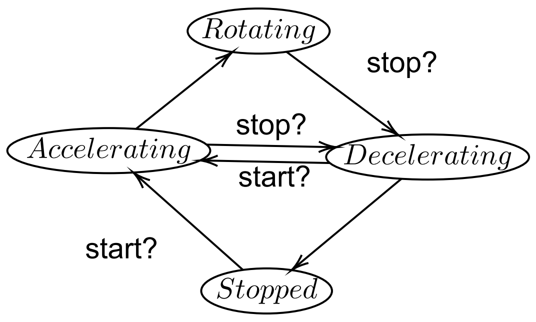


Figure 3.2: Simple engine example. The engine can be sent a desired velocity to which it will accelerate or decelerate. if the target velocity is reached the engine transitions to the 'rotating'-state, except if the target velocity is 0, then the engine transitions to the 'Stopped'-state. On every transition a 'state'-message is returned which represents the state of the engine.

Thales components. To answer the second research question, both the PeriodicTask and the DriveControl will be modelled while using a ComMA specification of these models as a base. In contrast to the previous section where a tool is being determined, here the details of O2N are important. While modelling the components it will be determined whether every ComMA feature can be translated to the language of the formal tool. If certain features appear to be difficult or impossible to translate to the formal model, then there will be looked for possible workarounds such that the behaviour can still be verified.

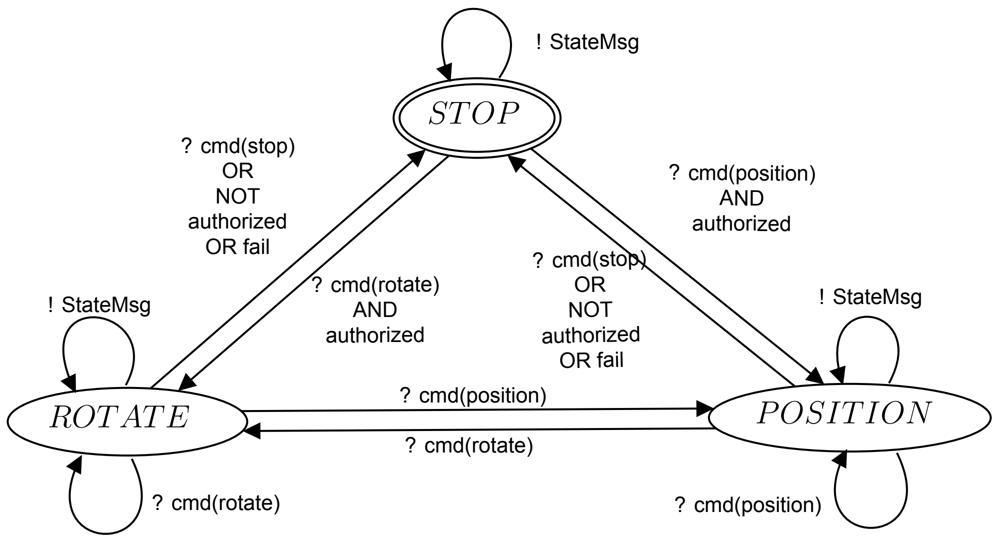
It is important for both the DriveControl and PeriodicTask to note that it is not only their state-machine that will be verified. The state machines of these components, as are shown in Figure 3.3 and Figure 3.4, are not the full automata that will be verified: it is only one part of the dialogue. For both the components a client will be created as well that communicates with them. The interaction between the two automata is what is of interest in this research.

DriveControl

The DriveControl is a component that, as the name implies, controls some sort of engine. The details of this engine is implementation-specific. Here it is important to realize that the ComMA specification specifies an interface that represents state of a component; it does not specify a component in itself. A DriveControl ComMA specification can therefore be (re)used on different components. By using this approach it is possible that two quite different components can still receive Rotation messages in the same fashion. Therefore, the presence of the DriveControl interface on a component enables a component to receive Rotation messages or other command messages that are used to control a drive. The exact implementation is left to the developers from Thales. Figure 3.3 shows a state diagram for the drive control, and it shows how the behaviour that is allowed by this interface. The engine can have three states, *stop*, *rotate*, or *position*. The stop state indicates that the drive does not turn. The rotate state indicates that the drive rotates with a given velocity. The position state does not take a velocity, but it takes a position instead to which the engine will turn. From both the *position* and *rotate* state it is always possible that the engine must stop due to either a failure, no authorization, or a stop command. When the drive is stopped, it can only be started when the rotation is authorized and when a Command message is sent. There are currently no issues known for the specification of the DriveControl interface. The full ComMA specification for the DriveControl can be found in appendix E.

Interesting properties to verify in the DriveControl are:

- **Deadlock**: Check for invalid endstates. Can the system end in a state where no other



! StateMsg
STOP
? cmd(stop) OR NOT authorized OR fail
? cmd(position) AND authorized
? cmd(rotate) AND authorized
? cmd(stop) OR NOT authorized OR fail
! StateMsg
! StateMsg
ROTATE
? cmd(position)
POSITION
? cmd(rotate)
? cmd(rotate)
? cmd(position)

Figure 3.3: Drive control state machine.

transitions can be taken? Is there a message, or a sequence of messages, that causes a deadlock?
- **Livelock**: Can the system end up in a problematic endless cycle with no progression?
- **Liveness**: Can the model reach the STOP state from all states? And does `authorized == false` or `fail == true` mean that this STOP state will be reached?

PeriodicTask

Another example provided by Thales for this research is the PeriodicTask. Like DriveControl, it has a descriptive name, since PeriodicTask simply performs a task in a periodic fashion. Every time the task has been performed the PeriodicTask returns a *performance* message. The task it performs can be quite different since it is very much based on the implementation.

The behaviour of a PeriodicTask is simple. The task is either being run periodically or not. This is represented by the `OnOffState` which is either *ON* or *OFF*. When the PeriodicTask is *OFF*, it can only be turned on. If the PeriodicTask component is *ON*, it can be either be turned *OFF*, stay on, or it can turn itself *OFF*. An example of the PeriodicTask turning itself off is a hardware problem that forces the system to shut down. A common problem for the PeriodicTask component is a delay in transmission. For example a livelock may occur whenever a client tries to turn on but, due to a defect, the PeriodicTask cannot turn *ON* and keeps shutting itself down. The state machine of the PeriodicTask is shown in Figure 3.4. The ComMA specification for the PeriodicTask can be found in appendix D. Interesting properties to verify in PeriodicTask are:

- **Deadlock**: Check for invalid endstates. Can the system end in a state where no other transitions can be taken? Is there a message, or a sequence of messages, that causes a deadlock?
- **Livelock**: Can the system end up in an endless cycle with no progression? An example would be where a client of the PeriodicTask would turn the PeriodicTask on while the PeriodicTask is turning itself off.
- **Liveness**: Can the OFF state eventually be reached from all states? and does `enabled == false` mean that the OFF state can be reached?

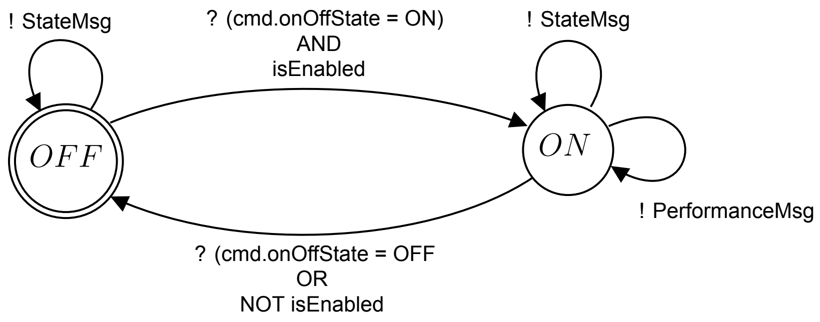


Figure 3.4: Visualization of the PeriodicTask state machine. State messages can still be sent in the OFF state to indicate that no periodic task is being executed. Performance messages can only be sent from the ON state when the periodic task is being executed.

3.4 Approach on verifying the interaction between components

When formal models have been created of components implementing either the DriveControl or PeriodicTask interface, it is aimed to formally validate interactions with these models. The other side of the interaction still needs to be modelled at this point. There are multiple ways to do this, and each approach aims to validate different properties of the model. Some possible approaches are discussed in this section. To illustrate the examples, for each approach a new component, C_{new}, is being added to a system that already contains C_a and C_b.

3.4.1 Approach 1: model a component in the system context

The key idea for this approach is to view C_{new} together with the set of other components that it communicates with. Using the *signature* it can be derived how this component communicates.Using this approach, a specification for every connected component must be available, and if those components also have other connections, specifications may be necessary for those too. This may result in a large state-space when multiple complex components are involved. A possible communication of C_{new} with other components is shown in the figure, where C_{new} is only directly connected to C_a. To do this, it is necessary to have the signatures and behaviour of all components that are involved. (left) However, it may be possible to disregard C_b (right) if it is of no relevance to the interaction between C_a and C_{new}. This could potentially reduce the state-space. However, there are difficulties that may arise when using this approach. For example finding what to do with unused interface inputs that change the state of the interface and/or component. Another downside is that the component under verification is not thoroughly verified, but only it's role within a larger system. When a small part of that system (for example another component) changes, then the component under verification can still appear to have problems. Because of this, a new verification run is needed when a component in this system changes.

Pros and cons

- **+** It shows the real system context.
- **+** Simple to implement.
- **+** As long as there are not too many components involved, the state-space is small.
- **-** For every new component you have to update the formal model and simulate again.
- **-** Components may show invalid behaviour when another component changes.
- **-** Some parts of C_{new} may be left unexplored by the verifier.
- **-** Every relevant component must have a formal specification (these are currently not available)

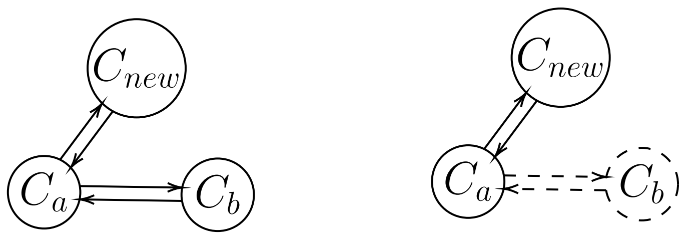


Figure 3.5: Left: Visualization of the first approach. The formal specification of new component C_{new} is connected with the formal specification of component C_a and indirectly with C_b. Therefore a formal representation of the whole system is being validated. **Right:** Visualization of pruning indirect connections with other components. It may be possible to disregard indirect components if they do not affect the behaviour of components that are directly connected.

3.4.2 Approach 2: Modelling the whole context into one component

The second approach works quite different. Instead of verifying C_{new} in the context of other existing components, this approach generates a dummy component using the signature from the same ComMA specification as C_{new}. This dummy component can be seen as a mirror component, as the signature of this component is mirrored: the inputs for C_{new} are outputs for C_{dummy} and vice versa. The dummy component is modelled in such a way that it tries to mimic the behaviour of other components. That means, every input and output of C_{new} is triggered by the dummy component like other components would do. By using this approach every relevant interaction with C_{new} is simulated. Also for this approach there are unknown factors. Although interfaces themselves may be deadlock free, is a network of component then as well deadlock free? What happens when three or more components are involved?

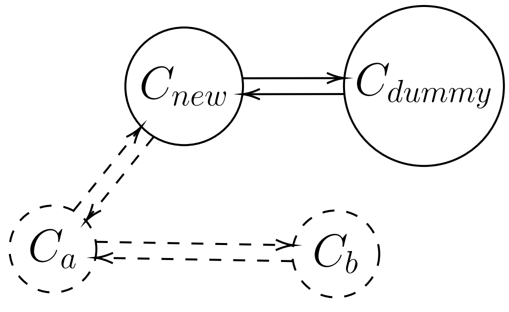


Figure 3.6: Visualization of the second and third approach of modelling a new component. The formal model of the component is connected to a dummy component instead of other existing components. Assuming the dummy component is correct, if C_{new} works with the dummy component, it is then also known that C_{new} will safely work with C_a and C_b.

Pros and cons

+ No other ComMA specifications are needed.
+ During verifications it is likely no irrelevant edge-cases will be found.
+ search shows that an interface is behaving as expected.
+ It can be shown that the interface is deadlock-free in every situation.
- Mimicking a context is hard to implement.
- if the context changes the dummy client needs to be changed as well.
- Information about the context is needed.
- With complex components still a chance of state space explosions.

3.4.3 Approach 3: Exhausting the state-space of the component

This approach is very similar to the second approach. But instead of trying to simulate the context of the component, it is simply tried to do an exhaustive search through all the states of C_{new}. The dummy component will send every possible message in every possible configuration to C_{new}. The problem with this approach may be obvious: a state-space explosion is likely to occur. When messages are simple this approach is probably feasible. However, a solution must be found when messages become more complex, for example when integers are involved.

Pros and cons

- **+** Exhaustive search, so it is not necessary to repeat verification when the context changes.
- **+** A component can be tested separately.
- **+** It is possible verify the behaviour for every message that is being sent.
- **+** Easy to implement: just try every possible form of message that can be sent.
- **-** Deadlocks may be harder to find, because that may require knowledge on the behaviour of the dummy client.
- **-** Large probability of state space explosions.
- **-** Irrelevant edge-cases in verification.

3.4.4 Choosing an approach

In Table 3.1 an overview is given of the different approaches, listing the most important properties on which an approach is chosen. Reliability indicates the completeness of the verification, if it does not catch all problems, or if it catches too many problems then it is less reliable. Robustness means that if another component changes, the same model needs to be verified again because the context has changed.

The third approach is likely to be the easiest to implement, since no other ComMA specifications are needed, and not even information about other components is needed. Therefore an exhaustive client can be created that exhausts the state-space of the component under verification. Approach 1 has the challenge that multiple ComMA specification are needed and that these somehow must be linked. Approach 2 has the large downside that the context needs to be modelled within a single client.

The effect on the state-space is never ideal. With approach 1 and 2 the state-space may be kept relatively small and a model of the real system context can be validated. However in model 3 a larger amount of states can be expected because the client is not limited to a given context: it just sends every kind of message, which, with complex messages can result in a large state-space.

All approaches allow for a multitude of verification options. It is expected that deadlocks, livelocks, invariants and other liveness properties can all be verified with all models. However, it is expected that deadlocks may be more difficult to verify in approach 3. From the table it is shown that approach 3 is the most promising option, but with a risk of having state-space explosions.

Like the verification options, the reliability is not a large factor when choosing an approach, since this works well for all approaches. For approach 3 it is expected that more problems may be found due to irrelevant edge-cases. In the other approaches this is less of a problem because in approach 1 and 2 more is known about other components. The reliability factor is in this case a direct consequense of more robustness.

Approach 3 is expected to be most robust because it exhausts the state-space. While it can find irrelevant edge cases; these cases may be relevant when another component changes. Approach 1 and 2 do not have this, although in approach 2 it is easier to anticipate to such cases.

From Table 3.1 it can be observed that approach 1 and 3 are the most favourable approaches. Although approach 2 finds the middle ground between the two approaches the expected difficulty and robustness make it a less favourable option. A downside for approach 1 however is that no other ComMA specifications are currently available to interact with PeriodicTask or DriveControl. Therefore, for this research approach 3 is chosen.

	Approach 1	Approach 2	Approach 3
Expected difficulty to implement	+-	- -	++
Expected effect on state-space	+-	+ -	- -
Expected verification options	++	+	+
Expected reliability	++	++	+
Expected robustness	- -	+ -	++

Table 3.1: Overview of different approaches. (++): very good, (+): sufficient, (+-): neutral, (-): bad, (- -): very bad.

4 TOOL SELECTION

This chapter describes the process of selecting a tool that will be used for the remainder of this research. First, in section 4.1 three tools are introduced that will be investigated further. In section 4.2 the tool mCRL2 is described, in section 4.3 the tool UPPAAL is described and in section 4.4 the tool SPIN is described. In section 4.5 an overview is given of these tools and it is decided which tool will be used for the remainder of this research.

4.1 Tools under consideration

Three tools are selected based on the criteria described in the method (section 3.2). The selected tools are: mCRL2, UPPAAL and SPIN. Although each of these tools fits the criteria, they still have differences and thus different advantages and disadvantages. By picking tools that have these differences it is possible to determine whether some approaches work better for O2N than others. The tools are all analyzed by creating an engine example as described in the method. Code samples are given to support the explanation of the model and tool.

mCRL2 (version 201908.0) is chosen because of the rich documentation and the concise language it is built upon. Since it is successfully adopted as engine for the Dezyne verification tool it has potential for industrial applications. A disadvantage of mCRL2 is the seemingly steep learning curve, especially when advanced use is needed.

UPPAAL (version 4.1.20-beta3) is chosen because of the insightful user-interface, which shows clearly what is going on between the processes. This may be very beneficial if developers from Thales need to work with the tool to gain more insight in the collaboration between their components. Disadvantages of UPPAAL may be that it does not offer enough features to properly model O2N components, and that the timing aspect in UPPAAL may not be useful for modelling O2N components. A beta version of this tool had to be used to support MacOS Catalina.

SPIN (version 6.5.1) is chosen because SPIN is designed for modelling communication protocols, and because the modelling language Promela has similarities with the C programming language. Spin is one of the first model checkers and it has evolved over the years with new functionalities, therefore it seems that SPIN has proven itself over the years. Disadvantages of SPIN may as well be the learning curve for Thales developers, and the visualizations are not always helpful.

4.2 mCRL2

mCRL2 [6] is a formal language created by the Eindhoven University of Technology and the University of Twente. It is based on the Algebra of Communicating processes (ACP) which is a form of process algebra. For mCRL2 an extended version of ACP is made that includes data and time. The mCRL2 toolset provides tools for describing behaviour through the mCRL2 modelling language, which has similarities with purely functional programming languages such as Haskell. To mitigate the steep learning curve of this tool, on the homepage of mCRL2 [32] a link is found where a lot of content, such as video lectures and documentation, is available that provides the necessary information. In mCRL2 the first step in the process is to create a specification. This specification is then 'linearized' to obtain a Linear Process Specification (LPS). This is an mCRL2 specification where all parallelism has been removed, and the whole behaviour of the system is modelled as a series of condition-action-effect rules that specify how the system as a whole react to inputs given its current state [6]. Most mCRL2 tools operate on this LPS. An LPS can be used to generate an LTS for visualization purposes. An LPS can also be converted to a *'parametric boolean equation system'* (PBES) such that it can be verified given a certain condition given by the user [33] using μ-calculus. Because mCRL2 applies μ-calculus it offers a lot of options when a model needs to be verified.

4.2.1 Engine model

Developing the engine example in mCRL2 appeared to be difficult. Although the syntax of mCRL2 is not complicated, it is a lot of work to model the engine properly. One thing that gives problems in particular is the return of state messages when a command message is received. The full source code of the engine example in mCRL2 is given in Appendix B.

Communication

The syntax of mCRL2 can be seen as a direct description of the state machine. First all possible *actions* (also called *transitions*) are defined. These actions can represent the transmission of values, such as a `Nat` which represents a natural number. More complex message types are also possible and can be denoted by the '#'-symbol. Examples are shown in Listing 4.1. It is also possible to create custom types with the `sort` keyword. This works similar to an enumeration in programming languages. In this example an action is created to indicate the engine is `done`, and actions are defined for `control` values and `feedback` values. The control messages represent the `command` messages from O2N, and the feedback messages represent the `state` messages in O2N. The complex message that contains a group of types is shown as demonstration, but is commented out since it is not used in the engine model.

```
% All actions in the system
act done;
    send_control, get_control, send_feedback, get_feedback, control, feedback: Nat;
%   complex_message: Nat # Nat # bool;
sort engine_state = struct STILL|ACC|DEC|ROT;
```

Listing 4.1: Actions and custom types for the mcrl2 engine specification. The action `done` has no type. The other defined actions are of type `Nat`.

In the example in Listing 4.1 it can be observed that the control and feedback actions have quite a few possible actions. There is an action for sending a message (`send_control`) by the client (which is in this case the engineControl), and receiving a message (`get_control`) by the server (which is the Engine). However, there is also a third message defined, called `control`. This action represents the synchronization between the `send_control` and `get_control` actions. This needs to be specified as well, and this part of the specification is shown in Listing 4.2. The `allow` statement indicates which actions are allowed in the running processes (here EngineControl and Engine). In the `comm` statement it is indicated what these actions represent: only `control` and `feedback` are legal actions, and these actions are a synchronisation of their respective `send` and `get` actions.

```
init
    allow({
            control, feedback
        },
        comm({
            send_control|get_control -> control,
            send_feedback|get_feedback -> feedback
        },
        EngineControl || Engine(STILL, 0, 0)
        )
    );
```

Listing 4.2: Defining which transitions are legal and if there are synchronizations between actions in mCRL2. The code block can be read as: *"The processes* `EngineControl` *and* `Engine` *are executed in parallel, and the only allowed actions are* `control` *and* `feedback`*, where* `control` *is a synchronization between* `send_control` *and* `get_control`*, and* `feedback` *is a synchronization between* `send_feedback` *and* `get_feedback`*"*.

Defining transitions

mCRL2 is based on ACP, which is a form of process algebra. Parts of the ACP syntax are visible in the mCRL2 specification syntax. The '+'-operator is the *choice*-operator, which is used for non-determinism, and the '.'-operator is the *sequential* operator, which is used for sequential operation of actions. Listing 4.3 defines the engine process. The process has parameters which represent the state of the process. In this case, there is an engine state, a velocity and a target velocity. The easiest way to understand what is going on here is to view it like a state; which does some actions and by doing so it traverses to a new state. The `Engine` process can do several actions based on it's state. It can receive a control message (the received value is stored in the variable `new_target`), or it can decelerate and decrease it's velocity, or accelerate and increase it's velocity, or it can reach a target velocity.

```
proc
    Engine(state: engine_state, velocity: Nat, target_velocity: Nat) = sum new_target: Nat

        % get control message and store it in target_velocity for the next iteration
        .    get_control(new_target) . Engine(state, velocity, new_target)

        % decelerating
        +    (velocity > target_velocity) -> Engine(DEC, Int2Nat(velocity - 1),
    target_velocity)

        % accelerating
        +   (velocity < target_velocity) -> Engine(ACC, velocity + 1, target_velocity)

        % target velocity is reached. Either still or rotating
        +   (velocity == target_velocity) -> (
                (velocity == 0) -> (
                    Engine(STILL, velocity, target_velocity)
                ) <> ( % else
                    Engine(ROT, velocity, target_velocity)
                )
        );
```

Listing 4.3: Engine process from the mCRL2 engine specification.

However, the specification shown above is not the full specification of the Engine process, since it is missing a critical part of the O2N interactions: it does not return state messages. Adding these state messages will result in a considerably more complicated specification. Listing 4.4 shows a small part of the specification with state messages. A new variable is introduced to the state of the engine, called `major_state_change` which indicates whether the state of the engine has changed between one of the four `engine_state` values (`STILL`, `ACC`, `DEC`, or `ROT`). Whenever this value changes it needs to be specified that only once this notification must be send. Because of the recursive notation of mCLR2 it is now forced to use this extra variable to send this state message in the next 'cycle'. Therefore, each transition that potentially leads to a new `engine_state` now becomes duplicated, and has to be divided in a version where the `engine_state` changes and a version where `engine_state` remains unaltered. When creating the simple engine a problem occurred when trying to generate the state-space of the model due to an 'unbounded recursion error'. Because mCRL2 already showed quite some challenges it was decided to not spend more time on this tool and instead continue the research by trying the other tools before the complete model of the simple engine in mCRL2 was finished.

```
    Engine(state: engine_state, velocity: Nat, target_velocity: Nat, major_state_change: Bool)
     = sum new_target: Nat

       % If there was a major state change, send the message that the state changed
       . (major_state_change) -> (
           send_feedback(velocity) . Engine(state, velocity, target_velocity, false)
       ) <> ( % else
       ...
       % Decelerating
       + (velocity > target_velocity && state != DEC)
               -> Engine(DEC, Int2Nat(velocity - 1), target_velocity, true)
       + (velocity > target_velocity && state == DEC)
               -> Engine(DEC, Int2Nat(velocity - 1), target_velocity, false)
       ...
```

Listing 4.4: Part of the engine process from the mCRL2 engine specification. Only the states for deceleration are defined. When comparing the deceleration definition with the deceleration definition from Listing 4.3 it can be seen that two versions of the deceleration state now must be defined to return state messages, increasing the complexity of the model.

4.3 UPPAAL

UPPAAL [34, 35] is a collection of tools developed by the university of Uppsala and the university of Aalborg. The tool is appropriate for "systems that can be modelled as a collection of non-deterministic processes with finite control structure and finite clocks, communicating through channels or shared variables. Typical application areas include real-time controllers and communication protocols in particular, those where timing aspects are critical" [36]. UPPAAL uses timed automata for model checking. UPPAAL consists out of three main parts: a description language, a simulator and a model-checker. With the description languague a (non-deterministic) model can be created. This model can be simulated, allowing the user to see through the execution of the processes. The model can also be model-checked. When a given query is not satisfied a trace is given which can immediately be put in the simulator to see what is going on in the system.

UPPAAL has many features. The user interface allows for easy visualization of the system, and the simulator shows the different subsystems while simulating. UPPAAL is also able to apply statistical analysis to these models, allowing to show charts and calculate probabilities. When a given query is not satisfied it can also show the (shortest) trace where this behavior occurs. Therefore it shows clearly the behaviour that gives the problem. The problematic trace can even be put in the simulator immediately to help users debug their system. These features make UPPAAL stand out as a very approachable tool in terms of usability.

4.3.1 Engine model

When modelling the engine, UPPAAL has shown great potential but also some drawbacks. Figure 4.3 shows the automaton that is manually created. The model is similar to the example in

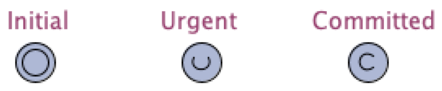


Figure 4.1: An initial, urgent and committed location in UPPAAL.

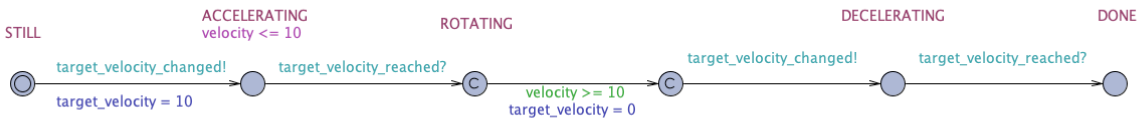


Figure 4.2: Engine Control modelled in UPPAAL.

Figure 3.2 and can easily be created using the user interface of UPPAAL. A client has been made for the engine as well, as is shown in Figure 4.2. Using UPPAAL the state machine of the engine could be modelled much like the initial example of the engine. Except from the visualizations shown in the figures, the only additional code are some definitions of the global variables in Listing 4.5. However, the definitions from Listing 4.5 also show the main problems with UPPAAL when applied in O2N-based systems.

The main problem is that data cannot be transmitted using the channels. A channel is just a notification, therefore, in the engine example there is a notification that either the `target_velocity` and/or the `velocity` has changed. This means that instead of sending data over the channels, a global variable representing the data is changed and the channel is used to indicate that the variable has changed. Although it works, it is not an elegant solution, since in this case it is not possible for two components to transmit on the same channel. Also the state-space gets larger than necessary. This problem is a clear indication that this is something that UPPAAL is not designed for. Another problem is that UPPAAL does not support grouped messages, or *structs*. Therefore, when it is needed to send a message with multiple variables over a channel, this must be done using a set of global variables that is changed by a process and that then notifies over the channel that the corresponding variables have changed.

```
int target_velocity = 0;
int velocity = 0;
broadcast chan target_velocity_changed, target_velocity_reached;
```

Listing 4.5: UPPAAL global definitions for engine example.

Although UPPAAL shows problems for O2N when it comes to communication, it performs very well at many other things, especially the ease-of-use when creating a model and when analyzing this model. Figure 4.4 shows the communication between processes. For each step in this simulation it can also be observed what the state is in each statemachine.

4.4 SPIN

SPIN [37] is a model checker. The name SPIN as actually an abbreviation for *Simple Promela Interpreter*. Promela is the modelling language used by SPIN. Using this language a formal model can be made which SPIN can verify using LTL logic and/or by using assertions. SPIN is designed for modelling and verification of multithreaded software, especially the C programming language, as SPIN provides tools to convert C programs to Promela models. However, it can also be used to verify other kinds of systems, such as distributed systems. The Promela

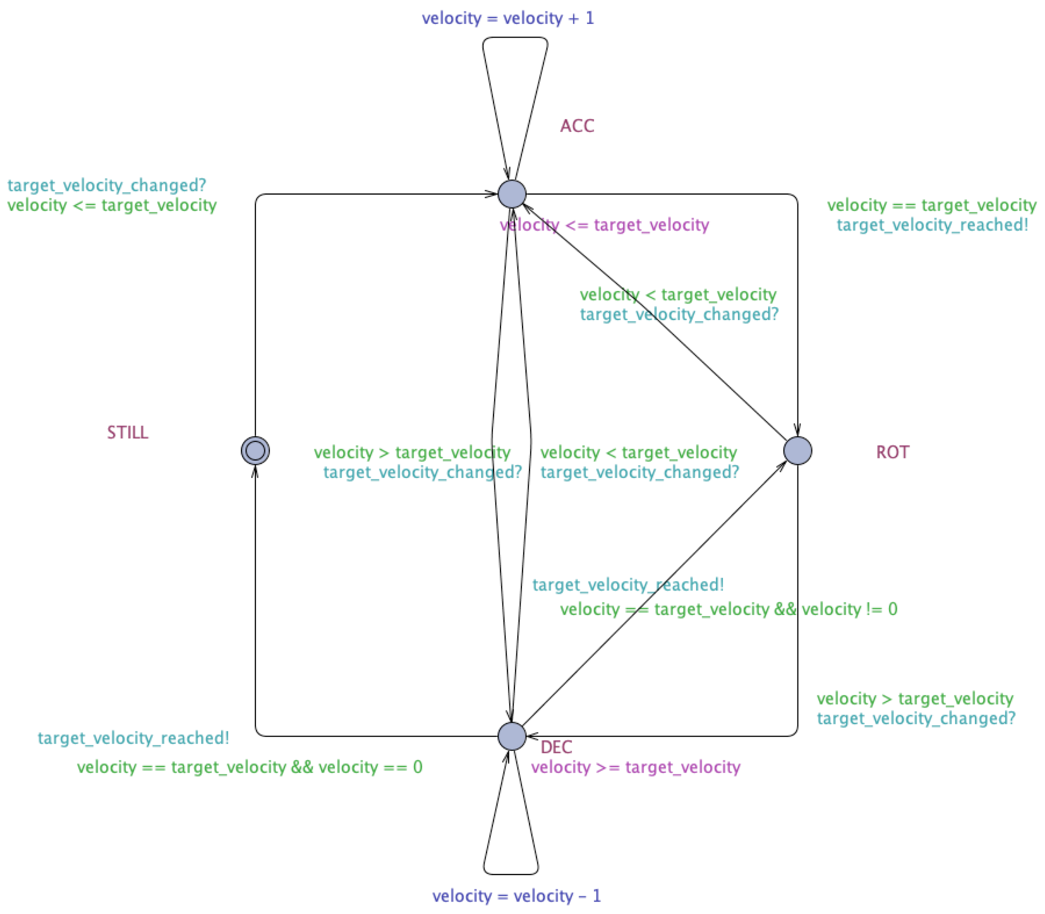

velocity = velocity + 1
ACC
target_velocity_changed?
velocity <= target_velocity
velocity == target_velocity
target_velocity_reached!
velocity <= target_velocity
velocity < target_velocity
target_velocity_changed?
STILL
velocity > target_velocity
target_velocity_changed?
velocity < target_velocity
target_velocity_changed?
ROT
target_velocity_reached!
velocity == target_velocity && velocity != 0
velocity > target_velocity
target_velocity_changed?
target_velocity_reached!
velocity == target_velocity && velocity == 0
DEC
velocity >= target_velocity
velocity = velocity - 1

Figure 4.3: Engine model in UPPAAL. UPPAAL uses colors to differentiate between properties of a state or transition. Red text indicates the name of a state. Pink text represents an invariant, which is a condition that has to hold in a given state (otherwise a transition must be taken). Green text indicates a *guard*, which is a condition that has to hold to allow the transition to take place. Purple is an *update* which is some code that is executed when the transition is taken. Finally, light blue is a *synchronization*, which will contain either a '!' or a '?' character to indicate respectively a *send* or *receive*.

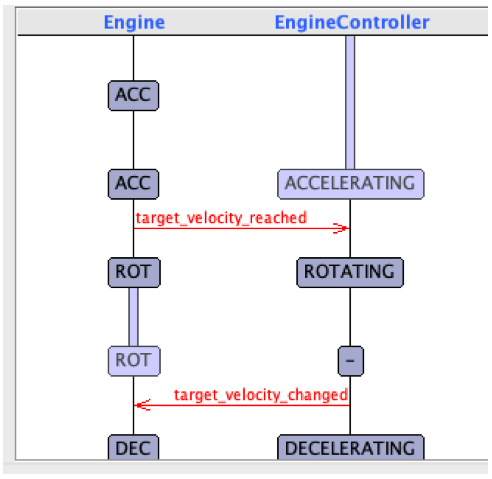


Figure 4.4: Partial screenshot of an UPPAAL simulation. Vertically, for each timesteps the state of the process is displayed (in this image the engine is first accelerating for two timesteps, then rotating, and after that decelerating). Communication between processes is indicated using a red arrow.

language is loosely based on Dijkstra's guarded command language notation and it is also influenced by CSP for I/O operations [37]. SPIN initially provides a command line interface. SPIN can parse Promela files (.pml) and show the state-space as automata using graphviz (dot) [38]. It can also use the verifier to verify the model defined in the Promela file. The command line interface will indicate whether the model has been parsed correctly, the verifications hold, and other information, such as size of the state-space, memory usage and other properties such as safety checks to determine if the program is free of deadlocks, or liveness to detect if states are unreachable. Although SPIN is initially a command line tool, there are multiple graphical interfaces available to make it easier to work with spin, such as iSpin, which is the default that is installed together with spin, or jSpin [39]. Model checking in SPIN can be done using LTL statements and/or assertions in the Promela code. When the Promela code is interpreted the LTL statements are converted to SPIN "Never"-claims. The LTL statements can be placed in the Promela file as well. However, only one verification can be run at a time, so to check for multiple verification queries it is required to run the simulation multiple times with a different verification query.

4.4.1 Engine model

Modelling the engine in Promela was achieved without many problems. The channels of SPIN proved to be very useful for mimicking the message structure of O2N. A downside of using SPIN is that the visualizations of SPIN can sometimes be hard to read. This section shows an overview of the Promela model. The full Promela specification of the engine can be found in Appendix C.

Syntax

A Promela program consists of processes, message channels, and variables. Process are global objects, and variables can be declared either globally or locally within a process. Processes specify behaviour, while channels and global variables define the environment in which the processes run. In Promela there is the notion of executability, and the execution of a process waits until the statement can execute. Listing 4.6 shows two lines of Promela code which are equivalent to each other due to this concept.

```
    while (a != b) skip
    (a==b)
```

Listing 4.6: Promela executability example.

The *Atomic* keyword indicates that a sequence of statements within the curly braces is executed as one indivisible unit, and does not interfere with any other processes. Promela also has the notion of channels, which are used to model the transfer of data from one process to another. Repetition in Promela is done using the 'do' keyword. The do-keyword repeats the statements in it as long as they are executable. Otherwise it blocks until at least one of the statements are executable. Breaking out of a do-loop can be done using the 'break' keyword or using a *goto*.

Communication

First some global definitions are specified, which is shown in Listing 4.7. There is a global definition for a message type for the engine state (*STILL, ACC, ROT* and *DEC*), and there are also global definitions for the message channels. A control channel is defined that transfers an integer which represents the desired velocity of the engine. The '[0]' means that there is a buffer of 0 indices in the channel. This is called a *rendezvous channel* (sometimes abbreviated to *rv channel* or *rv chan*). A rendezvous channel has some important differences with buffered channels. Most importantly is that a rendezvous channel is synchronous (to send, there has to be a receiver available) while a buffered channel is asynchronous (to send, there has to be place in the buffer, and to receive, there has to be *something* in the buffer). For this simple engine, synchronous communication will suffice. The engine can also reply to control with *state messages* which in this specification is called the *feedback_channel*. This channel transmits a more complex message that consists of an `mtype:EngineState` and two integers.

```
mtype:engineState = { STILL, ACC, ROT, DEC}  // Define engine states

// Set up global channels
chan control_channel = [0] of { int };
chan feedback_channel = [0] of { mtype:engineState, int, int }; // state, target_vel, vel
```

Listing 4.7: Promela ”simple engine” global definitions.

An alternative method of specifying the complex message in the feedback_channel is shown in Listing 4.8. This shows that Promela is able to group variables together, which is a very useful tool to have when modelling O2N components. The methods from Listing 4.7 and Listing 4.8 are furthermore similar and there is no difference in behaviour.

```
typedef FeedbackMessage {
    mtype:engineState engineState;
    int target_velocity;
    int velocity;
}

chan feedback_channel = [0] of { feedbackMessage };
```

Listing 4.8: Promela global definitions with a ’typedef’ instead of separate variables.

Behaviour

In Listing 4.9 a part of the behaviour of the engine is shown. The `do` statement represent non-deterministic repetition. When a message is received a new state is being determined: if the velocity is lower than the target velocity the engine must start accelerating. When the velocity is too high, the engine must decelerate. If the target velocity is reached it is checked if the velocity is 0 or not, to indicate that the engine is either rotating or still. The new state is temporarily stored in a variable called *new_state*. This variable is compared with the *state* variable: if it has changed, then a message is send over the feedback channel. The `atomic` keyword is an indication for SPIN that the content after the brackets has to be seen as a single step. This means that, in the engine example the new state is being set and this is being communicated at the same time. Therefore, SPIN will not consider a moment that the new state is being communicated but not yet changed, or that the state has changed but not yet communicated. In this example, whenever the state changes, this is communicated.

If-statements in Promela work different as compared to do-statements, but they are also differences with if-statements in conventional programming languages. The if-statement can have multiple branches, and if more than one guard statement is executable, one of them will be selected non-deterministically. If none of the guards are executable, the whole if-statement will block and the process will wait until a branch can be taken. It is possible to use the `else` keyword, which is true if at that point no other statement is executable. To make sure an if-statement does not block in Promela a branch with `:: else -> skip;` could be added to just continue executing if no branches can be taken.

```
:: control_channel?target_velocity -> {
    new_state = state;
    if
        :: (velocity < target_velocity) -> new_state = ACC;
        :: (velocity > target_velocity) -> new_state = DEC;
        :: (velocity == 0) -> new_state = STILL;
        :: else -> new_state = ROT;
    fi
    if
        // if the state has changed, we notify this to the EngineControl
        :: (state != new_state) -> atomic {
            feedback_channel!new_state,target_velocity,velocity;
            state = new_state;
        }
    fi
    // other statements
}
```

Listing 4.9: Specification of the behaviour of the simple engine in Promela.

Visualizations

For this research the default interface ’ispin’ has been used. Although SPIN is a command-line tool, ispin is included in the installation as well. iSpin is an interface that generates a command for SPIN and helps to display the output of SPIN. It can provide an overview of the state machine of processes, and it can display simulation traces. However, iSpin is not the most insightful user interface. An example is shown in Figure 4.5, where the state machine of the simple-engine and the simulation view of iSpin is shown.

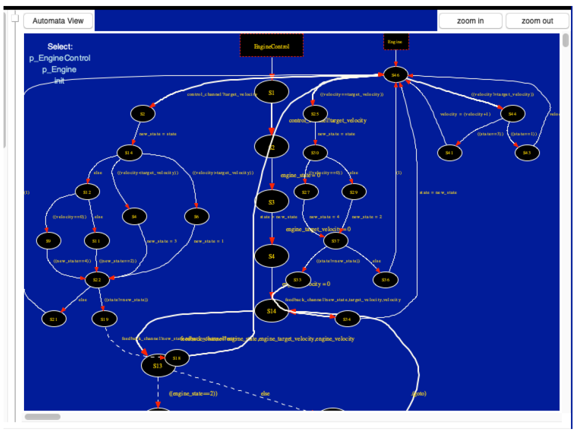
4.5 Conclusion on tool selection

The results of the tool selection can be observed in Table 4.1. The table also includes a short overview (above the dashed line) of features that where known while selecting these three tools. These features are described in section 4.1. Under the dashed line the properties that are described in the the method (section 3.2.1) are shown.

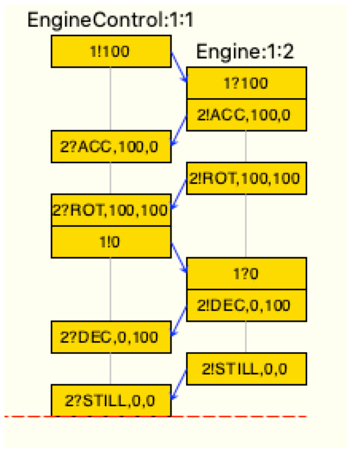
The syntax of mCRL2 is somewhat similar to that of functional programming languages. Where a functional program is a sequence of functions, in mCRL2 processes are defined as sequences of actions. within such a process description it is not possible to define variables. Therefore, state has to be passed using parameters. Effectively this gives a more intuitive approach to defining the state-space, since changing a parameter is equivalent to changing the state of the process. Therefore this approach stays true to the approach of doing a transition to another state. Also, having this ’functional’ approach also shortens the length of the specification. But, it also requires more thought when creating a formal model. Implementing simple things like loops and counters can be difficult due to recursion. This is a downside of mCRL2, as it makes it difficult to implement the command-state pattern.

Something that is very nice about the UPPAAL models is the graphical interface. The state machine from the engine, for example, is easily seen and is easy to understand from the graphical interface. However, passing values is more difficult, as communication channels do not allow parameters to be sent with them. Instead, it is necessary to have global variables shared by both processes, and use the channels to indicate that the global value has been updated. Therefore, also for UPPAAL applying the command-state pattern is difficult.

The programs created in SPIN where considerably larger. The imperative syntax of Promela has the advantage that it is easy to reason about the specification. While creating the models



Automata view



Simulation view

Figure 4.5: Left: Visualization of the automata view in iSpin. The visualization of the EngineControl state machine is visible as well. This visualization is hardly helpful, since the text is hard to read without zooming in. **Right:** Visualization in iSpin of a random simulation for the simple engine. In each yellow box the communication is shown which shows the Promela syntax for sending, but instead of variable names this example shows only the variable values, making the visualization hard to use. Therefore the simulation does not show the name of the channel. Instead, the name of the channel is represented by a number: `control_channel` is represented with 1 and `feedback_channel` is represented by 2. The upper yellow box can be read as: *"The* `EngineControl` *process sends a* `target_velocity` *with value 100 over channel* `control_channel` *to the* `Engine` *process".*

in SPIN it was noticed that there was a lot of support available online, which sometimes was of great help. Although SPIN does not stand out at the documentation and verification options like mCRL2, or has such a nice graphical user interface, it does stand out when it comes to applying the command-state pattern. The channels that are present in Promela are intuitive to use and they are actually quite similar in notation to the ComMA specifications.

To summarize, mCRL2, UPPAAL and SPIN all clear advantages and disadvantages compared to each other. However, for O2N it appears that SPIN is the most suitable tool. This is due to the fact that SPIN offers the most functionality that are useful when modelling O2N. In this research, where the interactions between components are being modelled it is important to have proper communication tools and this is what SPIN does better than UPPAAL and mCRL2. Therefore, for the continuation of this research SPIN is used as formal tool.

	mCRL2	UPPAAL	SPIN
Verification options	++	+	+
User interface	+-	++	+-
Documentation	++	+	+
Passing values between two processes	+	+-	++
Publishing state after a command message	-	+	++
Publishing state after an internal change	-	+	++
Support complex message types	+	- -	++
Visualizing the state-machine	+	++	+-
Simulating and visualizing verification runs	+	++	+-

Table 4.1: Overview of findings on tool selection (++): very good, (+): sufficient, (+-): neutral, (-): bad, (- -): very bad. The results are separated by a dashed line: the results listed above the dashed line are derived from section 4.1 and is knowledge that was known before the simple engine was modelled, while the results under the dashed line are the properties described in the method (subsection 3.2.1), and their respective results are found by modelling the engine example in the tools.

5 IMPLEMENTATION

This chapter describes the implementation of the PeriodicTask and DriveControl models in Promela. In section 5.1 the modelling of communication channels is described, which is a part of the implementation used in both PeriodicTask and DriveControl. In section 5.2 the process of converting ComMA specifications to Promela specifications is described. In section 5.3 the basic test setup is shown, which is the main file that combines the models with verification clients to test these models. Section 5.4 describes the implementation of these verification clients.

5.1 Modelling communication channels

To properly model the communication between O2N components it is necessary to also properly model the buffers between processes. Embedded in the Promela language are two types of communication between processes: Rendezvous communication, which is synchronous and does not have buffer, and buffered communication, which is asynchronous. With buffered communication there is an option to do a 'normal send' or a 'sorted send' operation when pushing to the buffer. In case of a 'sorted send' the messages in the buffer will be sorted on numerical value. However, there are more options on how buffers can be used in O2N. In Promela an important type of buffer that is used in O2N is not available by default. This section describes the buffer that is needed for O2N and describes the implementation of this buffer in Promela.

When modelling communication between processes it is important to think about the details of this communication, since sometimes a communication method may be used that is not considered a buffer in O2N, while in Promela it must be modelled as a buffer anyway. This is due to the fact that in O2N each component has a polling thread that receives incoming messages. When a new message comes in, the message is temporarily stored until the process has finished its current task. This is an intrinsic feature of a reactive architecture. The temporary storage must be seen as a buffer with a size of 1. To give another perspective on the same behaviour: when a publisher in O2N sends a message, the current process does not block. Instead, the message is handed over to the middleware and the process continues. This already is an indication that the components are fully asynchronous, and this asynchronous behaviour is only possible when buffers are used.

5.1.1 The problem with buffers

When modelling buffers a problematic situation may occur. It is possible that buffers are being completely filled. When full, a standard Promela buffer will either block new messages from the transmitting process, or it can be set up to discard new messages using the `-m` option when analyzing the model in SPIN. However, this behaviour does not always equal the behaviour of O2N communication. For example the variable stored in the polling thread of a component: when a process receiving the variables is slow, and another process is sending messages fast, the variable in the polling thread may be overwritten. Therefore, this needs to be regarded as a

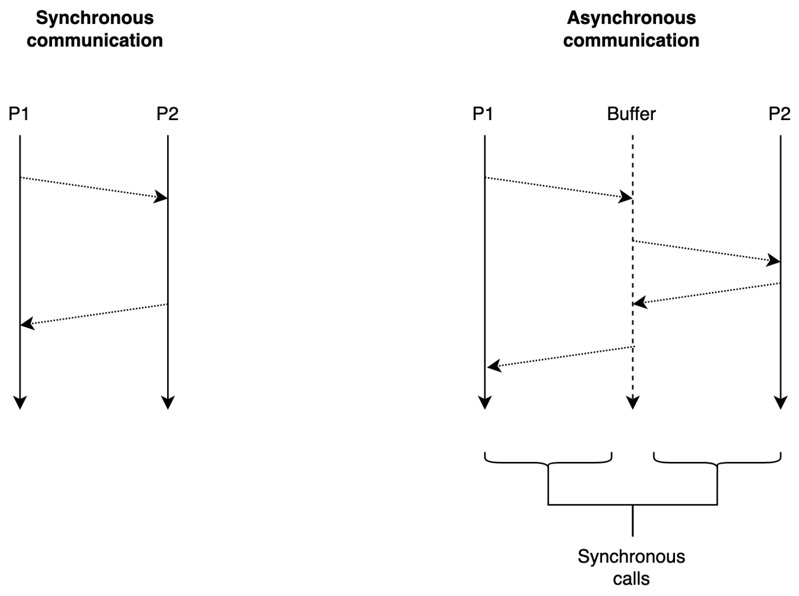


Figure 5.1: Buffer visualization. **Left:** Synchronous communication between two processes. **Right:** Asynchronous communication between two processes. The asynchronous communication can be made up of two parts that are synchronous, where the buffer in the middle stores the message, allowing for asynchronous message passing. Since synchronous message passing is available in Promela it is possible to build custom buffers using two synchronous channels.

buffer that, when full, discards the oldest messages in the buffer and still stores the new messages. Using incorrect buffer protocols can result in serious problems. For example in a system that can be turned either on or off using a buffer that has one entry. If this system is receiving two messages, first a message that says it must turn on, and then immediately a message that indicates it must turn off, what must it do? Although it may depend on the system, it seems reasonable that the system does not have to turn on at all. If the newest message is being discarded then the 'off' message will never be processed. If the oldest message is discarded, then the 'on' message will never be processed. Determining which buffer is preferable depends heavily on the system itself. However, for the PeriodicTask and DriveControl components described in this chapter a buffer is needed where oldest messages are being discarded if the buffer is full. Such a buffer is not present in the Promela language, therefore it has to be built manually.

5.1.2 Creating a custom buffer implementation

A buffered channel can be simulated using a separate process and two synchronous channels, as is visualized in Figure 5.1. The separate process is used to store the values and therefore allow asynchronous communication. The workaround is based on the concept of a circular buffer, which is a buffer of a known capacity where the locations in memory of the variables in the buffer are not moving. Instead, a location is stored where data should be read and where data should be written. An implementation in Promela is given in Listing 5.1. This example represents a buffer that stores 5 integers, however the buffer can be changed to contain any number of values by changing the `BUFFER_SIZE` variable. The buffer can also contain variables of any type by changing the `int` definitions on line 4,5 and 9. Like Promela buffers, it can also contain sets of types, but in that case this example needs to be altered a bit more: also the send and receive operations of this buffer need to be updated, and more `internal_buffer` variables are needed (one buffer for each type in the set). Using this buffer manually requires some work but it can easily be automated.

```
#define BUFFER_SIZE 5;

// Define both the transmitter and receiver channels. In this case we use Integers.
chan tx = [0] of {int};
chan rx = [0] of {int, bool};

active proctype CircularFifoBuffer() {
    int entries, head, tail = 0; // Variables use to determine where to write and/or read data
    int internal_buffer[BUFFER_SIZE]; // Internal buffer(s) where the data is stored

    // this do-loop is annoted with an ”end_”-label, such that SPIN does not consider this
    state an invalid end state (deadlock)
    end_circular_fifo_buffer: do
        // Put value in buffer. This is always possible.
        ::  tx?internal_buffer[head] -> atomic {
            head = (head + 1) % BUFFER_SIZE;
            entries = ((entries < BUFFER_SIZE) -> entries + 1 : entries);
            tail = (head+BUFFER_SIZE-entries) % BUFFER_SIZE;
        }

        // Read from the buffer and remove the entry,
        // this is only possible if the buffer is not empty.
        :: rx!internal_buffer[tail],(entries > 0) -> atomic {
            entries--;
            tail = (head+BUFFER_SIZE-entries) % BUFFER_SIZE;
        }
    od
}
```

Listing 5.1: Promela implementation of a FIFO buffer that discards oldest messages. This example shows a buffer that stores integers, however this could be any set of message types.

First of all, the two rendezvous channels are created, which are called `tx` (meaning *transmit*) and `rx` (meaning *receive*). Note that the whole buffer is created in such a way that the variables make sense when using the buffer. This means that the `tx` and `rx` variables have an opposite meaning within the definition of the buffer. For example, a client sends a message using the `tx` channel and the buffer receives these messages on the `tx` channel as well. Therefore, the name makes sense for the client. The `rx` channel also contains a boolean. This is a side effect of this buffer implementation which will be discussed soon.

The `CircularFifoBuffer` process is annotated with the `active` keyword, which indicates that one process of this type is always running and does not need to be started separately. Within the process, the behaviour of the buffer is described. An internal array called `internal_buffer` stores the values while `head` and `tail` point to an index in this array. The variable `entries` keeps track of the amount of values stored in the array: it increments on every write and decrements on every read. When a message is received from a client this message must be stored in the buffer. In this buffer it should always be possible, even when the buffer is full. The message will be inserted at `internal_buffer[head]`, the `head` variable will move one index in the buffer, the amount of entries is incremented by one (with a maximum of `BUFFER_SIZE`) and the tail is incremented by one if the buffer is full, otherwise it will not increment and stay in the same place.

When a message is being read from the buffer, the buffer has to transmit this value. In that case the amount of entries is reduced by one and the tail will move one position in the buffer. However, here is a particular thing going on in the Promela syntax which is necessary for this buffer to work properly. The buffer can only send values if there are entries. Therefore, the send operation has to be conditional. In Promela, it is illegal to mix logic statements and IO operations (for example: `do :: ((entries > 0) && rx!internal_buffer[tail]) -> {...}` is illegal). This is because IO operations take time: the rendezvous transmitter has to propose a handshake with the receiver, during that time the other condition can change it’s value. Because of this it

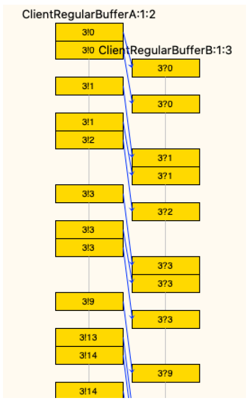
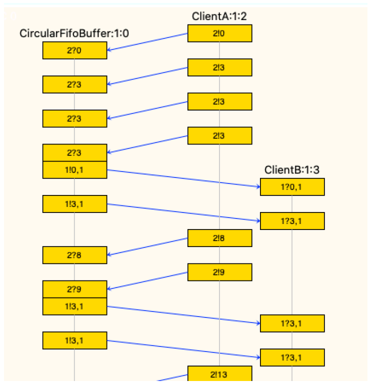



Figure 5.2: Two simulation outputs in iSpin using random traces. **Left:** Using a regular buffered Promela channel. **Right:** Using the custom circular buffer. When the custom buffer is used it can be observed that each interaction between the two client processes is shown as two communications that go through the buffer. Note that due to the random traces the communication is not entirely similar.

becomes unclear at what point in the execution the if-condition must be checked. Therefore, Promela does not support this. The manual proposes a solution by adding a `bool` to the channel. When sending, the outcome of the condition (in this case `entries > 0`) is sent together with the message (ensuring that the condition is checked at the same time as the other conditions). A receiver must then receive the message like usual, but appended with `,true`. This will filter out all messages where the condition evaluates to false, thus the receiver will only receive the messages when `entries > 0`.

In Figure 5.2 a comparison is shown between the two channels. It can be observed that the custom buffer has some disadvantages when it comes to viewing the behaviour in a simulator. This is not a very major problem however, it can still be quite easily deduced where message originate and where possibly incorrect behaviour could originate when using the simulator like this.

5.1.3 Handling non-progress cycles

Because the custom buffer can lose data when it is full, it introduces a non-progress cycle. Figure 5.3 shows that the problem occurs if a sequence of values with the exact same size as the buffer size, and the exact same values is repetitively put in the buffer. After all variables in the buffer have been replaced with the exact same value, then all internal buffer variables (`entries`, `read`, and `write`) match and the state is exactly the same. This only happens when data is lost, since the older variables are being overwritten. A possible mitigation for this problem is to indicate data loss with a progress label to indicate to the SPIN verifier that this is not a non-progress cycle that is interesting for this research. The implementation for this is shown in Listing 5.2. The progress label can be enabled and disabled using the `_PROGRESS_LABELS_ENABLED` macro.

```
#if _PROGRESS_LABELS_ENABLED
if
:: (entries == BUFFER_SIZE) -> progress: skip;
:: else -> skip; // make sure execution continues if the other condition is false.
fi
#endif
```

Listing 5.2: Optional progress label for buffers.

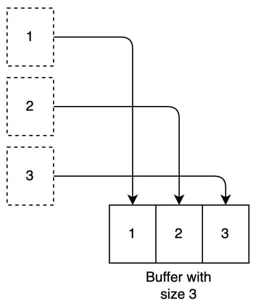


Figure 5.3: Visualization of non-progress cycles in the custom buffer implementation. In this example the maximum buffer size is chosen to be 3. The problem occurs when the exact contents of the buffer are placed in the buffer in the exact order.

Other progress label implementation for buffers

Another solution, which is more strict, would be to determine if a variable is overwritten with the exact same value. If that happens a variable can be incremented. If the variable $>=$ the buffer size, a non-progress label can be used to indicate that the verifier should not care about this. However, this is a significant change to the buffer implementation. section I shows a possible implementation. However, it should be noted that this example is not complete: Comparing the parameters to the original values in the buffers cannot be done directly if `typedef` statements are used. Instead, every member value of the `typedef` must be compared. This means that the second implementation is better for verification, but also much more work to implement. In this research the less strict, but much simpler implementation will be used, because when a livelock occurs in an interaction, and this livelock can occur without data loss, then the livelock will still be found. Therefore it is not expected that the more strict implementation will find many more livelocks.

5.1.4 Applying the buffers

It has been shown that O2N buffers can be modelled in detail in Promela. The custom buffer implementation will be used in both the PeriodicTask and DriveControl models. The used buffer size is set to 1, because this allows asynchronous communication, and the most recently sent value is present in the buffer. Having more variables present in the buffer is not expected to give different behaviour, but if the need arises to change the buffer size then it is easily changed by altering the `BUFFER_SIZE` variable.

5.2 Modelling ComMA specifications in Promela

Converting a ComMA model to Promela can be done using a given ComMA specification. The structure of a ComMA model has similarities with a Promela model and the conversion could be achieved using a systematic approach. A complete translation table to convert a ComMA specification to Promela is given in appendix F. This table is used to create the PeriodicTask and DriveControl models in Promela. There are parts of the ComMA specification where the conversion table is not enough to convert the ComMA specification to Promela. This is mostly in the signature, where more data is needed to determine the type of buffer to use, and it also is the case for applying invariants or LTL statements. This section describes the most important aspects of the conversion from ComMA to Promela. All examples are taken from the PeriodicTask models, however, some of these examples are heavily simplified to better illustrate the conversion process.

5.2.1 Types

A ComMA `enum` can be converted to a Promela `mtype`. The name of an `enum` can be taken over directly, which is useful in maintaining the similarity of the models. A ComMA `record` can be converted to a Promela `typedef`. The types it consists of can be taken over directly. A demonstration of the conversion is shown in Figure 5.4. Not every type can be converted however, for example `real` types cannot be represented by a type in Promela. An estimation could be given by using integers, however, this is not recommended. If the behaviour of the specification does not rely on such a value then it is best to keep the variable out of the model. There are some methods to add floats to a Promela model, by using embedded C code. This is also not recommended, because the use of floats indicates that a wrong abstraction is being used. Therefore it is best to have no behaviour directly rely on real values.

ComMA

```
enum OnOffState {
    ON
    OFF
}
record PeriodicTaskCommand {
    int id,
    OnOffState onOffState,
    real searchBudget
}
```

Promela

```
mtype:OnOffState = {
    ON,
    OFF
}
typedef PeriodicTaskCommand {
    int id;
    mtype:OnOffState onOffState;
    // int searchBudget; //
}
```

Figure 5.4: Type definitions in ComMA and Promela.

5.2.2 Signatures

Converting signatures from ComMA to Promela gives some challenges. The problem is that at this point assumptions have to be made about how these signatures are being communicated: channels can be either synchronous or asynchronous, and if they are asynchronous, the buffer can have different sizes and different behaviour when it is full. Therefore, to convert the `signals` and `notifications` properly it is necessary to think about the context of the component. Figure 5.5 shows the conversion from ComMA to Promela with rendezvous (synchronous) channels. However, for the model for PeriodicTask and DriveControl used in this research the more complex circular buffer implementation is used. These signatures can be found in appendix G for PeriodicTask and in appendix H for DriveControl. Aside from the buffer implementation, it is recommended to extend the name of the channel in Promela to contain a hint that it is a channel. In the example above the suffix `_chan` is used. This gives a clear indication that it is a communication channel and not another variable.

ComMA

```
signature PeriodicTask
signals
    PeriodicTaskCommandMsg(
            SystemHdr system,
            CommandHdr header,
            PeriodicTaskCommand Data)

notifications
    PeriodicTaskStateMsg(
            SystemHdr System,
            ResponseHdr Header,
            PeriodicTaskCommand Data)
```

Promela

```
// In the global scope
// Signature
chan PeriodicTaskCommandMsg_chan = [0] of {
    SystemHdr,
    CommandHdr,
    PeriodicTaskCommand};


chan PeriodicStateMsg_chan = [0] of {
    SystemHdr,
    ResponseHdr,
    PeriodicTaskCommand};
```

Figure 5.5: Variable definitions in ComMA and Promela using regular Promela buffers.

5.2.3 Behaviour: variables

The behaviour part of a ComMA specification consists of several different parts. In both DriveControl and PeriodicTask, first the definition of global variables is given. The `local` keyword used in Promela indicates that the variable is used in only one scope, and can be regarded as a local variable in that scope. Doing so reduces the size of the state space for SPIN as compared to a global variable. The difference between a global variable annotated with `local` and a regular local variable is that the global-annotated variable can be analyzed using LTL statements and never claims, while this is not possible with regular local variables. It is possible to leave out the `local` keyword in the Promela specification. However, by using it the state space is reduced and the model is therefore more efficient. In the example in Figure 5.6 most variables are annotated with `local` since they are only used in the PeriodicTask process. However, the `isEnabled` variable can be directly changed by the client and is thus used by two processes.

ComMA

```
variables
    bool isEnabled
    SystemHdr system
    ResponseHdr responseHeader
    PeriodicTaskCommand periodicTaskCommand
    DataHdr dataHeader
```

Promela

```
// In the global scope
bool isEnabled;
local SystemHdr system;
local ResponseHdr responseHeader;
local PeriodicTaskCommand periodicTaskCmd;
local DataHdr dataHeader;
```

Figure 5.6: Variable definitions in ComMA and Promela.

5.2.4 Behaviour: state machine

In Figure 5.7 the conversion from a ComMA state machine declaration to a Promela `proctype` is shown. Note that the example given in Figure 5.7 is taken from the PeriodicTask models, but the examples are simplified for demonstration purposes. In the comparison between ComMA and Promela it can be observed that the state declarations in ComMA are being modelled in Promela using labels, goto-statements, and do-statements. In Promela there is no `initial` keyword. However, the first label that is defined is the first state that will be reached by the process and is therefore the initial state. Using a do-statement it is ensured that a state will not be left unless a transition is being taken using one of the branches of the do-statement. The labels are extended with `STATE_`, this makes the labels more distinguishable. It also helps preventing name duplicates with other variables or keywords. In this example, the transitions are simplified to show the relation between the state labels and the goto-statements. The `transition do` keywords mean in this case that transition can be taken at any moment. However, transitions can be more complicated in the real models, and are thus discussed next.

ComMA

```
machine StateMachine {
    initial state OFF {
        transition do:
            next state: ON
    }
    state ON {
        transition do:
            next state: ON

        transition do:
            next state: OFF
    }
```

Promela

```
proctype Component() {
    STATE_OFF:
    do
        :: goto STATE_ON;
    od
    STATE_ON:
    do
        :: goto STATE_OFF;
        :: goto STATE_ON; // This is optional
    od

}
```

Figure 5.7: State machine definitions in ComMA and Promela.

5.2.5 Behaviour: transitions

In Figure 5.8 a transition is shown in both ComMA and Promela. These transitions are defined within a state: in ComMA by using a state declaration, or in Promela by using a label. The `transition trigger` in ComMA indicates that the transition can be taken when a message is being received. The statement is followed by the `signal` on which to receive the message, and the values of these messages are available in the parameters. In Promela this is done by receiving a message from the corresponding channel. The received values are stored in the given variables which need to be present in the scope. A `guard` can give extra conditions whether the transition can be taken or not. In Promela this is easily modelled using an `if` statement when a message is received. The `do` part then describes what has to happen when the transition is being taken. In this case, some variables are being assigned a value and a new state message is being sent. After that, the transition will take the process to the next state. In Promela this works exactly the same, although the syntax has some small differences.

ComMA

```
// in the 'state' and 'machine' scope
transition trigger: PeriodicTaskCommandMsg(
        SystemHdr system,
        CommandHdr header,
        PeriodicTaskCommand cmdData)

    guard: (isEnabled == false)
    do:
    response := REJECT
    cmd.onOffState := OFF
    PeriodicTaskStateMsg(
        system,
        response,
        cmd)
    next state: OFF

...
```

Promela

```
 // in the proctype scope, in a do statement
:: PeriodicTaskCommandMsg_channel_rx?
        received_system,
        header,
        cmdData,
        -> {
    if :: (isEnabled == false) -> {

        response = REJECT;
        cmd.onOffState = OFF;
        PeriodicTaskStateMsg_channel_tx!
            system,
            response,
            cmd;
        goto STATE_OFF;
    }
...
```

Figure 5.8: Transition definitions in ComMA and Promela.

5.2.6 ComMA properties that cannot be modelled in Promela

In this research the concept of time is not taken into account. It is only taken into account in what order transitions can occur, but not exactly how long these transition will take or other factors that require strict timing. Although ComMA does support the definition of strict timing constraints, this is not supported by Promela. Because it is not needed for this research, this is not a problem.

Real values, such as doubles and floats, cannot be modelled in Promela as well. A workaround for this is to extend the range of integers to simulate real values up to a given precision. However, it is not recommended to do so. Floating point values have been deliberately kept out of Promela, such that it is forced to use a different abstraction in the model [40]. If really necessary, floats can be used by using embedded C code in Promela.

5.3 The test setup

Both the PeriodicTask and DriveControl will be formally verified using multiple techniques. Because different properties will be verified it is likely that for some properties different clients need to be used to interact with the models. Therefore, to combine the model with the respective client a 'test-setup' file is created which is simply the entry point of the formal model. It is shown in Listing 5.3. The DriveControl/PeriodicTask models will be fully defined in, for each model, three files: `thales-types.pml`, `signature.pml` and `behaviour.pml`. The clients are created in a separate file as well, and depending on the client that is needed, the respective client can be included. For this research, a variety of clients had to be created. This is because some verification properties need different clients. In the next sections these clients are described. When the correct model and clients are included the models can be initialized which means that they both will be executed concurrently during verification. To keep all the configuration in one file it also contains the definition for the `_PROGRESS_LABELS_ENABLED` to enable or disable progress labels in the buffers.

```
#define _PROGRESS_LABELS_ENABLED 0

#include "thales-types.pml"
#include "signature.pml"
#include "behaviour.pml"

#include "client-base.pml" // or: "client-base.pml", "client-waiting.pml"

init {
    run PeriodicTask();
    run client();
}
```

Listing 5.3: Example of the PeriodicTask main file.

5.4 Clients for PeriodicTask

5.4.1 Base client (deadlocks)

For the PeriodicTask it is known that problems occur when the PeriodicTask component is quickly turned on and off repetitively. This is behaviour that can be analyzed using SPIN. To do so, a second process in Promela is created that repeatedly sends commands with the `onOffState` being set to either `ON` or `OFF`. The client is shown in Listing 5.4. Also the global variable `isEnabled` is being changed. Modelling this variable is interesting, since it's state alters the behaviour of the PeriodicTask, but when this variable changes is unknown. It could eventually be changed in another interface, or in the implementation of the developer (however, in that case it probably should still belong in the specification). Therefore the `isEnabled` variable can be altered at any moment. A visualization of this behaviour is shown in Figure 5.9. This client can perform all relevant communication with the PeriodicTask model, so this client will be used as a base for other clients.

5.4.2 Non-progress client

To find non-progress cycles a different version of the base client will be used. The base client can perform any transition because it has only one state from which it can perform all transitions. An endless loop is possible by toggling only one value. It can for example toggle the `isEnabled` variable infinitely, resulting in a non-progress cycle. This is behaviour of the client that is of no interest for finding non-progress cycles in the interaction with the PeriodicTask model. To prevent SPIN from giving an error on this client behaviour, `progress` labels are used in the implementation for the non-progress client. The implementation is similar to the base client except for the do-statement. The do-statement that is used in the non-progress client is shown in Listing 5.5. If a progress label is present in a cycle, SPIN will behave like there is progress in that cycle. Due to limitations in the Promela language each label must be unique within it's scope, therefore numbers are added to provide this uniqueness. Also, a label cannot be directly placed in a branch of a do-statement, so a workaround is applied using a guard that is always `true`. This workaround has no consequences for the outcome of the verifications.

```
// variables used for sending cmd messages
local SystemHdr test_SystemHdr;
local CommandHdr test_CommandHdr;
local PeriodicTaskCommand test_cmd;

// variables used for receiving state messages
local SystemHdr received_SystemHdr;
local ResponseHdr received_ResponseHdr;
local PeriodicTaskCommand received_cmd;

// variables used for receiving performance messages
local SystemHdr received_systemHdr_performance;
local DataHdr received_data;
local PeriodicTaskPerformance received_performance;

proctype Client() {
    isEnabled = true;
    test_cmd.onOffState = OFF;
    do
        :: isEnabled = !isEnabled; // global variable
        :: test_cmd.onOffState = ON;
        :: test_cmd.onOffState = OFF;
        :: PeriodicTaskCommandMsg_channel_tx!test_SystemHdr,test_CommandHdr,test_cmd;
        :: PeriodicTaskStateMsg_channel_rx?received_SystemHdr,received_ResponseHdr,
    received_cmd,true;
        :: PeriodicTaskPerformanceMsg_channel_rx?received_systemHdr_performance,received_data,
     received_performance, true;
    od
}
```

Listing 5.4: Base client for PeriodicTask. The variable declarations at the top are all annotated with the `local` keyword such that they can be used for verification using LTL.

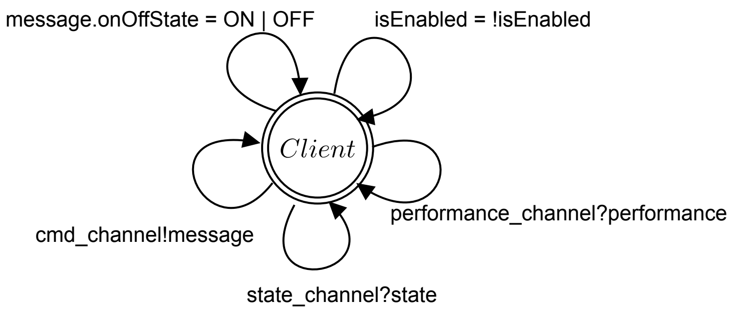


Figure 5.9: Base client for PeriodicTask. It has just one state, and it contains a message variable. This message variable can be changed in different transitions. Two transitions are different: one transmits the message, and another can receive state messages.

```
do
    :: true -> progress_0: isEnabled = !isEnabled; // global variable
    :: true -> progress_1: test_cmd.onOffState = ON;
    :: true -> progress_2: test_cmd.onOffState = OFF;
    :: PeriodicTaskCommandMsg_channel_tx!test_SystemHdr,test_CommandHdr,test_cmd;
    :: PeriodicTaskStateMsg_channel_rx?received_SystemHdr,received_ResponseHdr,received_cmd,
    true;
od
```

Listing 5.5: Different client implementation where each change of variable is indicated with a progress label, except the communication.

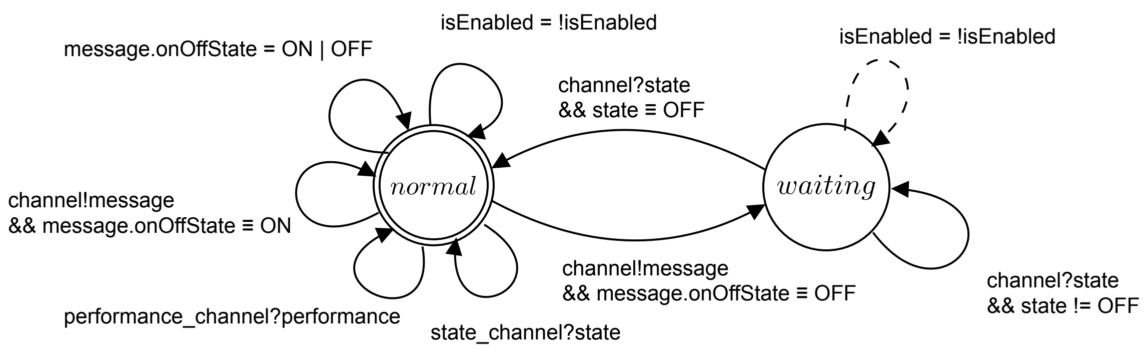


Figure 5.10: State machine of the waiting client. Like the base client, it can change the message that it eventually sends. However, when an OFF message is sent, the client will wait for a state message indicating that the PeriodicTask has indeed turned off. A detail is the presence of the `isEnabled` toggling transition in the `waiting` state. Enabling this transition in the client enables an infinite cycle in the waiting state, preventing the deadlock from being found. Therefore this transition is not present.

5.4.3 Waiting client

Another client that will be verified is the waiting client. It is a variation on the base client as well, but with one major difference: it has a different interaction with the PeriodicTask component. If the client wants the PeriodicTask to shut down, it will wait for a state message indicating that the PeriodicTask has shut down. The state machine of the waiting client is shown in Figure 5.10 and the implementation is shown in Listing 5.6. Using this client it may be possible to find different kinds of problems that may not be found in the base client.

5.5 Clients for DriveControl

5.5.1 Base client

To verify the DriveControl component a similar approach has been taken as with PeriodicTask, although messages that are sent in the DriveControl example are more complex than in the PeriodicTask component. Listing 5.7 shows the base client for the DriveControl component. It is visualized in Figure 5.11. Like the base client of PeriodicTask this client defines a do-loop that non-deterministically chooses to either transmit/receive a message or to change the contents of the message to be transmitted.

In this model also some global variables are present. The global variables are toggled at line 4 and 5 and are marked with a comment. These global variables are also present in the ComMA specifications and they alter the behaviour of the DriveControl component. These global variables represent data that is coming from another interface. The `authorized` variable is true if it is authorized that the DriveControl is turning. If `authorized` becomes false, the drive is not allowed to rotate. If the drive is rotating and `authorized` changes to false, then the drive has to stop. The `fail` variable represents the state of the hardware. If a hardware issue occurs then the circuitry triggers the `fail` variable to become true. In that case, the drive has to stop rotating as well. The assumption made in the Promela model is that both the authorization and the fail variable can be switched at any moment. The behaviour of the interface that changes these variables is unknown. However, in a real-world example it can be safely assumed that a hardware failure can occur at any moment, and authorization may change at any moment as well.

```
end_normal:
    do
        :: isEnabled = !isEnabled; // global variable
        :: test_cmd.onOffState = ON;
        :: test_cmd.onOffState = OFF;
        :: PeriodicTaskStateMsg_channel_rx?received_SystemHdr,received_ResponseHdr,
    received_cmd,true;
        :: PeriodicTaskCommandMsg_channel_tx!test_SystemHdr,test_CommandHdr,test_cmd -> atomic
     {
            if
                :: (test_cmd.onOffState == OFF) -> goto waiting;
                :: else -> skip;
            fi
        }
    od

    waiting:
    do
        :: isEnabled = !isEnabled; // global variable
        :: PeriodicTaskStateMsg_channel_rx?received_SystemHdr,received_ResponseHdr,
    received_cmd,true -> atomic {
            if
                :: (received_cmd.onOffState == OFF) -> goto end_normal;
                :: else -> skip;
            fi
        }
    od
```

Listing 5.6: Waiting client for PeriodicTask. A waiting state is added with a different do-loop. In that do-loop it waits for the other component to send a state message where the `onOffState == OFF`.

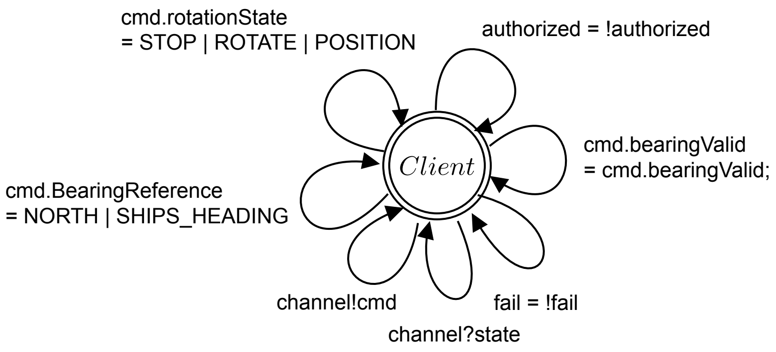


Figure 5.11: State machine for DriveControl base client.

In this implementation it is also shown that there are integers in the message, although they are commented out in this client. Since integers are being used for the `rotationSpeed` and `bearing` properties of a rotation message, the possible variations of rotation messages becomes infinite. Therefore, these values need to be limited in the formal modal to prevent a state-space explosion. In fact, either the bounds or the increment/decrements of the values have to be relatively small as well, since the use of large integers already has extreme effects to the size of the state-space. In fact, these lines remain unused and are commented out. Uncommenting these lines would not have any practical effect on the verification other than that the verification time would be much longer. The fact that they are commented out and not completely removed is to illustrate that it is possible to verify message with integers while keeping the state-space small enough to allow verification in a reasonable amount of time. However, these variables have no effect on the behaviour of the interaction and can be commented out.

```
    ... // (declaration and initialization of variables)

    end_client: do
        :: authorized = !authorized; // global variable
        :: fail = !fail; // global variable
        :: new_rotation.rotationState = STOP;
        :: new_rotation.rotationState = ROTATE;
        :: new_rotation.rotationState = POSITION;
        // :: (new_rotation.rotationSpeed < 10) -> new_rotation.rotationSpeed++;
        // :: (new_rotation.rotationSpeed > 0) -> new_rotation.rotationSpeed--;
        :: new_rotation.bearingValid = !new_rotation.bearingValid;
        // :: (new_rotation.bearing < 10) -> new_rotation.bearing++;
        // :: (new_rotation.bearing > 0) -> new_rotation.bearing--;
        :: new_rotation.BearingReference = NORTH;
        :: new_rotation.BearingReference = SHIPS_HEADING;
        :: RotationCommandMsg_channel!new_systemHdr,new_commandHdr,new_rotation;
        :: RotationStateMsg_channel_rx?received_system,received_responseHeader,
    received_rotation,true;
    od
```

Listing 5.7: Base client for DriveControl.

5.5.2 Non-progress client

The non-progress client for the DriveControl is very similar to the non-progress client of the PeriodicTask. It is based on the base client and it has the `progress` labels on each transition except the communication. The implementation is found in Listing 5.8.

```
end_client: do
        :: true -> progress_0: authorized = !authorized;
        :: true -> progress_1: fail = !fail;
        :: true -> progress_2: new_rotation.rotationState = STOP;
        :: true -> progress_3: new_rotation.rotationState = ROTATE;
        :: true -> progress_4: new_rotation.rotationState = POSITION;
        :: true -> progress_5: new_rotation.bearingValid = !new_rotation.bearingValid;
        :: true -> progress_6: new_rotation.BearingReference = NORTH;
        :: true -> progress_7: new_rotation.BearingReference = SHIPS_HEADING;
        :: RotationCommandMsg_channel_tx!new_systemHdr,new_commandHdr,new_rotation;
        :: RotationStateMsg_channel_rx?received_system,received_responseHeader,
    received_rotation,true;
    od
```

Listing 5.8: Non-progress client do-loop for DriveControl

5.5.3 Waiting client

The waiting client of DriveControl has the same functionality as the base client with one difference: When it sends a STOP command it waits until a state message is received that indicates that the drive has stopped. Therefore, this example is very similar to the 'waiting' client of PeriodicTask. The waiting client for DriveControl is visualized in Figure 5.12. The implementation is rather big and is therefore not shown here. It can be seen in appendix H (file: src/drive-control/client-waiting.pml). The implementation is similar to the PeriodicTask implementation.

5.6 Modifications for verification

When a model is verified for deadlocks, it is not necessary to change the model. However, to apply invariants or liveness properties it is necessary to make some minor additions to the model. Although invariants are a subset of liveness properties, a distinction is made because they can be applied in a different method, and also because they serve a different purpose.
Invariants indicate boundaries for values. They can be applied using LTL statements, or by using assertions. Both options are shown in Listing 5.9. When these additions are applied to

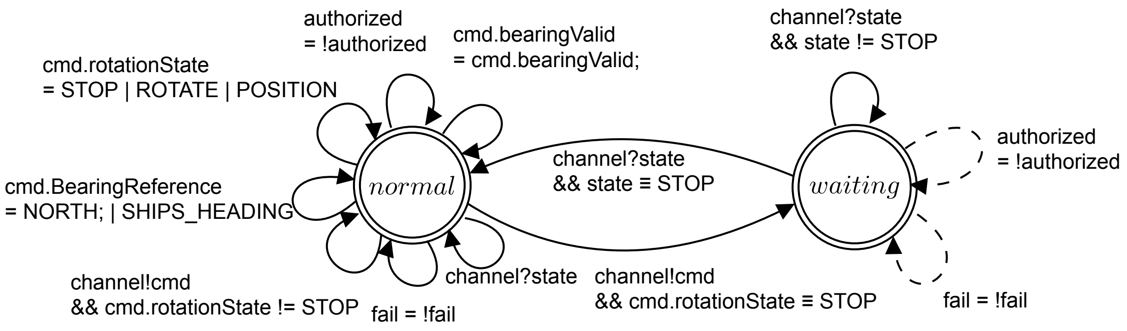


Figure 5.12: State machine for DriveControl waiting client.

the model they are added under the `init` statement. In appendix H the used LTL statements for Drivecontrol can be seen in the drive-control.pml file, and in appendix G the used LTL statements for PeriodicTask are found in the Periodic-task.pml file. Liveness properties can only be specified using LTL statements or Never claims. These are statements that for example indicate a given state to be reached or to specify that a variable has a specific value in a given state, etc. Internally, SPIN always uses never claims. An LTL statement is automatically converted to a never claim that is being verified by SPIN. A never claim is a Promela construct that uses `accept` labels. A never claim is violated if the final parenthesis is reached. Therefore, a never claim should describe incorrect behaviour an nothing more. If the incorrect behaviour is matched by the never claim then it is violated. Never claims are only indirectly used in this research through the use of LTL statements.

```
// approach 1
ltl {[] velocity < 100 }

// approach 2
active proctype checkBound() {
    do
        :: assert(velocity < 100);
        :: skip;
    od
}
```

Listing 5.9: Two different approaches to verify a system for invariants and/or LTL statements.

6 RESULTS

This chapter shows the verification results based on the models created in the implementation chapter. section 6.1 shows an overview of the results of the research. The sections that follow show more details on how these results have been achieved: deadlocks results are described in section 6.2, invariant results are described in section 6.3, verification of non-progress cycles is described in section 6.4 and the results for liveness verification is shown in section 6.5.

6.1 Overview of results

Table 6.1 shows an overview of how the different properties could be verified for O2N based systems. The results for PeriodicTask and DriveControl are remarkably similar, therefore in this table their results could be put together, although the reason of success/failure is may differ. As the table shows, sometimes an 'inappropriate' client is used. For example in the case of non-progress cycles, where the non-progress verification is run for each client. As expected, these clients fail which is desired behaviour: it shows that SPIN can find the problems that are present in these clients. These 'inappropriate' verification runs are marked with one or two asterisks (*) in the table. The results marked with no asterisk are the most interesting results, since these results can be failures where a success is expected, or a success when a failure is expected. For liveness different verification runs have been performed with both successful and unsuccessful results.

In the remainder of this chapter a more detailed overview is given of the verification results. For each verification run slightly different settings and/or a different client is used. To clarify which settings and clients are used, a table is added that shows the relevant details. The table describes which base model is used, and within that model a few settings can be enabled: The respective client can be inserted, the variable `_PROGRESS_LABELS_ENABLED` can be set, and the used SPIN command is shown. More information on SPIN commands is given in Appendix A. Using the information given for each verification and the source code in the appendices it is possible to obtain the same results.

Model	**Both PeriodicTask and DriveControl**					
Client	**Base client**		**Waiting client**		**NP client**	
`_PROGRESS_LABELS_ENABLED`	**0**	**1**	**0**	**1**	**0**	**1**
Deadlocks	✓	✓	×	×	✓	✓
Invariants	-	-	-	-	-	-
Non-progress cycles	×*	×*	×*	×*	×**	×
Liveness properties	×*	×*	×*	×*	×**	×✓

Table 6.1: Overview of verification results. (✓): no errors found, (×): errors found, (-): no results. *: expected violation in client, **: expected violation in buffer.

6.2 Deadlocks

When verifying for deadlocks it does not matter if progress labels are being used. Therefore, the non-progress client will have the same results as the base client, and also enabling or disabling the `_PROGRESS_LABELS_ENABLED` variable has no effect to the results.

6.2.1 PeriodicTask - Base client and non-progress client

First the base client is verified for deadlocks. The verification setup is shown in Table 6.2. The non-progress clients have the same results as the base client when verifying for deadlocks, because the progress labels have no effect when determining. The verifier indicates that no errors are found. This means that, if the client is always able to send any `PeriodicTaskCmdMessage`, the interaction will not deadlock. However, the base client is only a one-way communication, therefore deadlocks will only occur in this model when a message can be sent that forces the PeriodicTask statemachine in a state that it cannot leave, otherwise the client can always do a transition, and no deadlock is present.

Model	periodic-task.pml
Client	client-base.pml or client-nonprogress.pml (gives same results)
_PROGRESS_LABELS_ENABLED	`0 or 1` (gives same results)

SPIN command	`spin -a  periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DSAFETY -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000  -c1`

Result	`no errors found`

Table 6.2: Setup for verifying deadlocks, using the base client or non-progress client for PeriodicTask.

6.2.2 PeriodicTask - Waiting client

The waiting client is also verified for deadlocks. The setup is shown in Table 6.3. The verifier indicates that an invalid endstate (deadlock) is found. After inspection, it appears there is a problem with the specification. This means that SPIN has correctly found a problem.

Model	periodic-task.pml
Client(s)	client-waiting.pml
_PROGRESS_LABELS_ENABLED	`0 or 1` (gives same results)

SPIN command	`spin -a  periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DSAFETY -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000  -c1`

Result	`Invalid endstate (deadlock)`

Table 6.3: Setup for verifying deadlocks, using the 'waiting' client for PeriodicTask.

The deadlock occurs whenever the client sends an `OFF` command to PeriodicTask, while the periodicTask is already turned off, and the `isEnabled` variable is set to `true`. In that case, it is specified that the PeriodicTask does not need to do anything. However, the client is waiting for an acknowledgment that the PeriodicTask is `OFF`. The problem can be fixed by adding the behaviour that a state message is returned anyway. The code where the problem occurs, and the fix to the problem are shown in Listing 6.1. When the fix is applied, the same setup (Table 6.3) can be used and no errors are found instead.

```
1  end_STATE_OFF: do // initial state
2      :: PeriodicTaskCommandMsg_channel_rx?received_system,header,cmdData,true -> {
3              if
4              :: (isEnabled == false
5                      // START DEADLOCK FIX
6                      || cmdData.onOffState == OFF
7                      // END DEADLOCK FIX
8                      ) -> {
9                  responseHeader.commandSourceId = header.commandSourceId;
10                 responseHeader.response = REJECT;
11                 periodicTaskCommand.onOffState = OFF;
12                 PeriodicTaskStateMsg_channel_tx!system,responseHeader, periodicTaskCommand;
13                 // implicity stay in STATE_OFF
14             }
15             :: (isEnabled == true && cmdData.onOffState == ON) -> {
16                 responseHeader.commandSourceId = header.commandSourceId;
17                 responseHeader.response = ACK;
18                 periodicTaskCommand.onOffState = ON;
19                 PeriodicTaskStateMsg_channel_tx!system,responseHeader,periodicTaskCommand;
20                 goto STATE_ON;
21             }
22             fi
23         }
24     od
```

Listing 6.1: Part of the PeriodicTask Promela model displaying invalid behaviour. The part that is annotated as "Deadlock fix" on line 6 was initially not present in the specification. Therefore, when `isEnabled == true` and `cmdData.onOffState == OFF` no behaviour is specified and the model deadlocks. Adding the deadlock fix to the model resolves the issue.

6.2.3 DriveControl - Base client and non-progress client

Using the Base client or non-progress client no deadlocks are found in the modelled interactions with DriveControl. Since the complexity of DriveControl is higher, the maximum search depth for this verification run must be increased with factor 100 to make sure all possible states are reached. When the verification run is completed SPIN indicates that no errors are found. The verification run is shown in Table 6.4.

Model	drive-control.pml
Client(s)	client-base.pml or client-nonprogress.pml (gives same result)
_PROGRESS_LABELS_ENABLED	`0 or 1` (gives same result)

SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
	`./pan -m1000000`

Result	`No errors found`

Table 6.4: Setup for verifying deadlocks, using the base client for DriveControl.

```
:: (authorized == true) -> {
        if
            :: (fail == false) -> {
                ...
            }
            // START DEADLOCK FIX
            :: else -> {
                responseHeader.commandSourceId = header.commandSourceId;
                responseHeader.response = REJECT;
                rotationData.rotationState = STOP;
                RotationStateMsg_channel_tx!system,responseHeader,rotationData;
            }
            // END DEADLOCK FIX
        fi
    }
```

Listing 6.2: Part of the DriveControl Promela implementation. The part that is annotated as a deadlock fix was initially missing from the specification, leading to a deadlock. When the missing behaviour was added into the model the dialogue seems to be free of deadlocks.

6.2.4 DriveControl - Waiting client

When using the waiting client, SPIN shows that there is a deadlock present. The verification run that is used is shown in Table 6.5. The deadlock is caused by the fact that this client waits for a state message before it continues, but there is a state in the DriveControl where it does not send a state message back.

Model	drive-control.pml
Client(s)	client-waiting.pml
_PROGRESS_LABELS_ENABLED	`0 or 1` (gives same result)

SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DSAFETY -DNOCLAIM -w -o pan pan.c`
	`./pan -m1000000`

Result	`Invalid endstate (deadlock)`

Table 6.5: Setup for verifying deadlocks, using the waiting client for DriveControl.

In the behaviour of the DriveControl the problem can be found. When `authorized` is `true`, and when `fail` is `true` as well, there is a state where nothing is defined in the specification. The absence of this behaviour is also seen in the ComMA specification. Most likely this problem was a human mistake when making the ComMA specification. SPIN has rightfully found a mistake. Adding the missing behaviour (which could be derived from Figure 3.3) solves the issue, and then the same verification run succeeds. The part of the specification with the deadlock fix added is shown in Listing 6.2.

6.3 Invariants

There are no results for applying invariants to PeriodicTask and DriveControl, since invariants are not applicable to the PeriodicTask and DriveControl components. This is not necessarily a problem. There are simply no properties in both the PeriodicTask and DriveControl that would need an invariant to be verified. However, in the process of searching for applications for invariants still some interesting insights have been found, which are described in this section.

The ComMA specification of PeriodicTask contains some data and timing constraints, which are used for the run-time monitoring functionality that ComMA offers. Timing constraints indicate the behaviour of the model with respect to time. This is something that cannot be modelled using SPIN, since SPIN does not have a notion of time. In Listing 6.3 the specification for the timing constraints are shown.

```
timing constraints
    // Depending on the client the command is received between once per second and once per 10
     seconds
    cmd_timing signal PeriodicTaskCommandMsg -[ 1000.0 ms .. 10000.0 ms ]-> signal
    PeriodicTaskCommandMsg

    // The periodic task is scheduled every scan period (between 2 and 5 seconds)
    state_msg_timing notification PeriodicTaskStateMsg and notification PeriodicTaskStateMsg
    -> [.. 5000.0 ms] between events
```

Listing 6.3: Timing constraints for PeriodicTask in ComMA

The data constraints in ComMA for PeriodicTask show potential. For PeriodicTask the data constraints are shown in Listing 6.4. As the name implies, data constraints can be seen as boundaries for the data values that should not be crossed. This is something that could easily be specified in SPIN in multiple ways, for example using assertions or by using LTL statements. However, the data constraint shown in Listing 6.4 does not work for the Promela model, since it is not specified in ComMA what happens with this data. The variable `budgetUsed` is not updated in the model so from this specification it is impossible to say if this constraints will hold or not.

```
data constraints
    variables
        SystemHdr system1
        CommandHdr header1
        PeriodicTaskCommand cmdData1

        SystemHdr system2
        DataHdr header2
        PeriodicTaskPerformance performance

    budget_req signal PeriodicTaskCommandMsg(system1, header1, cmdData1)
               until notification PeriodicTaskPerformanceMsg(system2, header2, performance)
               where performance.budgetUsed <= cmdData1.searchBudget
```

Listing 6.4: Data constraints for PeriodicTask in ComMA

The constraints specified in the ComMA specification for PeriodicTask did not help much. The problem is that the values that are used as data constraints in ComMA can be observed in ComMA because ComMA applies runtime monitoring of these variables. In a Promela model these variables are only relevant if the behaviour of these variables is known. If there is no behaviour specified for these variables, then in Promela it must be assumed that it the variable can have any value. On the other hand, if the value of the variable does not matter for the behaviour of the system, it does not need to be modelled anyway, as it only pollutes the state space. For DriveControl the same problem appears to be present, but with some differences. The `rotationcommand` does contain integers. For example the `rotation.rotationSpeed` variable. But, applying invariants to these values is not useful, since it is not known what happens with these variables, and therefore they cannot be modelled.

6.4 Non-progress cycles/Livelocks

6.4.1 PeriodicTask - base client or waiting client

The base client and waiting client do not have progress labels. Therefore, it is expected that non-progress cycles will be found in the client. This is indeed the case, regardless of the progress label in the buffer. In all cases an infinite cycle is found where the line `isEnabled = !isEnabled` from the client is infinitely visited.

Model	periodic-task.pml
Client(s)	client-base.pml or client-waiting.pml (same results)
`_PROGRESS_LABELS_ENABLED`	`0 or 1` (same results)

SPIN command	`spin -a periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.6: Setup for verifying non-progress cycles, using the base/waiting client for PeriodicTask without progress labels in the buffer.

6.4.2 PeriodicTask - nonprogress client

In Table 6.7 the verification run is shown for livelocks without a progress label in the buffer. Therefore, it is expected that a non-progress cycle in the buffer will be found. This is indeed the case. A simulation trace is shown in Figure 6.1. It can be seen that the same values are pushed into the buffer. Since the buffer size is 1, it contents get overwritten with the exact same values resulting in the non-progress cycle. The verification runs with missing progress labels are not a relevant situation, but it does show that with these settings non-progress cycles can be found.

Model	periodic-task.pml
Client(s)	client-nonprogress.pml
`_PROGRESS_LABELS_ENABLED`	`0`

SPIN command	`spin -a periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.7: Setup for verifying non-progress cycles, using the non-progress client for PeriodicTask without progress labels in the buffer.

In Table 6.8 the same verification run is attempted, but now with a progress cycle in the buffer. As described in the method, it is expected that a livelock will occur when the client keeps turning the PeriodicTask on, while PeriodicTask keeps turning itself off. This livelock is indeed found. The simulation trace of this livelock is shown in Figure 6.2. It can be seen that the last step from

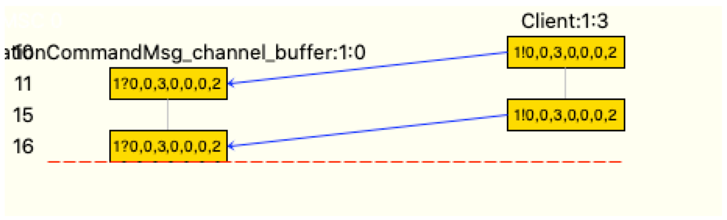


Figure 6.1: Simulation of livelock occurring in the buffer. The simulation shows that the client puts the same message into the buffer, and the buffer is not emptied in between.

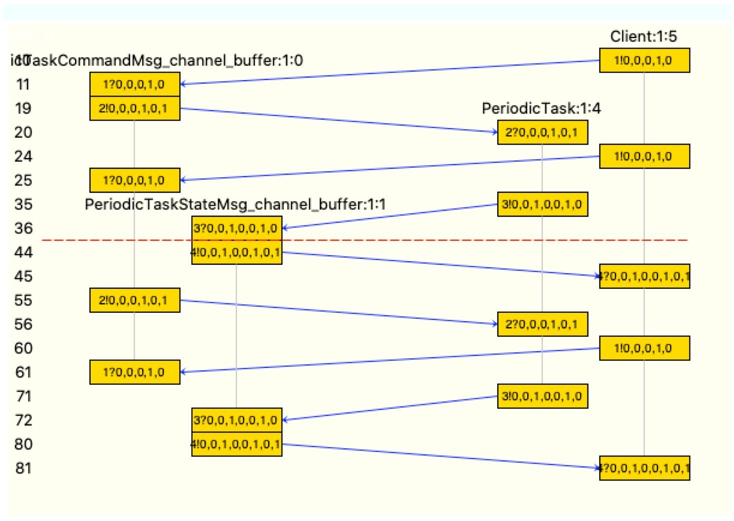


Figure 6.2: Simulation of the livelock found in an interaction with PeriodicTask. (line 10-20): the client turns the PeriodicTask on with a command message. (line 24-25): The client puts another `ON` message in the buffer. This could also be done after PeriodicTask has turned itself off, this order does not matter. (line 35-45) PeriodicTask turns itself of and notifies it with a state message. (line 55-56) PeriodicTask receives the next `ON` message, which is similar to line 19-20. From there the cycle repeats with the same steps.

PeriodicTask in the cycle is a communication to the state message channel. This part of the communication can only be triggered by an internal trigger. There is only one case when that happens: when it turns itself off.

This is not really a problem in PeriodicTask, since it is valid behaviour that the PeriodicTask can be turned on, and that the PeriodicTask can turn itself off. But it is the dialogue that gives problems. It cannot be assumed that the client knows what happens in the PeriodicTask component and it can be expected to try to turn the PeriodicTask on. A quick solution to this problem could not be found, however, this problem will be elaborated more in the discussion chapter.

Model	periodic-task.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	1

SPIN command	`spin -a periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.8: Setup for verifying non-progress cycles, using the non-progress client for PeriodicTask with progress labels in the buffer.

6.4.3 DriveControl - base client or waiting client

When using the base client and the waiting client it is expected that the client will contain a livelock. Like PeriodicTask this is the case: the `fail` variable is toggled infinitely often.

Model	drive-control.pml
Client(s)	client-base.pml or client-waiting.pml (same results)
_PROGRESS_LABELS_ENABLED	`0 or 1` (same results)

SPIN command	`spin -a drive-control.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.9: Setup for verifying non-progress cycles, using the base/waiting client for DriveControl without progress labels in the buffer.

6.4.4 DriveControl - non-progress client

Table 6.10 shows a verification run for the non-progress client for DriveControl, with no progress label in the buffer. Like the PeriodicTask it is expected that the non-progress cycle is found. This is indeed the case, a non-progress cycle is found in the buffer.

Model	drive-control.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	`0`

SPIN command	`spin -a drive-control.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.10: Setup for verifying non-progress cycles, using the non-progress client for DriveControl without progress labels in the buffer.

Now the DriveControl component will be verified for deadlocks with progress labels in the buffer. The verification run is shown in Table 6.11. It is not expected that a non-progress cycle will be found. However, a non-progress cycle is found by SPIN. The simulation trace is shown in Figure 6.3. It appears this problem occurs when the client sends the same command and the DriveControl simply receives this, and stay in the same state. When the client keeps repeating the same command over and over and it has no effect on the DriveControl then this is indeed a non-progress cycle. However, it is not a problematic cycle. It appears to be normal system behaviour. Therefore, it is difficult to apply a fix to a functioning system.

Model	drive-control.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	`1`

SPIN command	`spin -a drive-control.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000 -l`

Result	`Error: non-progress cycle`

Table 6.11: Setup for verifying non-progress cycles, using the base non-progress for DriveControl with progress labels in the buffer.

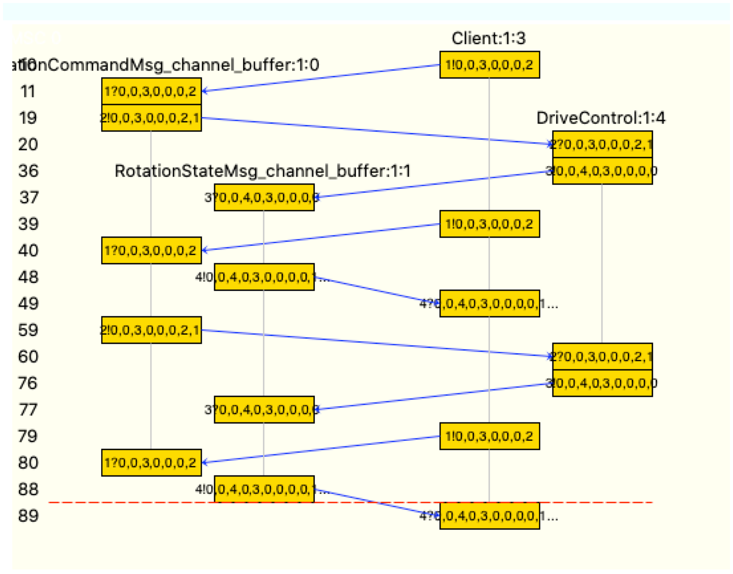


Figure 6.3: Simulation of the livelock found in an interaction with DriveControl. The infinite cycle starts at line 39/40, where the client indicates with a command that the DriveControl must start rotating. The DriveControl receives this (line 59/60) but it is already rotating, so the DriveControl does not change anything and replies with a state message (line 76/77) This can happen infinitely often, so SPIN sees it as a non-progress cycle.

6.5 Liveness

In this section the liveness results are shown. For these verification runs an additional LTL statement is listed in the table to indicate which LTL statement is used. Liveness properties are satisfied when no infinite cycle is found where the given property does not hold. If an infinite cycle is found where the property does not hold, the property is not satisfied and SPIN will show the cycle where the problem occurs. Often these cases are non-progress cycles that are of no interest, so it is important to introduce fairness constraints. Because verifying liveness properties is not as straightforward as verifying other properties, it will first be explained how the verification attempts will be performed.

6.5.1 Details on verifying Liveness

Fairness constraints

In previous results it is shown that non-progress cycles were present in the buffers and in the clients when no progress labels are used. Fairness can be used to make sure that non-progress cycles are not executed infinitely often when other transitions are infinitely long/often (different weak or strong fairness) available as well. SPIN allows users to enforce weak fairness to prevent uninteresting non-progress cycles from violating the liveness properties. When enforcing weak fairness constraints to a liveness verification the non-progress cycles will still result in a violation of the liveness property, due to the non-progress cycle in the buffers. At this point it appears that the weak fairness constraint from SPIN does not suffice for this model.

Weak fairness in spin indicates that a process that is infinitely enabled (such that it can do transition) it eventually has to do that transition. It does not work for enabled transitions in a do-loop. This could be because weak fairness enforces that transitions that are infinitely long enabled eventually must be taken, (such as in a waiting process) while in a do-loop the transitions become infinitely enabled over and over again. Which is not weakly fair, but strongly fair. Forcing weak fairness constraints has a large effect on the state space. For weak fairness this effect is linear with the number of active processes, while for strong fairness this number is quadratic. Therefore, the effect on the performance of the verification is large. This is the reason it is not included as a default command in SPIN [41].

The problem with weak fairness in Promela is illustrated in Listing 6.5 and Listing 6.6. Listing 6.5 shows the problem with weak fairness: within a process a transition can be continuously enabled but with enforced fairness there is still a possibility that not all transitions are taken. A solution could be to give each branch a separate process to mitigate this issue, which is shown in Listing 6.6. However, with the complexity of PeriodicTask and DriveControl this is not feasible since the amount of separate processes would become too large. In SPIN, it is possible to enforce strong fairness using the following statement: `[]<> enabled(0) -> []<>_last==0`. The `enabled` keyword indicates if a process can do a transition, and the `_last` keyword returns the id of the last process that performed an action. Therefore, this statement means: *if the process with id 0 is always eventually enabled, then in all cases it eventually must be the last process to have performed an action*. To perform strong fairness on all process this statement must be written for each process. This is not much work: the model with most process is the PeriodicTask model with 6 processes, consisting of 3 buffer processes, the PeriodicTask process, the client process and the default `init` process. However, this does not work when rendezvous channels are used, Which is the case with the custom buffers. Therefore, enforcing strong fairness is not a viable options for the models that have been created.

```
proctype a() {
    do
        :: statement_a;
        :: statement_b;
        :: statement_c;
    od
}
```

Listing 6.5: Promela do-loop illustrating the problem with weak fairness. With weak fairness enabled SPIN can (for example) still endlessly toggle between `statement_a` and `statement_b`, and never execute `statement_c`.

```
proctype a() {
    do
        :: statement_a;
    od
}

proctype b() {
    do
        :: statement_b;
    od
}

proctype c() {
    do
        :: statement_c;
    od
}
```

Listing 6.6: Because weak fairness in Promela is enforced on process label it is possible to enforce that `statement_c` is reached by splitting the branches in different processes.

Verifying liveness without enforcing fairness constraints

Enforcing weak or strong fairness on the full models is not going to work. But this does not mean that liveness properties cannot be used at all. Promela offers a special variable that can be used in LTL properties: `np_`. This variable is read-only, and returns `true` whenever the current cycle is not a progress cycle. To attempt liveness, it will therefore be attempted to use this variable to

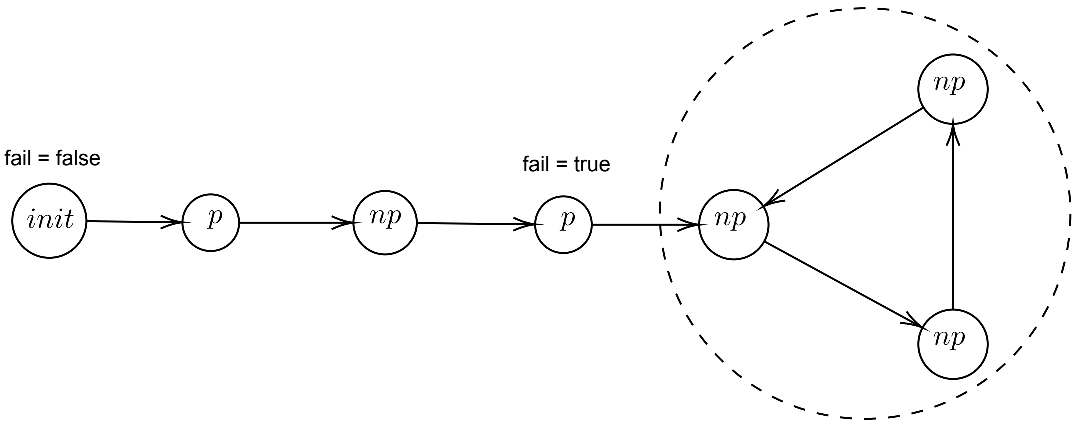


Figure 6.4: Example system showing that a non-progress cycle does not indicate that progress labels are never reached during an execution. It indicates that progress labels are not *infinitely* reached. The arrows indicate transitions between states, and each state (except the initial state) is annotated with either 'p' for a progress state and 'np' for a non-progress state.

rule out cycles that are of no interest, and to only keep cycles that are interesting. This process is already done for the process of verifying progress cycles: the non-progress client already contains progress labels on all self-loops. Also the buffers contain a progress label, so it appears that exactly those cases that provide unnecessary non-progress can be ruled out using the `np_` variable. The variable will be used as shown in Listing 6.7, which can be read as: *"If it eventually happens that from a point every state is a non-progress state, the LTL statement must hold"*.

```
ltl { ( (<>[] np_) -> ltl_statement } // 'ltl_statement' must hold for non-progress cycles
```

Listing 6.7: The implication-operator is used to create LTL statements to verify liveness properties an non-progress cycles.

However, a possible system execution might take some time before eventually reaching the non-progress cycle. This is visualized in Figure 6.4, where states are displayed as circles. After the initial state, the system can visit 'progress'-states. However, eventually it will reach a cycle where `np_` always holds (hence `<>[] np_`). In the part of the execution before the actual non-progress cycle occurs, it is still possible that variables are toggled. For example in the LTL statement: `ltl ( (<>[] (np_)) -> (!fail -> ltl_statement)` . This can be read as *"in every execution where there is eventually a non-progress cycle, if fail is false then the* `ltl_statement` *must hold"*. Figure 6.4 shows that `fail` can be `false`, but when the non-progress states are reached fail could be set to `true`, therefore `ltl_statement` could still be violated since `fail` could toggle it's value before the non-progress cycle without affecting other processes. Because of this, it is important to limit the LTL statement to states where eventually all states are both non-progress states where a precondition holds. This is shown in Listing 6.8. In the example described above, `!fail` is the precondition, so `ltl ( (<>[] (np_ && !fail) -> ltl_statement` limits the statement such that `ltl_statement` only must hold in non-progress cycles where `fail` is always `false`.

```
// 'ltl_statement' must hold only for non-progress cycles where precondition holds
ltl { ( (<>[] (np_ && precondition)) -> ltl_statement }
```

Listing 6.8: An extra precondition can be used to only verify non-progress executions where a given precondition must hold during the cycle.

From the non-progress-cycle verification it is already known that the base and waiting client shows non-relevant progress cycles. Therefore, the non-progress cycle results for these clients will also apply for the liveness results.

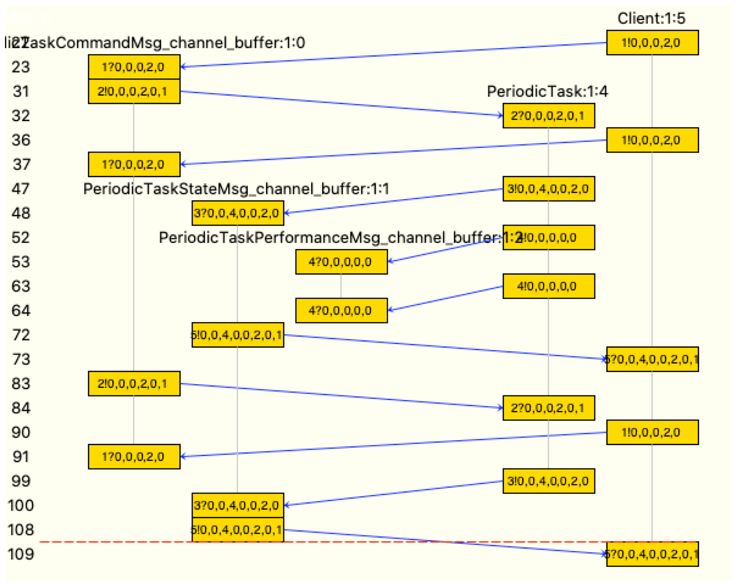


Figure 6.5: Visualization of the liveness violation found in SPIN for the verification run shown in Table 6.12. The state of all process are the same when comparing lines 36-52 with lines 90-100. Within this cycle `isEnabled` is always `false` and `periodicTaskCommand.onOffState == OFF` also always `false`.

6.5.2 PeriodicTask

LTL 1: `!isEnabled` must result in `OFF`-state

When the `isEnabled` variable is `false` then eventually the OFF state must be reached as well. The used LTL statement is shown and explained in Listing 6.9.

```
ltl p1 {  (<>[] (np_ && !isEnabled)) -> ([]<> (periodicTaskCommand.onOffState == OFF)”}
```

Listing 6.9: *"For every execution where eventually all states are non-progress states, and in those states* `isEnabled` *is* `false`*, then the* `onOffState` *always must eventually be* `OFF`*".*

The details of the verification run are shown in Table 6.12. It can be observed that the used LTL property is violated. The simulation visualization is shown in Figure 6.5. It shows a trace where the `isEnabled` value is set to `false`, but a cycle is found where the `periodicTaskCommand.onOffState == OFF` is never true. When looking into the specification this is indeed a possible cycle. when the periodicTask is on, and it receives a message to stay on it is not checked if the `isEnabled` property is still valid. Instead, this is only checked if the PeriodicTask is first off and then turned on. It is possible that this is a mistake in the specification.

Model	periodic-task.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	1

SPIN command	`spin -a  periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DCOLLAPSE -w -o pan pan.c`
	`./pan -m10000000  -a -N p1`

LTL query	`#define p (!isEnabled)`
	`#define q ([]<> (periodicTaskCommand.onOffState == OFF))`
	`ltl p1 {  (<>[] (np_ && p)) -> q  }`

Result	`Errors found`

Table 6.12: Verification run for PeriodicTask for LTL statement 1.

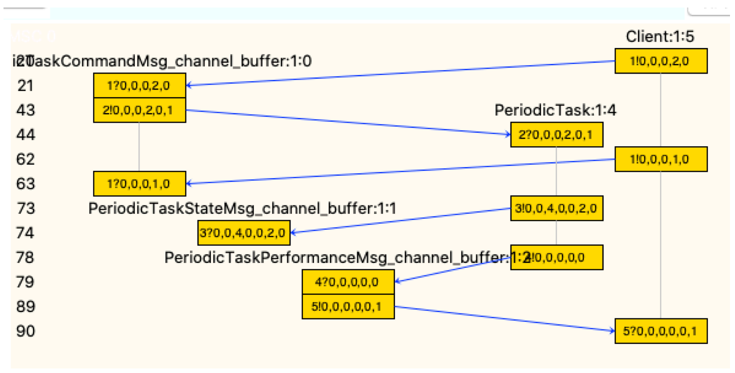


Figure 6.6: Visualization of the liveness violation found in SPIN for the verification run shown in Table 6.13. The infinite cycle starts at line 78 where only performance messages are being sent. The `periodicTaskCommandMsgChannel_buffer` contains the `OFF` message, but it is never actually received by the PeriodicTask.

LTL 2: OFF message must result in OFF-state

Next it is attempted to verify if sending an OFF message will always result in the OFF state being reached. The LTL statement that is used is shown in Listing 6.10. Because data can be lost in the buffer this property should be violated. However, because there is a progress label present in the buffers this cycle will not be detected. Therefore, this example illustrates a case where a liveness property is incorrectly satisfied due to the trick with progress labels. Removing the progress label in the buffer will expose the cycle in the buffers. Hence, the problem cannot be found with SPIN using this method. It is expected that with the other buffer implementation (**??**) this can be fixed.

```
ltl { (<>[] (np_ && (test_cmd.onOffState == OFF))) -> (<> (periodicTaskCommand.onOffState ==
    OFF) }
```

Listing 6.10: *"For every execution where eventually all states are non-progress states, and in those states the message that is being transmitted is* `onOffState.OFF`*, then the* `onOffState` *of the PeriodicTask always must eventually be* `OFF`*".*

Interestingly, this still results in a violation. The simulation is shown in Figure 6.6. This time it is not a buffer overflow, but merely a buffer that is never being read from in an infinite cycle where the system is behaving normally in the ON state. Again, this behaviour is not a fair behaviour. The verification run shows a problem that is not fair and is not likely to be a problem in the real system. Therefore, strong fairness is needed, because the receiving end is sending performance messages and can not always receive messages; therefore the transmission of the message is not continuously enabled (required for weak fairness). Instead, it is enabled infinitely often. To enforce the transition to be taken, strong fairness must be enforced which is not possible with the current buffer implementation.

Therefore, the expected result that a violation in the buffer would not be found is met, that violation is indeed not found. However, another violation is found, but it is only occuring with unfair behaviour.

Model	periodic-task.pml
Client(s)	client-nonprogress.pml
`_PROGRESS_LABELS_ENABLED`	1

SPIN command	`spin -a  periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DCOLLAPSE -w -o pan pan.c`
	`./pan -m10000000  -a -N p2`

LTL query	`#define q (<> (periodicTaskCommand.onOffState == OFF))`
	`#define r (test_cmd.onOffState == OFF)`
	`ltl p2 { (<>[] (np_ && r)) -> q }`

Result	`Errors found`

Table 6.13: Verification run for PeriodicTask for LTL statement 2.

LTL 3: `ON` message results in `ON` state

Another attempt will be to show that if an ON command is sent and `isEnabled == true`, that the next state will be ON. The LTL statement is shown in Listing 6.11.

```
ltl p3 {(<>[] (np_ && (test_cmd.onOffState == ON))) -> (<> (periodicTaskCommand.onOffState ==
    ON)) }
```

Listing 6.11: *"For every execution where eventually all states are non-progress states, and in those states the message that is being transmitted is* `onOffState.ON`*, then the* `onOffState` *of the PeriodicTask must eventually be* `ON` *as well".*

Table 6.14 shows the verification run for this property using LTL statement `p3`. The property is violated since the ON message can be rejected. Because the `isEnabled` variable can change. This is a mistake in the LTL statement. In LTL statement `p4`, shown in the same table, this problem is resolved and then no errors are observed.

Model	periodic-task.pml
Client(s)	client-nonprogress.pml
`_PROGRESS_LABELS_ENABLED`	1

SPIN command	`spin -a  periodic-task.pml`
	`gcc -DMEMLIM=1024 -O2 -DXUSAFE -DCOLLAPSE -w -o pan pan.c`
	`./pan -m10000000  -a -N p3`

LTL query	`#define s (test_cmd.onOffState == ON)`
	`#define t (<> (periodicTaskCommand.onOffState == ON))`
	`ltl p3 {(<>[] (np_ && s)) -> t }`
	`ltl p4 {(<>[] (np_ && s && isEnabled)) -> t }`

Result	`Errors found`

Table 6.14: Verification run for PeriodicTask for LTL statement 3.

6.5.3 DriveControl

LTL 1: `!authorized` or `fail` will ensure that the STOP state is reached

For DriveControl it can be verified that `!authorized` or `fail` will result in the stop state being reached. The used LTL statement is shown in Listing 6.12.

```
ltl { (<>[] (np_ && (!authorized || fail))) -> ([] <> (rotationData.rotationState == STOP)) }
```

Listing 6.12: *"For every execution where eventually all states are non-progress states, where* `authorized` *is* `false`*" or* `fail` *is* `true`*, then in all cases eventually the DriveControl must reach the* `STOP` *state."*

When running the verification run, which is shown in Table 6.15, the verification run is stopped before all the states could be explored because too much memory is being used, even when

4GB of memory is available. However, by performing two verification runs, one for `!authorized` and one for `fail` shows that eventually the STOP state will always be reached.

Model	drive-control.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	1

SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=4096 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000  -a -N p2`

LTL query	`#define p (!authorized || fail)`
	`#define q ([] <> (rotationData.rotationState == STOP))`
	`ltl p2 { (<>[] (np_ && p)) -> q } // this hits the memory limit`
	`ltl p2 { (<>[] (np_ && !authorized)) -> q } // success`
	`ltl p2 { (<>[] (np_ && fail)) -> q } // success`

Result	`-DMEMLIM reached - no errors found.`

Table 6.15: Verification run for the first LTL statement for DriveControl.

LTL 2: sending a `STOP` message results in the `STOP` state being reached.

Verifying whether sending a STOP messages results in the STOP state being reached appears to be consuming a lot of memory. When allowing SPIN to use 4Gb of memory and with the 'collapse compression' setting turned on, it is still not enough for the exhaustive verification to complete. However, when using the bitstate-hashing/supertrace storage method the verification run succeeds with no errors. This gives an indication that the LTL property holds, but it is not guaranteed. The failing verification run is shown in Table 6.16. The used LTL statement is shown and explained in Listing 6.13.

```
ltl p4 { (<>[] (np_ && (new_rotation.rotationState == STOP))) -> ([] <> (rotationData.
    rotationState == STOP)) }
```

Listing 6.13: *"For every execution where eventually all states are non-progress states, where the message the client sends contains*`rotationState == STOP`*, then in all cases eventually the DriveControl must reach the* `STOP` *state."*

Model	drive-control.pml
Client(s)	client-nonprogress.pml
_PROGRESS_LABELS_ENABLED	1

SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=4096 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000  -a -N p4`

Supertrace SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=4096 -O2 -DBITSTATE -DXUSAFE -w -o pan pan.c`
	`./pan -m10000000  -k3 -a -c1 -w20 -N p4`

LTL query	`#define q ([] <> (rotationData.rotationState == STOP))`
	`#define r (new_rotation.rotationState == STOP)`
	`ltl p4 { (<>[] (np_ && r)) -> q }`

Result	`-DMEMLIM reached - no errors found.`
Supertrace result	`no errors found.`

Table 6.16: Verification run for the second set of LTL statements for DriveControl.

LTL 3: sending a `ROTATE` message results in the `ROTATE` state being reached.

This verification run indicate whether sending a `ROTATE` message results in the `ROTATE` state being reached.
The used LTL statement is shown and explained in Listing 6.14.

```
ltl p4 { (<>[] (np_ && (new_rotation.rotationState == ROTATE) && authorized && !fail)) -> ([]
    <> (rotationData.rotationState == ROTATE)) }
```

Listing 6.14: *"For every execution where eventually all states are non-progress states, where the message the client sends contains*`rotationState == ROTATE`*, while it is authorized and there is no failure, then in all cases eventually the DriveControl must reach the* `ROTATE` *state."*

Using the verification run in Table 6.17 this appears to be the case: SPIN does not find a violation. However, also for this verification run it appeared that exhaustive verification was not possible due to memory limitations and a bitstatehashing/supertrace storage method has been applied, therefore, it is not guaranteed that there is no violation.

Model	drive-control.pml
Client(s)	client-nonprogress.pml
`_PROGRESS_LABELS_ENABLED`	1

SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=4096 -O2 -DXUSAFE -DNP -DNOCLAIM -w -o pan pan.c`
	`./pan -m10000  -a -N p6`

Supertrace SPIN command	`spin -a  drive-control.pml`
	`gcc -DMEMLIM=4096 -O2 -DBITSTATE -DXUSAFE -w -o pan pan.c`
	`./pan -m10000000  -k3 -a -c1 -w20 -N p6`

LTL query	`#define p (!authorized || fail)`
	`#define s (new_rotation.rotationState == ROTATE)`
	`#define t ([] <> (rotationData.rotationState == ROTATE))`
	`ltl p5 { (<>[] (np_ && s)) -> t } // failure`
	`ltl p6 { (<>[] (np_ && s && !p)) -> t } // success`

Result	`-DMEMLIM reached - no errors found.`
Supertrace result	`no errors found.`

Table 6.17: Verification run for the third set of LTL statements for DriveControl.

7 DISCUSSION

This thesis describes a depth-first exploration on how formal methods can be applied to O2N based systems. The research starts with an analysis of the characteristics of O2N. Using these characteristics, a selection of formal tools is found and eventually a subset of these tools is selected. From these tools, eventually SPIN is chosen. Although there can exist infinitely many O2N components, PeriodicTask and DriveControl have been used for analysis. This chapter aims to review the results in the light of previously made decisions.

In section 7.1 the result outcomes from chapter 6 are discussed. a reflection on the tool selection is described in section 7.2 and the implementation of the models is discussed in section 7.3.

7.1 Result outcomes

In a research like this it is easy to lose sight on the primary goal. It is not the goal of this research to show that PeriodicTask and DriveControl are valid components, which in this case means that they are free of deadlocks, livelocks, and they hold to certain liveness properties. Instead, the goal of this research is *to determine to what extent it is possible to verify such properties*.

Therefore to determine this extent it is necessary to find the limits to what can be verified, and what not, and to reason about the outcome of those verification results. Finding errors in a model can help with gaining an understanding of the model, and also help gain an understanding of what can be verified and what not. When a problem in the model is found by SPIN, and when that same problem is fixed and SPIN indicates that the problem is gone, then that is an indication that SPIN is properly able to find such mistakes. In other words: To show that SPIN can find problems it is not enough to show that no problems are found. This is why, during the verification, 'inappropriate' clients have been used: they introduce such problems. Although the problems in essence are irrelevant (they are problems in the client), they do show that SPIN is looking for the problems and finds them if they exist.

From the result outcomes it can be determined to what extent the verification properties can be applied to O2N components. This is shown in Table 7.1. For each row a section is dedicated below to further describe the outcomes.

	PeriodicTask	**DriveControl**	**Notes**
Deadlocks	++	++	
Invariants	+	+	Possible, but limited applications
Non-progress cycles	+	+	Limitations expected
Liveness properties	-	-	

Table 7.1: Overview of properties that have been verified. (++): Useful for analyzing O2N-based systems (+): Useful, but has some limitations (+-): Can be used but has major limitations (-): (partially) possible, but not very useful (- -): impossible

7.1.1 Deadlocks

During this process, it has been shown that PeriodicTask and DriveControl both contained a deadlock in their specification. SPIN has properly found these deadlocks such that these problems could be fixed. After a fix has been applied SPIN indicates that no deadlocks are present. This shows that SPIN is capable of finding deadlocks in an O2N interaction. However, there is an interesting likeness in the deadlocks that are found in PeriodicTask and DriveControl. There are a few interesting observations to be made:

1. The deadlocks where both found by using the 'waiting' client.
2. The deadlocks both originated from not returning state messages under all circumstances.
3. When no state messages are returned, the component was already in the state that the client 'commands'.

This raises a question whether it is accidental that both ComMA specification contain the same type of mistake, or does the ComMA model not need the behaviour to be specified like this? In the first case, if it was a mistake, then this would be a nice result of the verification: a real problem is found in the specification. But if it is not a mistake, then it may be that there may be difficulties ahead when using exhaustive formal tools on ComMA specifications. Both deadlocks originate when no state messages are returned and the component is actually in the state that the client desires. The problem lies therefore in the dialogue, when the client has incorrect expectations of the component. The component is sending a state message when it's internal state is different. But why would the component send a state message if it is already in the proper state, since nothing changes? Therefore, it is likely that this is a design choice in the component, where either one or two expectations of the client are made: either the client does not wait for state messages, or the client does not send an OFF command when the component is already OFF. However, if the client does do at least one of these things then the dialogue will deadlock. In any case, SPIN has shown that it can find the deadlocks present in the models so it can be concluded that SPIN is suitable for finding deadlocks in O2N components.

7.1.2 Invariants

Invariants have not been used in this research. In the implementation chapter it is explained that there was no useful invariant to implement. However, this does not have to mean that invariants cannot be applied to O2N-based systems. It means simply that invariants cannot be applied to PeriodicTask and DriveControl. An invariant can be easily applied to a model. So whenever an O2N model has a variable, and the behaviour of this variable is known then an invariant can be used to verify if that variable stays within the boundaries. In SPIN this is perfectly possible. The fact that invariants can be used in SPIN can be shown by using the 'simple engine' example, which is used in the tool selection chapter. The engine has a `velocity` variable. An invariant can be created to verify that the value of `velocity` always stays between `0` and `100`. The invariant looks like this: `ltl { [] (velocity <= 100 && velocity >= 0) }`. This is actually a useful invariant for the simple engine. The velocity variable behaves differently in different states, so it is good to know that it always stays within these boundaries.

7.1.3 Non-progress cycles

In the PeriodicTask model the expected livelock has been found. This indicates that SPIN is useful for finding livelocks. The livelock occurs however in behaviour that is exactly specified as such, so it is really the designed interaction that allows the livelock to occur. Therefore, the interaction with PeriodicTask must be changed to make sure that this does not happen. However, the problem is not necessarily a problem in the PeriodicTask component, since the behaviour of being turned on and turning itself off is reasonable. The livelock therefore occurs when the PeriodicTask component and it’s client do not agree on what the state of PeriodicTask should be. At this point the application of the behaviour starts to play a role. If the PeriodicTask shuts itself down due to a problem, then it is reasonable that it shuts itself down all the time. A possible solution could then be that the PeriodicTask may have an ’error’ state in which it can communicate that it cannot be turned on. When it is tried to turn on and it is in the error state, the PeriodicTask may check if the error is still present and if that is the case it can send a state message where it indicates that the ’ON’ command is rejected. Obviously, this is a redesign of the interaction between the components.

For DriveControl an interesting non-progress cycle was found: whenever the client would send the same messages, and these message did not affect the state of the DriveControl, a non-progress cycle is found. This is also what would happen in the real component if the client keeps sending the same messages and therefore it is not a problem. However, it may hide other livelocks that could occur after more transitions, so it would be best to indicate that the found non-progress cycle is not a problem. This is also done with the buffer, which contains a non-progress cycle that is of no interest.

SPIN has found non-progress cycles in every verification run. For most implementations this was expected, for example when no progress labels where present in the clients or in the buffers. But also when the behaviour of the models was correct still for both DriveControl and PeriodicTask non-progress cycles have been found. This can be explained by the fact that both components can always return to their initial state, which SPIN sees as a non-progress cycle. Therefore, if SPIN indicates that there is a non-progress cycle present, it is important to make sure whether this non-progress cycle is desired behaviour or not.

A potential problem when verifying non-progress cycles (and liveness)

A problem may be that cycles that require the client to sent alternating types of messages cannot be found. For example, a non-progress cycle that occurs only when the client sends two or more different messages will not be found. This is because a progress label is present whenever the message is sent. However, no example of such a livelock could be found in PeriodicTask or DriveControl, therefore this problem is not demonstrated anywhere in this research. To have a more clear view on the extent to which livelocks can be verified this specific could be researched in a later research.

Like deadlocks, SPIN is also suitable to detect non-progress cycles in O2N components. There may be non-progress cycles that are missed, but that did not occur in this research. Instead, non-progress cycles have been found in all cases, and in each case SPIN has correctly found it. To use SPIN to detect livelock (undesired non-progress cycles) it requires a manual inspection of the found non-progress cycles.

7.1.4 Liveness

When researching whether liveness could be applied to PeriodicTask and DriveControl a lot of challenges had to be overcome. Enabling the weak fairness option results in drastically slower verification times, and enforcing weak fairness is not enough to overcome an infinite cycle in the custom buffer implementation. Applying strong fairness is also not possible with the custom buffers, since they internally use rendezvous channels. When fairness cannot be enforced it may still be possible to verify certain liveness properties, but it requires workarounds. In this research, it is attempted to use non-progress cycles to verify liveness properties for given cycles.

Liveness in PeriodicTask

Using this approach, two violations have been found in PeriodicTask. The first violation (using LTL 1) was correctly found by SPIN: it was checked if `!isEnabled` made sure that eventually the PeriodicTask would turn off. This was indeed not the case in the specification. It is not known if this behaviour is intentional or not, but if it is assumed that `!isEnabled` must ensure that the PeriodicTask turns off, then it is an error which SPIN has found. Otherwise, if that behaviour is not an error, then simply the LTL statement does not make sense. But for this research it does not matter whether it is an error or not: what matters is that SPIN could successfully verify this behaviour. The second violation (LTL 2) however is not correct. It is a violation that occurs because the model is not fair. This is a limitation of the current workaround that does not enforce weak fairness. Therefore this is an incorrectly found error which may be solved by adding a progress label somewhere in the model. It is an indication that liveness violations using this workaround are not necessarily problems with the specification, and that the results by SPIN need to be analyzed critically. When using the third LTL statement SPIN did initially also find violations of the liveness property, although it appeared that the LTL statement was not specified properly. After fixing the LTL statement SPIN found no errors. This again shows that SPIN can find liveness violations in O2N models, and that it can also indicate when a model is correct.

Liveness in DriveControl

When using the liveness verification methods on DriveControl a few violations have been found, but none of them originated from the interactions in the model. In fact, the main problems that occurred when verifying liveness in DriveControl was the use of memory. DriveControl is more complex than PeriodicTask in the sense that the state contains more variables. Therefore, the state-space is larger. For the first LTL statement the preconditions had to be checked in two runs for the exhaustive verification to complete. And for the other LTL statements another non-exhaustive storage method had to be used for the verification to complete.

Is SPIN suitable for verifying liveness properties in O2N components?

While working with liveness there have been some challenges: fairness could not be properly enforced, and the state-space could become very large, resulting in memory problems. Also creating the LTL statements using a workaround was experienced to be a difficult task. It was experienced that a mistake was quickly made. To properly apply an LTL statement to a system requires a deep understanding of the system to be verified, and also an understanding of how the LTL statement works. Thus, there is quite a contrast to verifying a model for deadlocks, which can be largely automated and verified by a simple SPIN command. On the other hand, there are cases in both DriveControl and PeriodicTask where SPIN has successfully detected violations of the LTL statements.

Therefore, it is definitely possible to use SPIN to verify O2N for liveness properties, but the limitations with regards to fairness and memory use are significant drawbacks.

7.2 Reflection on tool selection

SPIN is a suitable tool for analyzing O2N components, as long as their behaviour is known. When ComMA specifications are used to describe this behaviour, then converting this ComMA specification to a Promela model is not a difficult task.

Initially, SPIN was selected together with UPPAAL and mCRL2 for further research. When modelling a simple engine implementation, it appeared that the Promela language was very suitable for describing systems like O2N. Especially the important command-state pattern was well-supported, as Promela allows for complex message types to be communicated between processes, and it also supports the transmission of state messages when the internal state of a process changes. These traits of SPIN have been exploited well during the research and have proven to be of great value. Using Promela there were no problems with the command-state pattern in both PeriodicTask and DriveControl. With regards to modelling the components it is recommended to use SPIN.

7.2.1 Missing features of SPIN

- **A better user interface** In the tool selection chapter it is discussed that iSpin does not have a good user interface. Throughout the research this gave some problems. When editing a file in the user interface it would sometimes not save properly, and a verification run would be ran on the previous version of the model instead. Also the simulator is heavily used, but exploring the trace was often a puzzling experience. Something that would already help a lot would be to display variable names in the simulator.

- **Verification properties** It is a limitation that only LTL could be used. mCRL2 would support more verification properties, but creating a model in mCRL2 would be difficult. However, there may be another way: there are tools, for example NuSMV [42], that allow the use of CTL. A tool called s2n [43] exists to transform Promela programs to NuSMV. CTL could give an extra set of options to help verify given properties. For example, using CTL it is possible to verify whether an execution path exists to a given state, which is not something that can be specified in LTL. Therefore, when using CTL it is possible to specify: If a ROTATE message is sent, then a path exists to the ROTATE state. In LTL this is not possible, it can only be specified that if a ROTATE message is sent, it eventually must be reached. This is however not the case when data loss is involved. Therefore, when using CTL it is possible to omit the progress labels since the properties don't always need to hold for all follow states.

7.3 Reflection on implementation

Creating the formal models of PeriodicTask and DriveControl could be done systematically. However, the research was quite limited since only two O2N components have been researched. Using such a small amount of components makes it difficult to give solid conclusions. There are some points of interest.

7.3.1 Incompatible types: floats and strings

Using these types for modelling behaviour is not recommended, if it is desired that a component is formally verified. In Promela, floats and strings are not supported. This has not given any problem when modelling DriveControl and PeriodicTask. The floats and strings that where present in the messages had no behaviour that depended on these variables. If an O2N component is specified such that floats and strings have no effect on the behaviour, then this is no problem. Otherwise, it becomes difficult, if not impossible, to model the components.

7.3.2 Buffers

Reflection on buffers

A problem that occurred during the implementation was the use of buffers. SPIN does not natively support every type of buffer that may be used in 02N. Creating a custom buffer was possible, but it came with a cost. The syntax of the signature in Promela became much more complicated as compared to regular Promela buffers. Also the simulator in SPIN could not display the communications as well as usual. In this research a lot of time was spent to get the buffers right. In hindsight, it is questionable if this has had any reasonable effect to the verification results. Many problems in the dialogue (for example the deadlocks) would also have occurred with 'regular' blocking Promela buffers. Therefore, in later research it may be interesting to observe differences between different buffer implementations.

A state-space explosion

During verification a problem in the buffer implementation was found. During a verification run (which is not documented in this research) with a buffer size of 2 for the DriveControl component the state-space increased a lot as compared to a buffer size of 1. Using a buffer size of 1 the verification would take about 7 seconds, while with a buffer size of 2 the verification would run out of memory (4096MB) after several minutes. This is possibly a problem with the buffer, which needs to be studied further. A problem here is that this problem is difficult to find, since the verifier cannot give a trace of this problematic behaviour. The problem therefore needs to be found by reasoning about the specified model. It could be simply be because the DriveControl model in itself is already fairly complex (it takes 7 seconds to verify with the integers commented out, which is already long). And with a larger buffers this amount of time and memory needed increases exponentially. For this research larger buffer sizes have not been necessary. It is recommended to try the buffers with less complex message, and to analyze the effects of the buffer size on the state-space to find the cause of this state-space explosion.

7.3.3 Client approaches

In the method multiple approaches have been discussed on how to verify the PeriodicTask and DriveControl model. The third approach, which means the exhaustive approach, seemed to be the best solution, with a risk of state-space explosions. During this research it has appeared that the state-space could be managed reasonably well: only with liveness properties in Drive-Control the memory limits have been reached. Although using integers in the messages did increase the state space significantly, these were often not needed for a proper verification and could be removed from the client, since no behaviour of the models was dependent on integers. This may be luck, since both PeriodicTask and DriveControl don't have behaviour based on integers. However, the fact that integers heavily increase the state-space does not mean that verification becomes impossible. It is possible to control the boundaries of these integer values, and also the increments can be adjusted, so not every possible integer value needs to be verified. By adjusting the boundaries of the values and the increments/decrements it is still possible to validate a model with a limited set of integers.

Another possible risk was finding errors that may not be present in real life. This is actually a problem that occurred during this research. For example, non-progress cycles have been found because the client indicated non-progress as such. But these problems could be marked such that the verifier would not see this as an error. Again, also in this case Promela and SPIN offer some convenient controls to mitigate such possible issues. During the verifications it is always important to be critical of the results of SPIN, and it must be reasoned whether an error should be an error, and that a successful verification should actually be successful.

When running the verifications it appears that the base-client was not very useful. Instead, the non-progress client had the same functionality and had the advantage of being useful for non-progress cycle verification and liveness verifications as well. During the development of the clients it was thought that the base client would be useful for both deadlock verification and liveness verification, however, for deadlocks the progress-labels are ignored, so the non-progress client works there as well, and for liveness it was not expected that a workaround was needed where the progress labels had to be used. If fairness could be enforced, then the base-client would have been useful when verifying the models for liveness.

8 CONCLUSION & RECOMMENDATIONS

To conclude this research the answers found to the research questions are summarized. Section 8.1 shows the answers to the research questions that have been stated in the introduction. Section 8.2 shows recommendations for further work and possible continuations of this research. Finally in section 8.3 some specific recommendations are given for Thales.

8.1 Research questions

What are characteristics of interactions between components in the O2N framework?

The characteristics of O2N are: asynchronous communication between components, complex messages sent over a network using the publish-subscribe pattern, and single-purpose components. The components are created from a specification, from which placeholder code is generated. The placeholder code is used by developers to create a complete component. In Thales systems, the command-state pattern is often used to describe a dialogue between components. By making a transition to ComMA, Thales can use ComMA specifications to replace the current component specifications. A ComMA specification specifies an interface, so a single component specification may be replaced by multiple ComMA specifications. This approach is chosen such that interfaces can eventually be reused. When using ComMA specifications it is possible to specify not only the inputs and outputs of a component, but also an abstraction of it's behaviour.

What tools are capable of verifying, and give insight in, the interaction between components in the O2N framework?

All three tools that are tested in this research are capable of verifying and giving insight in the interactions between components in the O2N middleware framework. However, in SPIN this worked much better as compared to the other tools, since SPIN proved to be the best tool for modelling and verifying the communication patterns that are used between O2N components, which is a crucial feature to have when verifying and analyzing interactions of the components. This was more difficult in mCRL2 and UPPAAL. However, mCRL2 and UPPAAL had other advantages: mCRL2 has many verification properties (using μ-calculus) and UPPAAL has an excellent user interface. These are features that would have been nice to have in SPIN as well. This information was known after the research described in the tool selection chapter (chapter 4) was performed, but this information has been confirmed by the rest of this research when the PeriodicTask and DriveControl have been modelled, which are more complex models as compared to the simple engine. Also there SPIN proved to be able to properly model these components without problems, while it was also recognized that the verification options and user interface allow some room for improvement.

To what extent can a single O2N component be modelled that it suits both formal verification and code generation?

Given a few criteria, an O2N component can be almost completely modelled such that it suits formal verification and code generation. But it does require that the O2N component is specified properly, since an O2N component in itself is just a software program. In theory, an O2N component can be a C++ program that is generated using O2N without having inputs and outputs, and having all functionality and behaviour in the implementation. Although in theory possible, this is not how components are supposed to be made. Therefore, the first criterion is that an abstraction of the behaviour must be created for the component, that displays the relevant behaviour and enough detail to find potential problems, but also not too much details to avoid a state-space explosion. The second criterion is that behaviour is not (or to a minor extent) based on types that can have infinite values, such as integers, floats or strings. If behaviour is dependent on these types it is more difficult to verify the components because the state-space gets larger. It is best if these values can be ignored in the formal model. Supporting both formal verification and code generation is possible using ComMA, which Thales uses to generate code from, and in this research it is used to generate formal models.

How can the interaction between a set of O2N components be modelled and formally verified?

The interaction between a set of O2N components can be modelled by creating exhaustive clients that interact with the modelled component. In the method more approaches have been discussed that may work, but these have not been used in this research. The exhaustive client is modelled such that it can send and receive any kind of message to the modelled component. This client also can have behaviour that is dependent on the responses it receives from the component under verification. This method of verification exploits the power of formal tools in the sense that the whole state-space can be explored.

Using the client approach, both PeriodicTask and DriveControl could be verified for deadlocks. Livelocks could be verified to a large extent as well. Invariants could not be verified, but that was not due to technical limitations, but simply because there was no use-case for it in the models. Liveness properties could be verified, although a workaround using non-progress-cycles was needed to verify fair system behaviour. This workaround for liveness did not solve all problems, therefore not all liveness properties can be verified using SPIN.

To what extent does the O2N framework need to be extended such that the interaction between a given set of O2N components can be modelled and formally verified?

The extent to which the O2N frameworks needs to be extended consists of a few separate steps. First an abstraction needs to be made for components to represent a high-level abstraction behaviour. In this research this abstraction is made using ComMA. Then from the ComMA specification a formal model could be generated in Promela, which could be analyzed using SPIN. To analyze this formal model it is necessary to also model the other side of the communication. In this research this is achieved using exhaustive clients that send every possible message, and also some variations of these clients have been applied. Using these clients the interaction can be verified fully for deadlocks, to a large extent for livelocks, fully for invariants, and partially for liveness properties. It may be the case that, by using other tools, it is possible to verify more properties, for example by using a tool that support CTL or other verification languages to support more variations of liveness properties.

8.2 Recommendations

8.2.1 Verify more dialogues

In this research only a few types of dialogues have been verified. However, there are much more options for dialogue patterns which could be studied. So far two types of dialogues have been tried: a client that never reads state messages (a monologue) and a client that only waits when it sends either a STOP message (for DriveControl) or an OFF message (for PeriodicTask). For the waiting client a deadlock was found in both cases, which was not found in the base client. Therefore, there may be more to learn by applying different clients. It could be an interesting step to determine what common client behaviour is in O2N-based systems and try to find out if there are patterns in the behaviours of the clients that influence the correctness of the system as whole.

8.2.2 More research on buffers

In this research a lot of time and effort has been spent to create custom buffer implementations that have many resemblance with the real communication channels used in O2N. However, these buffers came with a cost. Using the custom buffer implementation was one of the reasons that strong fairness could not be applied. Also the visualizations and debug traces became less clear as has been shown in Figure 5.2, because the buffer now uses a process as well. However, in the cases where a deadlock did occur, and also in the liveness and non-progress cycles eventually no data loss has been found because a progress label had to be placed at the point of data loss. Therefore, using regular promela buffers may have given the same results. If this is the case, then the implementation of the O2N components could be simplified even more. If it is aimed to analyze O2N further using SPIN then it is recommended that this is being researched to prevent unnecessary complexities in the models.

8.2.3 Research more O2N components

In this research only PeriodicTask and DriveControl have been researched. These components had some differences but also some similarities. With only two components it is hard to say which of these similarities is shared with all O2N components, and which similarities are coincidental similarities between PeriodicTask and DriveControl. This research could be improved by increasing the amount of O2N components that are being analyzed. In other words, the sample size of this research is small which may have influenced the results.

8.3 Guidelines for Thales

This section describes some guidelines that can be used by Thales to determine possible next steps to take according to this research.

8.3.1 Interface design

A question from Thales that came up often during this research, was about how developers could write ”proper” interfaces. This question is being asked in the context of formal methods and the conversation types described in the Background. An interface would be considered ”proper” if it is formally valid. In fact, this question is being handled in this research quite thoroughly in a reverse fashion. While in this research SPIN is used to analyze if an interface is correct or not, Thales wants to have guidelines on how to define interfaces that, when analyzed, are considered correct. The question can be rephrased as: ”Are there guidelines that can be followed while designing an interface such that it is formally correct?". In this section it is tried to give some of these guidelines.

Handling behaviour

Deadlocks have occurred for both PeriodicTask and DriveControl because a state message was not returned while the client expected it. It was not specified to return a state message if the component was already in the desired state. For both problems the solution was to add a missing branch from the respective if-statement, such that all possible messages where handled. A guideline could for example be to always check if all branches of if-statements have specified behaviour and return state messages if the command-state pattern is used.

Message types

A specific question on the guidelines is if it is preferred to use many messages with a small data structure, or only a few messages with a large data structure, when sending a given set of data. When looking at this from the perspective of formal tools both approaches have several advantages and disadvantages. Having many messages with a small data structure may bring an increase in parallelism. When the messages can come in in any order this can greatly increase the state-space of the component. A disadvantage of large messages on the other hand is that often only a part of the message is being used. From a formal viewpoint a large data structure is easier to model, as there are less channels to model. A large data structure makes the whole model less flexible which is expected to give a positive influence on the state-space. However, every message has to be transmitted at the rate of the member variable that must be updated most often, so in this large data structure often variables will be updated that are not changed. To conclude: with a large data structure the dialogue between components gets simpler, there is no flexibility on the rate of the messages so there can be a lot of unused traffic on the network, but there are no problems with the order of messages coming in. Which options is best is dependent on the application itself.

Separate variables used for data and behaviour

Integers, floats and strings can be difficult to model because of their effect on the state-space. But in a real application it is unavoidable to not send such values. To make it easier to verify a component it is useful if these values can be removed from the formal model without consequence. In other words, the integers/floats/strings are used as data by the implementation, while other variables indicate the behaviour of the system. An example of this is shown in DriveControl. The real values did not have any effect on the behaviour so they could be removed to keep the state-space small.

8.3.2 Using SPIN

This research has shown that it is possible to convert a ComMA specification to a Promela process. This section shows a proposal on how this process can be automated and shows where potential problems may occur based on findings in this research. An overview of the process is shown in Figure 8.1. The basic idea is to convert the specification from ComMA to Promela through a metamodel. Using a metamodel it is possible to indicate hints on how the Promela specification should be generated. The first step is generating a metamodel from the ComMA specification. This is possible using Xtext [9] and Xtend [10], in which ComMA itself is created as well. However, from the ComMA specification some details are missing that are relevant when creating the Promela processes. There are cases where such an action is

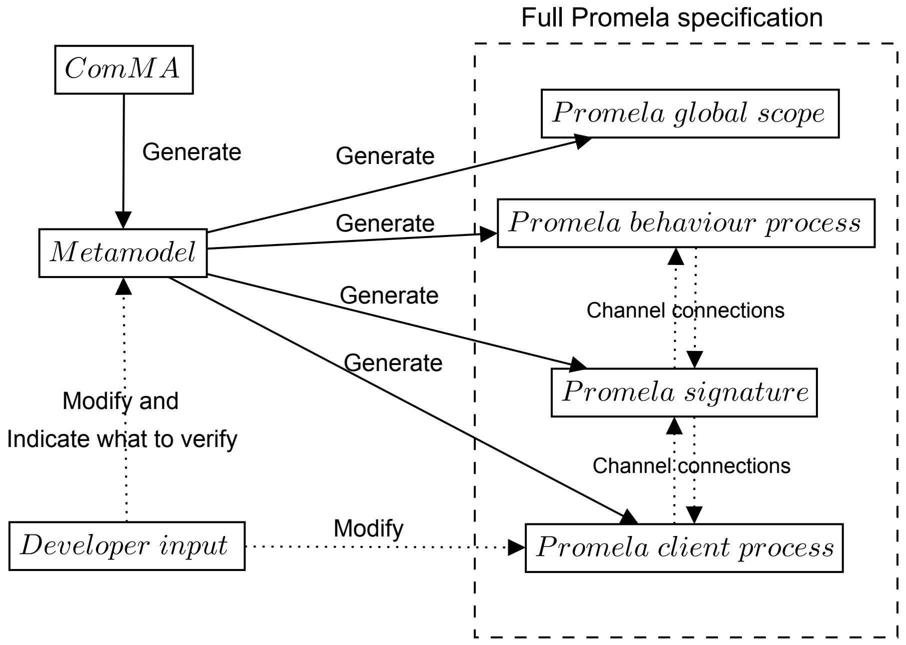


Figure 8.1: Overview of an automated process to convert ComMA to Promela.

needed. These are shown below. For convenience, these properties will now be called the *conversion settings*.

- The type of buffer and it's size for each channel.
- The type of verification.
- The effect of a dialogue on the client.
- (In case of verification for liveness and invariants) LTL properties to be used for verification.
- (In case of global integers/integers being transmitted) bounds and increments/decrements of the value of the respective integer.

Making the conversion process easy for these cases can be done by using either one or a combination of the following methods:

- **A developer modifies the Metamodel before the specification is ready to use:** With this approach a developer has to do some manual changes to the metamodel, by indicating in the model what buffers must be used, the type of verification etc. Using this approach an interface for the developer must be made to make these changes to the metamodel. An advantage of this approach is that the ComMA specification does not have to be altered.

- **Extend the ComMA specification with 'hints':** Using this approach a developer can add hints to the ComMA specifications. For example annotate the ComMA signature with comments that specify the buffer type and size. Using this approach the changes are more persistent, but the ComMA specification needs to be altered.

If the metamodel contains the relevant information from the comma specification, and it is clear what should be generated by hints from the developer, it is time to generate the Promela models. At this point the metamodel should contain:

- All variables from the ComMA specification.
- The signatures from the ComMA specification (it is recommended to keep the distinction between. 'notification' and 'signal' to describe incoming and outgoing channels).
- The states and transitions (behaviour) from the ComMA specification.
- The set of *conversion settings*.

Creating the main model

Using the conversion table in Appendix F the variables, states and transitions can be generated in Promela. These are not dependent on other properties of the model. At this point, it must be determined what buffers are used. For synchronous communication (buffer size 0) the regular Promela rendezvous buffers can be used. For asynchronous communication there is a choice to use blocking buffers or non-blocking buffers. Based on the indication of the developer the respective buffers can be generated. It is not recommended to use non-blocking buffers with the `-a` setting in SPIN: this causes new messages to be deleted when the buffer is full which can give erroneous verification results. Instead the custom buffers that are shown in this thesis provide a correct alternative by removing the oldest messages instead.

Creating clients

Like shown in this thesis, a client can be setup to send every possible message. From the metamodel it can be observed what types have to be sent by the client (these are the types used by the incoming signals. This is the reason to remember in the metamodel whether a channel is incoming or outgoing). Also global variables that are used by the model have to be altered. A risk at this point is that many variables that are transmitted by the client are not having an impact on the behaviour of the system. by determining whether the variable is present in the behaviour it could be determined if it is used, and if it not used, it can be ignored to reduce the size of the state-space. Another important detail is that the client should also be able to receive on the outgoing channels of the model under verification. Otherwise there will always be a buffer overflow, which will result in incorrect results for Liveness and Non-progress cycles. A client may need manual editing after it being generated. This is because a client may also have a given sort of behaviour. In this thesis it is shown that a deadlock is found using a 'waiting'-client, while the same deadlock was not found with the 'base'-client. Therefore, the behaviour of a client also matters. This cannot be deduced from the initial ComMA specification.

Creating the main file

The main file is responsible for starting the relevant processes, and by setting some global variables. In the examples used in this research the main file is used to enable/disable progress labels in the buffers, and to select a client. At this point also the *conversion settings* are needed. The proper client needs to be selected for the desired verification, and the progress labels must be enabled or disabled.

Running verifications

When running verifications, it is recommended to only verify for deadlocks and invariants. This is because checking for deadlocks and invariants is easy when a model has been generated. Verifying for livelocks is also possible, but it already requires some more work and insight in the interaction, because when SPIN finds a non-progress cycle it must be carefully examined whether the behaviour is as specified or not. It may then be required to manually change the model with progress labels, which is something that needs to be done carefully. Verifying for liveness is not recommended, because it requires a deep understanding of the interaction, and also of the formal tool and language. When SPIN is used there are many challenges at this point and a mistake is easily made. In this case, that can take up a lot of time. If there is a method to enforce fairness constraints, then it may become easier to verify for liveness, but in this research no such method was found.

Using concurrency (swarm runs)

SPIN also supports ’swarm verification’ which allows verification runs to be distributed along multiple computers. In this thesis this functionality is ignored but when it is desired to verify larger and more complex systems then this may provide a solution to access more processing power.

How this can fit in an industrial workflow

Using the conversion method described above to convert a ComMA specification to SPIN, using an automated process, can be used in an industrial workflow. If a tool exists to automate this process the workflow of analyzing ComMA specifications could be as follows:

- A developer creates A ComMA specification.
- It annotates the ComMA specification with ’hints’ for conversion to Promela.
- A tool creates a promela model and runs the verification.
- If the verification fails, the developer must find the problem and possibly write a new specification, or change some settings in how the formal model is generated.

Bibliography

[1] M. van Steen and A.S. Tanenbaum. *Distributed Systems, 3rd ed.* Amazon, 2017. distributed-systems.net.

[2] W.M.P. Aalst, van der et al. *Service interaction : patterns, formalization, and analysis*, pages 42–88. Lecture Notes in Computer Science. Springer, Germany, 2009.

[3] Benny Akesson, Jack Sleuters, Sven Weiss, and Ronald Begeer. Towards continuous evolution through automatic detection and correction of service incompatibilities. *CEUR Workshop Proceedings*, 2019.

[4] Verum dezyne. https://verum.com/ - Accessed at 07-08-2020.

[5] Rutger van Beusekom et al. Formalising the dezyne modelling language in mcrl2. In *FMICS-AVoCS*, 2017.

[6] Olav Bunte et al. The mcrl2 toolset for analysing concurrent systems. In Tomáš Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 21–39, Cham, 2019. Springer International Publishing.

[7] Ivan Kurtev et al. *Runtime Monitoring Based on Interface Specifications*, pages 335–356. Springer International Publishing, Cham, 2017.

[8] Ivan Kurtev et al. Integrating interface modeling and analysis in an industrial setting. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, page 345–352, Setubal, PRT, 2017. SCITEPRESS - Science and Technology Publications, Lda.

[9] xtext. www.eclipse.org/Xtext/ - Accessed at 05-02-2020.

[10] xtend. www.eclipse.org/xtend - Accessed at 05-02-2020.

[11] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[12] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 43 years of actors: A taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2016, page 31–40, New York, NY, USA, 2016. Association for Computing Machinery.

[13] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Caf - the c++ actor framework for scalable and resource-efficient applications. In *Proceedings of the 4th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '14, page 15–28, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Libactor. https://github.com/airplug/libactor - Accessed at 12-07-2020.

[15] Robin Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, January 1993.

[16] Marjan Sirjani and Ali Movaghar. An actor-based model for formal modelling of reactive systems: Rebeca. *Tehran, Iran, Tech. Rep. CS-TR-80-01*, 01 2001.

[17] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[18] Weng Jie Thong and Mohamed Ameedeen. A survey of petri net tools. *Lecture Notes in Electrical Engineering*, 315:537–551, 01 2015.

[19] Ichiro Suzuki and Tadao Murata. A method for stepwise refinement and abstraction of petri nets. *J. Comput. Syst. Sci.*, 27:51–76, 1983.

[20] D. Bera et al. *A component framework where port compatibility implies weak termination*. Computer science reports. Technische Universiteit Eindhoven, 2011.

[21] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2–3):131–146, May 2005.

[22] Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, USA, 1999.

[23] Luca Aceto. Some of my favourite results in classic process algebra. *Bulletin of the EATCS*, 81:90–108, 2003.

[24] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.

[25] J. A. Bergstra and J. W. Klop. Acp$_\tau$ a universal axiom system for process specification. In *Algebraic Methods: Theory, Tools and Applications*, pages 445–463, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.

[26] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984.

[27] C. A. R. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*, 1980.

[28] Rajeev Alur. Timed automata. *In: Proc 11th Int Confer on Computer-Aided Verification. Lect Not ComputSci*, 1633, 03 2001.

[29] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57, 1977.

[30] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 169–181, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg.

[31] E. Emerson and Joseph Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33:151–178, 01 1986.

[32] mcrl2 - analysing system behaviour - homepage.
https://www.mcrl2.org/web/user_manual/index.html - accessed at 04-02-2020.

[33] Jan Friso Groote and Tim Willemse. Parameterised boolean equation systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory*, pages 308–324, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[34] K. G. Larsen, P. Pettersson, and Wang Yi. Compositional and symbolic model-checking of real-time systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 76–87, Dec 1995.

[35] M. Hendriks et al. Uppaal 4.0. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*, pages 125–126, Sep. 2006.

[36] Uppaal. www.uppaal.org - accessed at 10-02-2020.

[37] Spin website. http://spinroot.com/spin/whatispin.html - Accessed at 06-04-2020.

[38] graphviz. https://www.graphviz.org/ - Accessed at 06-04-2020.

[39] jspin source code. https://github.com/motib/jspin - Accessed at 06-04-2020.

[40] Spin manual on float. http://spinroot.com/spin/Man/float.html - Accessed at 14-05-2020.

[41] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 01 2004.

[42] Nusmv homepage - accessed at 14-08-2020. http://nusmv.fbk.eu/.

[43] Yong Jiang and Zongyan Qiu. S2n: Model transformation from spin to nusmv. In Alastair Donaldson and David Parker, editors, *Model Checking Software*, pages 255–260, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

APPENDICES

A Verifying Promela models using iSPIN

Confidential

B mCRL2 ”simple engine” specification

Confidential

C Promela ”simple engine” specification

Confidential

D PeriodicTask ComMA model

Confidential

E DriveControl ComMA model

Confidential

F Comma/Promela conversion table

Confidential

G PeriodicTask Promela model

Confidential

H DriveControl Promela model

Confidential

I Buffer progress labels

Confidential