ICT GROUP

Transforming how we develop highly reliable and highly robust software with Model-Driven Engineering

ICT High Tech - Center of Excellence

Abstract Software systems tend to increase in complexity as they grow and age, making it difficult to maintain an understanding of the full system. Changes in requirements or personnel over time makes it even harder to oversee a system's functionality. We propose Model-Driven Engineering (MDE) – specifically state-based modelling – as a solution to deal with the complexity of modern systems, and to produce correct, robust and maintainable software. Despite challenges acquiring engineers with experience and expertise for these relatively new tools, from our experience in applying these tools, we believe MDE can be valuable to create high quality, maintainable software, even for extremely complex and evolving systems.

1 Problem Statement

Software – and especially software controlling hardware – has a complexity problem. Adding more variables into a system's behaviour will increase the possible states of the software exponentially. This is why it is easy to maintain small programs, but very hard to change large pieces of software: all the variables in the software combinatorially multiply into an extremely large state-space. This phenomenon is called state space explosion, and will be familiar to any developer with experience working on large code-bases.

As a result of this complexity, it can become prohibitively difficult to oversee full system behaviour, as it involves interaction between multiple different components that each have their own behaviour. This makes it impossible for a human mind to oversee full system behaviour and all edge cases of large systems. Unfortunately, this understanding of the full system behaviour is required to effectively create and maintain our software.

2 Model-Driven Engineering

The solution we present for these problems is Model- Driven Engineering (MDE). The central idea is to create an abstract model of the software, and to let a computer automatically verify the model's correctness, instead of putting the burden of understanding the entire system on the developer. This verification mathematically checks if the model conforms to its specification, and if it conforms to the specification of other components it communicates with. Once the model is successfully validated, code is automatically generated out of the model. Since the model is verified to be correct and code generation is fully automatic, the resulting code can also be assumed to be correct.

2.1 State-based behaviour modelling

The definition of MDE as given above provides room for a wide range of possible ways of modelling and verifying systems and software. For the rest of this paper we will narrow the scope to state-based modelling in combination with formal verification. This means modelling a piece of software as a set of states, and the different transitions between these states as a result of external stimuli.

Consider the step the programming field took when moving from low-level languages like assembly, into more strictly typed languages like C or C++. The addition of the type system meant that it is no longer possible to assign values of incorrect types. It is no longer possible to call functions with the wrong number of arguments. No longer can we accidentally call a function that does not exist. This stricter typing of the code has eliminated entire categories of bugs, which are now caught by the compiler.

State-based behaviour modelling can be seen as the next step in this direction. On top of strictly specifying the types of variables, functions and interfaces, the type system also contains the behaviour of these entities. This means the 'compiler' (or the verifier in this case), will prevent incorrect usage of interfaces not only from a type perspective, but also from a behavioural perspective.

2.2 Modelling behaviour

Consider the example model in Figure 1 describing the state behaviour of a door:

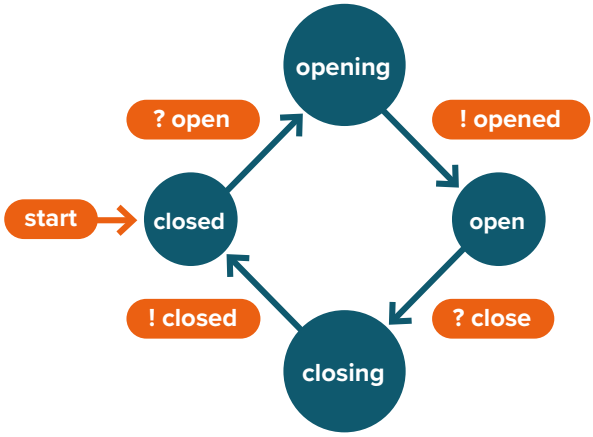

Figure 1: example state model of a door

The model in Figure 1 has four states: *closed, opening, open and closing*. Furthermore, it specifies that the action **open** can only be done while the door is closed, leading to state opening ; the action **close** can only be done while the door is *open*, leading to state *closing* ; and finally that the states *opening and closing* are guaranteed to at some point send out events **opened** and **closed** before ending up in the states *open or closed*, respectively.

This interface model is a contract between two components. On one hand, the door promises to provide this behaviour, and is **proven** by the verification to indeed do so. On the other hand, components using the door promise they will use this interface correctly, and are **proven** by the verification to do so. This verification ensures that this contract of behaviour is **always** adhered to. If, for example, at any point the software tries to close the door while it is not in the open state, the verification will show an example of a trace in which this happens, and throws an error. This means that all edge cases must be dealt with during development, making it impossible to 'forget' to think about some scenario in which your contracts are violated.

2.3 Modelling implementations

Implementations for interfaces like the one shown in Figure 1 also come in the form of state machines. Both models are written as 'code' in the same Domain Specific Language. Where an interface describes one state model with its states and events, the implementation model declares any number of required or provided ports, each with some interface type. The implementation state model then specifies how its internal state changes as a result of events from the various ports, and which events to send out to its connected components in response. For example the implementation of the door might provide the interface in Figure 1, and require an a port with interface for the actuator that opens/closes the door, and two ports with interfaces for sensors that detect if the door is fully opened or closed.

If **any** situation is reachable in which the implementation violates interface behaviour on any of its ports – no matter how long it might take to get there – the verifier will flag this violation and force the developer to handle this correctly.

2.4 Systems as composition of models

Now that we have interfaces that describe behaviour and components with provided and required ports (of some interface type), we can define our software as a system of inter-connected components. Required ports are connected to provided ports of other components to compose a system of multiple components, each with their own responsibilities. The remaining ports are connected to glue code to connect the system the the outside world.

An example of such a system can be found in Figure 2. This example shows how the system of an airlock can be composed of multiple components (with possibly multiple instances of the same type), building up more complex behaviour by composing multiple simpler models. The example shows the basic components of the system at the bottom: a sensor and a motor for each of the two doors of the airlock. The door components encapsulates the logic of using motors and sensors, and provides a simple open/close interface. Then the lock component encapsulates the correct behaviour of the airlock, for example verifying that no situation exists where both doors are open at the same time. This lock component can then provide a simple transferIn/transferOut interface that allows moving items from one side of the airlock to the other side. This interface can then again be used by a higher level model that is for example arranging the flow of items through the system, needing to pass through the airlock at some point.

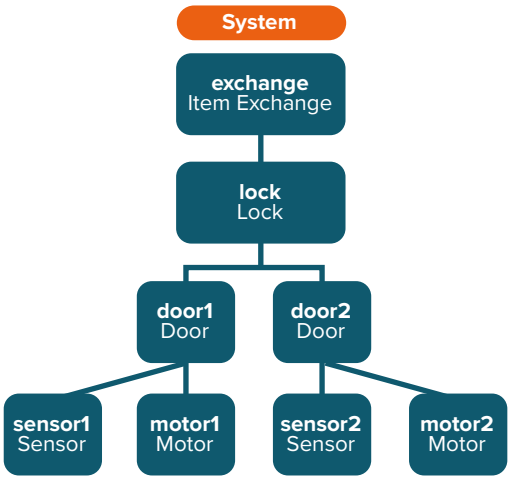

Figure 2: A hierarchical system of components, each modelled as a state model. Image generated with Popili[1]

Every component in this system will have its behaviour verified to always adhere to its provided and required interfaces, leading to extremely robust and predictable software, even when making large changes to the components internally. The verification always makes sure that: there is no regression on the provided interfaces; and that all consuming components are correctly using the new interface.

3 Applicability

As the name suggests, state-based modelling is especially applicable to systems that have a well-defined set of states and transitions between them. Usually these systems are very interactive, they start doing something by sending a stimulus to the outside world, then they wait for some kind of result back.

This means that state-based modelling is especially suited to systems with real-world hardware components. In these systems, it is very natural to model the state behaviour of low level components as state models. For example a motor that can be turned on or off, and then waiting for a sensor to switch, causing the motor to be turned off again.

Having hardware is no requirement however, MDE can also be used purely digitally, as long as the system has a set of well-defined states and events.

3.1 The difference between verification and testing

It is important to discuss the difference between verification and testing. Both serve the purpose of verifying the system's requirements, and to protect from regressions. The main difference is that tests are limited to the pre-defined set of scenarios: if some sequence of actions is not covered by any tests, regression or failure to meet specification will not be noticed.

In contrast, verification always considers **all** possible scenarios in the system, even scenarios that are infinitely long. While testing actually executes the software to assert its behaviour, verification will instead try to create a mathematical proof based on the model. If this proof cannot be created, a counter-example is produced showing where the system is deviating from its specification. If a verification proof is produced, that means the system is *guaranteed* to always behave according to its specification.

4 Benefits

4.1 More natural way of expressing behaviour

For many software systems, especially ones controlling hardware, it is natural to think about the functionality of the system in terms of states and transitions between those states. This is generally how engineers and architects will design the system, and how requirements are specified.

However, when it comes to programming these systems, the general-purpose programming languages we use (like C/C++) are not especially tailored to state-driven behaviour. In contrast, state modelling languages like Coco (part of Popili[1]) or Dezyne[2] are designed around modelling your system this way. This results in clearer code that is easier to understand and maintain. Additionally, these specific-purpose languages allow for useful visualisations that are difficult to create out of general purpose languages, see Figure 3 for an example of a state transition diagram generated from a formal model.

4.2 Highly reliable and robust software

Verification ensures by proof that contracts are always adhered to, even in infinitely long use of the system. This provides a high level of certainty when modifying implementation, all contracts in the system are still adhered to after the modification.

In practice, we see that the use of model-driven engineering as discussed in this paper leads to very low defect rates in the software it produces. Due to the formal verification, it is impossible to forget to handle certain edge cases, hence there are very few bugs of use-cases that were not considered during development. Because of the verification it is also impossible to accidentally modify the behaviour on your interface, leading to a low chance of regression when changing the software, especially in combination with a good set of tests.

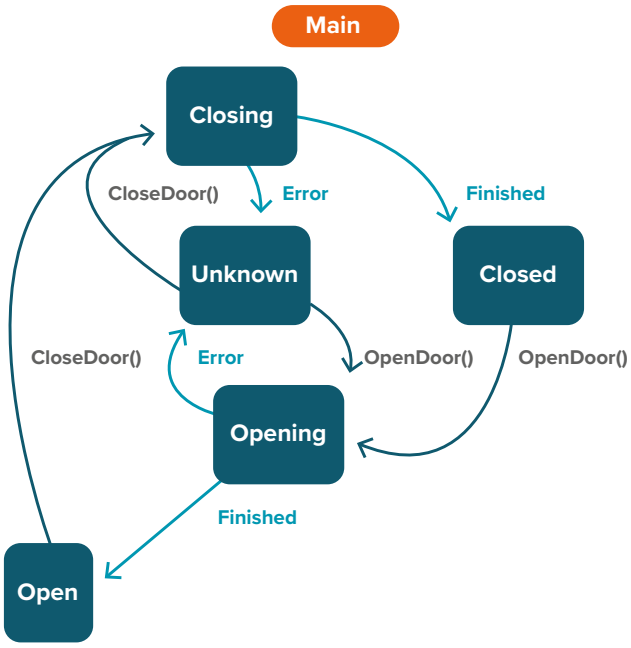

Figure 3: A state model of a door interface.

4.3 Requirements-focused test suite

As a result of the verification of our models and generating correct-by-construction code, there is no need to have low-level unit tests of individual components. What we see in practice is that the test suites in MDE projects tend to more closely reflect the requirements, and test a large vertical slice of the system at once, almost like an integration test. This means that as the system is built, the test suite created will mostly cover user scenarios as described by system requirements, as a collaboration of multiple software modules. As a result, the test suite is focused on safeguarding the requirements of the system. That is not to say there are no longer any unit tests: any glue or data-processing code external to the models does of course still benefit from a good test suite.

Model	lines of code	#ports	#states	#transitions	verification time
Oven	143	2	288	648	0.03s
Sorting line	100	2	1,765	4,154	0.01s
Processing station	186	5	16,439	33,314	0.1s
Supervisor	975	9	22,552,400	39,302,925	74.4s

Table 1: Comparing the verification times of various Coco[1] models

4.4 Early feedback on requirements

As mentioned before, the formal verification considers all possible combinations of events and use-cases. This means that if all interfaces are modelled according to their requirements, and the verification still manages to find a problem in their integration, that might indicate a gap in the requirements. In practice, this means requirements issues are found much earlier: already during the design or development phases, rather than during the testing of the system, or even worse after delivery.

This leads to faster development iteration, fewer delays, and since cost of defects are lower the earlier the defect is found[3], this also reduces costs.

5 Challenges

5.1 Verification times

The main challenge when working with state-based model-driven engineering is the verification times of your models. The formal verification of your models considers all possible combinations of events, for an infinite time period. The verification algorithm scales exponentially with its input size: this means that depending on the complexity of the interfaces, the verification might have to (in exceptional cases) check millions or sometimes even billions of states. This requires performant tooling and avoiding duplicate work. For this reason it is also a bad idea to put all behaviour into a single model, as the combinatoric multiplication of inputs will cause your verification time to explode. Intelligently distributing behaviour over multiple models therefore does not only serve to keep the code readable and maintainable, but can also drastically reduce time spent waiting for verification. Table 1 gives an indication of the verification times of our Coco[1] models, and how the verification time scales with the complexity of the models.

5.2 Different way of working need for expertise

Working with model-driven engineering tools is different to 'regular' programming: while the way of thinking about the system (in terms of states and actions) is the same, these are new tools to learn, and not many developers have existing experience using these techniques. Next to needing to learn the modelling language, it also requires knowledge and experience about the ecosystem in which the tools are being used, and how they integrate into the existing software and development environment. Furthermore, it takes experience to understand how to best architect your models in a way that keeps them correct, expressing/verifying the right things, and how to keep the verification times low (see section 5.1).

5.3 Reliance on tools

While the models produced for tools such as Popili[1] or Dezyne[2] are fairly 'generic' descriptions of state models that are valid in any context, the languages are tool-specific. Without the tools and their code generation, the models are only useful as a description, but you lose the benefits of the verification and code generation. Therefore there is some vendor lock-in when using these tools, in a way there is not when using something as generic as C++. Since models are highly abstract and verified to be correct, it is usually not that difficult to translate models from one language into another language, if really needed. Dezyne[2] has also open-sourced their verification and code generation, so it could be forked and maintained, were the current maintainers to disappear.

6 Conclusion

To conclude, we see many benefits from using model-driven engineering: Teams using MDE seem to be able to iterate quickly, deliver very robust code with low defect rates, and have a deep understanding of the behaviour of the system they are working on. Furthermore, we see that state modelling clearly describes system behaviour, and the models with their generated diagrams are a nice way to communicate behaviour to stakeholders.

The challenges with MDE mostly come down to experience. Managing the complexity of the models, designing a useful architecture, knowing what to verify and what not to, as well as integrating the tools and way of working into the existing software workflow takes experience, where experienced people are rare to find. With that in mind, the ICT Center of Excellence has been building this experience for a while now, and has been applying MDE at our customers. We want to help spread the use of what in our view is a very valuable technology. If you are interested, do check out Popili[1] and Dezyne[2], and feel free to contact us for questions, demos, trainings or consulting!

References

[1] *Popili: State-based Model-Driven Engineering tool by Cocotec. https://cocotec.io/ Their modelling language is named Coco.*

[2] *Dezyne: State-based Model-Driven Engineering tool by Verum. https://dezyne.org/.*

[3] *N. I. of Standards & Technology. The economic impacts of inadequate infrastructure for software testing. Technical report, NIST, 2002.*

ICT High Tech
Center of Excellence
centerofexcellence@ict.nl
+31 (0)88 908 2000

ICT High Tech