CASE STUDY

# Smooth adoption of Verum's Dezyne to model software for a service tool

Dezyne is a software development tool developed by Verum, based on a Model Driven Engineering approach. Dezyne is primarily used for designing complex software systems. Due to its built-in formal verification capability, Dezyne is especially suited for safety critical systems used in aerospace, automotive, chip manufacturing and the medical industry.

Companies using Dezyne have reduced their time to market by 20%, while reducing software bugs by 25% and costs by 50%. Dezyne is one of the Model Driven Engineering toolsets ICT Group works with, reshaping traditional approaches of software design. Using today's technology we educate tomorrow's software engineers.

Below we describe the functionality of Dezyne using a development project: the development of a service tool for a startup/shutdown controller. This project was carried out by our software engineer Saurav Paul.

ICT smarter solutions

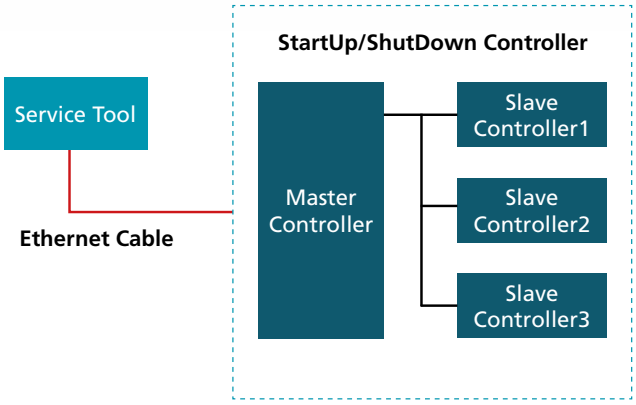
## First experiences with Dezyne

Saurav Paul was introduced to the methodology of Model Driven Engineering during his Master study Embedded Systems at the Technical University of Delft. When attending a Dezyne Community meeting shortly after his graduation, he recognized a strong parallel between Dezyne and what he had learned at university. Enthusiastic about Dezyne's possibilities, he started using the toolset for a small software design project. "According to me Model Driven Engineering is the future of software development," Saurav says. "So I was eager to start using Dezyne."

## Using Dezyne in a project commissioned by Philips

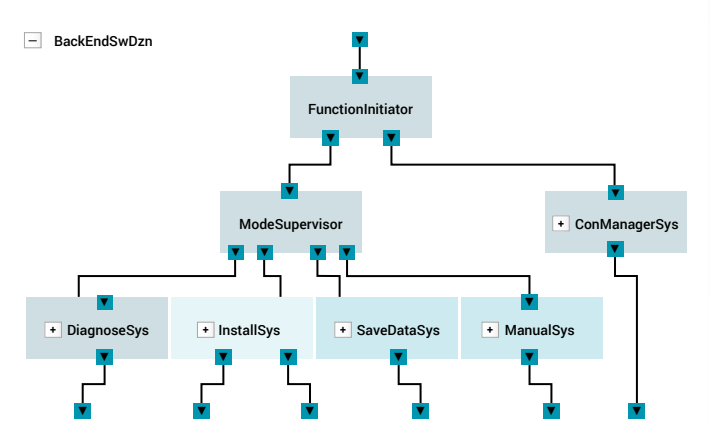
After his graduation Saurav Paul started working for ICT Group. He was chosen as the software engineer in a project commissioned by Philips. In this project Dezyne was used as the primary toolset.

Philips develops x-ray systems for image guided medical procedures. The x-ray machines are powered by complex startup/shutdown controllers that ensure reliability and safety. As the behavior of these controllers is complex, a dedicated service tool is used to check and service them. Saurav designed the software for this new service tool, using Dezyne.




*The service tool checks the behavior of the startup/shutdown controller*

Dezyne's approach is to start by creating a model that captures the behavior of the soft-ware system. The model serves as a means of communication between software designer(s) and stakeholders. It ensures that the requirements formulated by the stakeholders are thorough, complete and effectively implemented. Dezyne also allows the software engineer to simulate software behavior at every step of the development process, which helps to verify whether the system meets the requirements. Once tested and verified, computer code is automatically generated from the model with the press of a button.

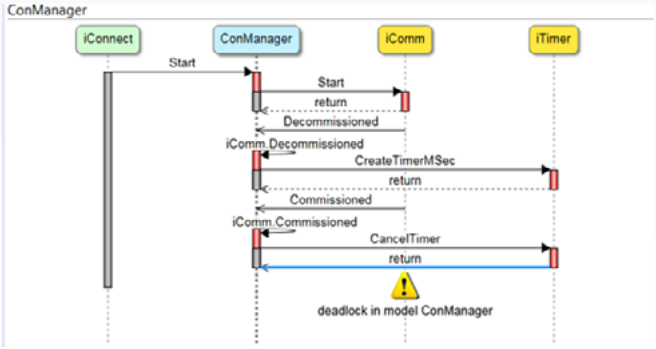



*Static overview of the Dezyne model showing the relations between the various compo-nents. Code is automatically generated from the model.*

## Dezyne's unique feature: formal verification

What distinguishes Dezyne from other Model Driven Engineering toolsets is its built-in formal verification engine. With one click the model is scanned for errors and checks for unwanted properties like deadlocks, livelocks, incomplete mapping of events and re-sponses, race conditions, illegal actions and compliance. When something is wrong, the verification engine not only finds the error but also pinpoints its exact location within the model.

Dezyne's formal verification functionality greatly helped Saurav Paul in designing the software for the service tool. "When you commit an error while programming in the traditional way, it may be detected by the compiler. But runtime errors are very difficult to detect and debug. Yet Dezyne shows you immediately where the mistake is located. After solving the error you can test and verify very quickly from the model. In traditional programming this takes much longer."




*Screenshot of a 'deadlock' error in the Dezyne model. Dezyne's formal verification func-tionality provides a sequence trace pinpointing the location of an error.*

## Ensuring correct functionality by combining TDD and Model Driven Engineering

The service tool Saurav Paul worked on has four functionalities or features: 'manual mode', 'diagnose', 'firmware install' and 'save data'. In order to ensure whether these functionalities were working properly, Saurav used a combination of Test Driven Devel-opment (TDD) and Model Driven Engineering.

In traditional software development testing takes up a lot of time. Due to time pressure thorough testing is therefore often not feasible. This increases the likelihood of errors showing up later when the software is implemented. Yet ideally, thorough testing is necessary to ensure that the software fulfils the requirements of the system. In Dezyne test-ing does not suffer from the same problems as it does in traditional software development.

What sets Dezyne apart is that the testing phase is much shorter and more efficient. The components comprising a software system do not have to be tested very thoroughly, since Dezyne's built-in formal verification engine already checks for unwanted properties, such as deadlocks and race conditions. The only thing left to be tested is the functionality of the entire software system. In the future Verum plans to add more functional property checking, so that eventually functional correctness can be formally verified and does not need to be tested separately.

```
component FunctionInitiator
{
    provides IBSw iBackEndSW;

    requires IMode iMode;
    requires IConnect iConnection;

    behaviour
    {
        enum States {IDLE,CONNECTED, DISCONNECTED, CONNECTING };
        enum Mode {MANUAL, INSTALL, DIAGNOSE, SAVEDATA, NONE};
        enum Wait   {TRUE,FALSE};
        enum Received {YES,NO};

        string modeResponse;

        States state = States.IDLE;
        Mode mode = Mode.NONE;
        Mode lastMode = Mode.NONE;
        Wait wait = Wait.FALSE;
        Received decomReceived = Received.NO;


        [state.IDLE]
        {
            on iBackEndSW.Connect(mc, t):
            {
                iConnection.Start(mc, t);
                state = States.CONNECTING;
            }
            on iMode.Decommissioned(msg):
```

*Screenshot of a software component modeled in Dezyne text based modeling language.*

Saurav designed the software for the four functionalities of the service tool by modeling them in Dezyne. He then ran unit tests in order to verify the intended behaviour. When errors showed up during the tests, he was able to correct them quickly in the model itself before re-testing the whole system.

This combination of testing and (re)modelling results in quick and efficient adaptations, even late into the design process. Saurav Paul: "The major difference in Dezyne is that you spend more time on design and less time on testing. Because of its formally verified implementation, Dezyne ensures quality and provides more opportunities to find and fix design issues in the initial phase of development."

## Programming by hand

When the features of the service tool were successfully tested, Saurav was able to pro-duce C# code with a single mouse click. Apart from C# Dezyne also supports the lan-guages C, C++, Java, JavaScript and Python.

Dezyne is not suited to design software for algorithms or to model data flow. These fea-tures cannot be modeled using Dezyne, but have to be programmed by hand. Saurav Paul: "Whenever the software had to take data based decisions or read data, we used handwritten code instead of the model." Handwritten code was also used to glue the software to the pre-existing code of the outside environment, the Startup/Shutdown con-troller.

Saurav then ran final integration tests to see whether the new software was working properly with the startup/ shutdown controller. This ensured the succesful completion of the project.

## Project results

In the Philips project the usage of Dezyne led to several positive results:

- Integrating the new software design went much more smoothly than a traditional software design approach. Saurav Paul: "The integration phase was relatively straightforward, because much of the software was already verified in the implementation phase. So you know that any errors that show up have to do only with the integration process and not with errors in the software itself."
- Functional testing went smoother as well. Because of its formal verification functionality Dezyne eliminates common design errors like deadlocks and race conditions. This greatly increases the speed and efficiency of the testing phase, as many unwanted properties had already been filtered out.
- The software resulted in a more reliable and efficient service tool, which handled errors in a better way. The time required to install new firmware on the startup/ shutdown controller using the new service tool was significantly reduced and stakeholders were happy with the functionality and usability of the new tool.

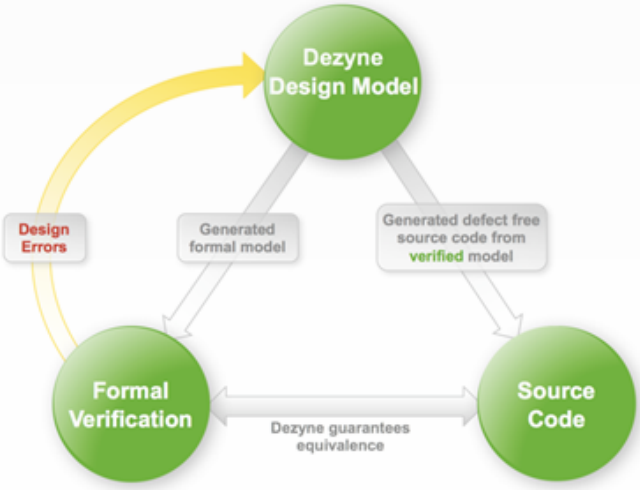
## Lessons learned

While working on developing software for the service tool of the Startup/Shutdown con-troller Saurav Paul learned valuable lessons about the Dezyne toolset:

- Modeling a software system as a composition of small and simple components that in-teract with each other works better than creating one big, complex model. Saurav Paul: "When you work with smaller components it becomes much easier to test, debug errors or add new functionality later on."
- Deciding when to use Dezyne generated code or handwritten code. Saurav Paul: "In traditional software development control logic and data handling are not separated, but are part of the code itself. In Dezyne you are forced to make the separation. I learned to make the distinction when to use one or the other."
- With Dezyne less bugs and issues showed up in the later stages of development when compared to traditional methods. "Especially during integration there are less bugs, because Dezyne found most of those errors before."
- When using Dezyne more time is spent on formulating requirements and making the design, but this time is gained back in the later stages. "You actually have more

questions when speaking to the stakeholders, especially about the 'what if' cases. This makes the requirements, and also the later software design clearer and more robust leading to greater consumer satisfaction."

- Most importantly, Saurav says, Dezyne minimises human programming and communication errors, saving time and improving efficiency. "Dezyne closes the triangle between requirements, formal verification and code generation. It provides a much better guarantee for a software solution that fits the stakeholder's needs."




*Dezyne closes the triangle between requirements, formal verification and code generation.*

## Dezyne: the choice of Philips and others

Our client Philips made a conscious choice to use Dezyne in this project. Because of the complexity of the the control logic of the startup/shutdown controller, the client preferred Dezyne's formal verification in order to model, check and verify the functionality adequately.

## Contacts:

Saurav Paul
Software Engineer
ICT- Hitech Unit
E: saurav.paul@ict.nl

Ronald Wiericx
Operations Manager
ICT- Hitech Unit
E: ronald.wiericx@ict.nl

Professor Doctor Dorgelolaan 30
5613 AM Eindhoven

Curious about the possibilities of Model Driven Engineering and Verum Dezyne? Feel free to contact our software engineer Saurav Paul to discuss the opportunities for your organisation.