DEVS MODELLING AND SIMULATION OF A MULTI-PARADIGM MODELLING TOOL

Yentl Van Tendeloo

University of Antwerp, Belgium
Yentl.VanTendeloo@uantwerpen.be

Hans Vangheluwe

University of Antwerp, Belgium
Flanders Make, Belgium
McGill University, Montréal, Canada
hv@cs.mcgill.ca

Multi-Paradigm Modelling (MPM) has been proposed to tackle the complexities of the systems we build today. MPM proposes to explicitly model all relevant aspects of the system, at the right level(s) of abstraction, using the most appropriate formalism(s), while explicitly modelling the process. This imposes stringent requirements on any tool supporting MPM, such as support for distributed execution, support for multiple users, all while having acceptable performance. In this paper, we wish to aid the development of such complex tools by explicitly modelling the tool using the Parallel DEVS formalism, thereby applying the MPM approach ourselves. This model is applied in two domains: performance evaluation and model debugging. For performance, the DEVS model is used for what-if analysis and automated benchmarks. For debugging, DEVS debuggers are used instead of code debuggers, raising the level of abstraction. In both cases, DEVS yields deterministic execution, aiding in reproducibility.

Keywords: DEVS, debugging, performance, Multi-Paradigm Modelling

1 INTRODUCTION

The complexity of engineered systems is increasing, mainly due to their heterogeneity. Multi-Paradigm Modelling (MPM) (Mosterman and Vangheluwe 2004, Vangheluwe et al. 2002) has been proposed to tackle the complexities found in such heterogeneous systems, by explicitly modelling all relevant aspects of the system at the right level(s) of abstraction, using the most appropriate formalism(s), while explicitly modelling the process. Tools for MPM, however, have to support a wide variety of features, such as interaction with multiple (existing) tools, multiple types of interface, and multiple users working jointly on a single system. These high-level requirements result in several lower-level requirements: the system must be distributed (to handle the "service" nature), must support concurrency (multiple users, each with their own domain of expertise), and performance must be monitored and optimized (to handle complex models). These low-level requirements are becoming more and more frequent in the software systems that we build today, though we will focus on tools for MPM in this paper, and specifically our prototype tool for MPM: the Modelverse (Van Tendeloo and Vangheluwe 2017b). We identify and address two types of problems encountered during the development of such complex software systems: performance evaluation and model debugging.

The first problem is related to performance evaluation. As the system is performance-critical, it is necessary for there to be benchmarks, or at least some way of analyzing performance. However, such benchmarks have limitations: they are executed on a shared-resource machine, meaning that performance results depend on the interleaving with other applications. For example, benchmarks executed while the anti-virus program is running or the operating system is being updated, result in incorrect performance results. Additionally, benchmark results strongly depend on the hardware platform, such as CPU and main memory, but also the used network and all intermediate network components (e.g., switches). For example, the benchmark might be adversely affected if communication happens over a congested network, over a wireless network, or if the CPU is doing automatic frequency scaling for power management. Such benchmarks therefore have limited predictive power: results are only valid for the machine on which they are run, and in those exact circumstances.

The second problem is related to the debugability of the system. As the program is to be executed on a shared-resource machine, its interleavings with other processes is non-deterministic. The use of a network renders it even less reproducible: network delays are non-deterministic by nature. While developing such systems, bugs can be hard to replicate, as they are often related to these interleavings. For example, bugs causing deadlocks might only occur in a very select number of situations, and cannot be reproduced with absolute certainty. Such bugs are often termed “Heisenbugs” (Gray 1985), and they represent the majority of bugs in distributed applications. As debugging is an invasive operation, due to instrumentation, behaviour might even change during debugging.

Both problems are caused by the non-determinism of the underlying platform, and is therefore a timing issue. Performance evaluations are affected by the hardware used, as operations take different times to execute. Results are therefore not reproducible on other machines, or even on the same machine, nor is there much predictive power. Debugging, and specifically instrumentation, affects timing as operations suddenly take longer to execute, or more operations have to be executed. This results in potentially different behaviour, making debugging harder. To address both problems, we do away with the current notion of time, and propose a time over which we have full control: simulated time (Goldstein and Khan 2017).

To use simulated time, we must make use of a simulation, instead of executing the actual program. Therefore, we create a Parallel DEVS (Zeigler et al. 2000, Chow and Zeigler 1994) model of the tool in question, effectively splitting the notions of wall clock time and simulated time.

The remainder of this paper is organized as follows. Section 2 presents necessary background on the architecture of the Modelverse, which is referred to in our DEVS model. Section 3 presents the different components of our DEVS model, and gives information on calibration. Section 4 presents the performance results, which are used to achieve reproducible benchmarks and what-if analysis. Section 5 presents the debugging results, which allows for deterministic debugging, and advanced debugging operations. Section 6 presents related work. Section 7 concludes the paper.

2 MODELVERSE ARCHITECTURE

Before explaining the DEVS models, it is necessary to elaborate on the architecture of the tool in question: the Modelverse. This architecture is mimicked in the DEVS models explained later. The Modelverse is built up out of three core components: the Modelverse Interface (MvI), the Modelverse Kernel (MvK), and the Modelverse State (MvS). In short, the MvI is the client-facing component (e.g., GUI), which interacts with the MvK through a set of high-level operations (e.g., model-create), forming an MPM API. The MvK is the computational part of the Modelverse, which expands these high-level operations into low-level graph operations (e.g., create node) and forwards them to the MvS. The MvS, finally, is a graph database, storing all data, such as models and the execution state, which offers a graph manipulation API. In our design, each of these components can interact over a network, and there is thus no strong coupling between any of them.

Modelverse Interface The Modelverse Interface (MvI) is a generic name for any tool that sends high-level requests to the Modelverse. Communication is done using XML/HTTPRequest, and is therefore relatively simple to implement, even in existing tools. In our case, we even have multiple implementations, depending on the requirements of the user: a textual interface, a graphical interface, and a batch processing interface for Python. For the remainder of this paper, we assume that the MvI merely has to send out high-level operations (e.g., create model, delete model, add class, set attribute).

Modelverse Kernel The Modelverse Kernel (MvK) takes in high-level operations, and has to translate them to low-level graph operations, thereby being responsible for all computation. This is non-trivial, as the MvK is stateful (e.g., supported high-level operations depend on the sequence of operations) and some operations are complex to expand. For example, more complex operations are "execute model transformation", which results in thousands of low-level operations. The code governing the MvK behaviour is written in a dedicated action language, which has an explicitly modelled semantics (Van Tendeloo 2015). It is important to note that the MvK can receive requests from multiple MvIs simultaneously, where some MvIs might belong to the same user. As such, some kind of task scheduling has to be defined, which sequentializes the requests to the MvS.

Modelverse State The Modelverse State (MvS) is a graph database, which processes low-level operations on a graph, thereby being responsible for all data storage. It is possible to receive operations from multiple MvK instances simultaneously, in which case they are processed in order of arrival. The MvS, however, does not care about the origin of these requests, or dependencies between requests: all user and task management is handled in the MvK, and the MvS merely reasons about graphs.

3 MODELLING

Now we model the previously presented components using DEVS, in the PythonPDEVS modelling and simulation tool (Van Tendeloo and Vangheluwe 2014). While other tools are possible as well, PythonPDEVS is a decent trade-off between performance and functionality (Van Tendeloo and Vangheluwe 2017a). We model the three main components of the Modelverse, as presented before: the Interface (MvI), Kernel (MvK), and State (MvS). For each of these components, we reuse as much of the original code as possible. As such, most execution logic is merely imported and reused in the DEVS model, though some minimal DEVS wrapper code is necessary to make this fit in the DEVS formalism. By reusing existing logic, we are sure that the same behaviour is visible. The network inbetween all of these components must also be modelled: this is part of the environment that we have to model explicitly.

All of the sources are available online at https://msdl.uantwerpen.be/git/yentl/modelverse.

3.1 Modelverse Interface

The MvI was responsible for the creation of the high-level operations. While normally done by the user interactively, for example by clicking a button, we assume that this is provided as input to the simulation. The set of high-level operations to execute, is that of a power window modelling and safety verification (Van Tendeloo and Vangheluwe 2017b), as often used in the context of MPM (Mustafiz et al. 2012, Mosterman and Vangheluwe 2004).

Model The DEVS model of the MvI is, due to the previously mentioned restrictions, rather simple. It contains a list of operations to execute, and a set of operations to execute for specific models (e.g., details on how to model a control model). Whenever an operation is sent, we wait for a reply before sending the next request. While usually the reply is presented to the user, it is now thrown away, as we operate in batch mode. Despite this simple explanation, the DEVS model is quite convoluted, as we have to implement an autonomous MvI from scratch. For each response that the MvI receives, it checks whether it has to stop the simulation. This is done by setting a state variable, which is used by the termination condition.

Calibration As there are not many parameters to the MvI, not a lot of calibration was required. The list of operations to be executed, however, needs to be defined. As previously mentioned, we used the power window case study, for which we needed the list of high-level operations. To obtain the list of actual requests used, a real run of the Modelverse was instrumented with logging, thereby logging the various high-level operations sent to the Modelverse. This resulted in a set of around 16,000 requests, grouped in 75 "request blocks", as many requests don't depend on the result of the previous ones. A single group is sent simultaneously, thereby reducing the round trip time, as it is also done in the real execution of the Modelverse.

3.2 Modelverse Kernel

The Modelverse Kernel was responsible for the computations: converting the high-level model-management operations to low-level graph operations. As we reuse much of the logic from the actual Modelverse code base, this model is mostly a wrapper around that logic. Nonetheless, task management had to be reimplemented to suit the DEVS formalism.

Model The DEVS formalism requires to have full control over timing behaviour. This did not suit well with the original specification of the task management of the MvK, which was modelled with SCCD (Van Mierlo et al. 2016), instead of coded. While this is ideal for execution, generated SCCD code contains timing information and threading, making it unusable in a DEVS model. All task management was reimplemented in DEVS, with behaviour similar to the original implementation. Most of the other code, specifically the translation code, was reused.

Calibration The MvK needs some calibration, as the computations take time. This was not needed in the MvI, as we abstracted away the time needed by the MvI to come up with the next high-level operation. We cannot do this for the MvK, as there are many steps involved in translating the high-level operations to low-level operations. We have implemented two types of interpreter: one based on pure interpretation, and one based on a Just-In-Time (JIT) compiler. The pure interpreter will take less time to do the conversion to low-level operations, but will generate more low-level operations. On the other hand, the JIT will take much more time for the conversion, but generates significantly less operations. Results were measured by doing a first DEVS simulation run, where the computation was actually performed, and the time measured. This gives a huge set of samples: $34,638,621$ to be precise. For the JIT, the distribution of these values is shown in Figure 1. It is clear that most operations take a relatively short time (those that were already compiled), and some operations take significantly more time (those where compilation happens). For the pure interpreter, all invocations are nearly instantaneous.

3.3 Modelverse State

The Modelverse State was responsible for state manipulations. Again, we reuse almost all logic from the graph database that underlies our implementation, and therefore the DEVS model for this is merely a wrapper which monitors the time the operation takes.

Model The DEVS specification of the MvS is trivial: when receiving a list of low-level operations, the operations are executed in order, and the simulation time is incremented with the time it takes to execute the operations, which is sampled from a distribution. Results are subsequently sent back to the MvK.

Calibration Calibration is again required to determine how long the low-level operations take. For this, the same approach is used as with the MvK: first do a "calibration run", in which timing information for each operation is recorded. Table 1 presents how many samples were made for each of the low-level graph

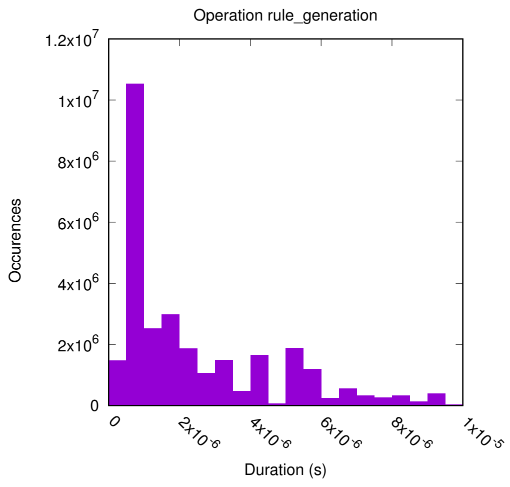

Figure 1: Distribution of time taken for rule generation in the MvK.

operation	samples	averages
read root ID	1	0.00000286s
read dictionary	13,456,485	0.00000111s
read dictionary keys	181,314	0.00000804s
read node value	14,538,084	0.00000034s
read dictionary by node	491,026	0.00000179s
read dictionary edge	503,147	0.00000289s
create node	775,370	0.00000068s
create value	2,157,118	0.00000092s
create dictionary	1,329,637	0.00000452s
delete edge	1,489,940	0.00000288s
delete node	31,092	0.00000854s
dictionary key lookup	9,093	0.00001803s
create edge	5,263,314	0.00000184s
read outgoing edges	4,627,855	0.00000150s
read incoming edges	1,325,308	0.00000178s
read edge	3,913,127	0.00000039s
garbage collect	21	0.88014233s

Table 1: MvS operations and the measured calibration results.

operations, and gives some information on the values found. Some operations are not executed that frequently, as the JIT was used, thereby lowering the number of low-level operations. Even though we present the averages, the actual model can make use of an inferred distribution, or can sample from the measured timings directly.

3.4 Network

Apart from the Modelverse components, our DEVS model also requires an explicit model of the network. While normally the network is used as-is, for example through the use of sockets, we now explicitly model this, giving us full control. This DEVS model only has to model latency and bandwidth. Other network characteristics, such as routing and packet loss, can be ignored, as the original application also relied on TCP/IP to handle these aspects. As we are not interested in any of these characteristics in particular, we can abstract these away in this model. Note that packet loss is still included by taking it into account in the latency distribution: while packet loss is handled in TCP/IP, the induced timeout and resend still increases the latency.

Model The DEVS model of our network is relatively simple, and only models latency and bandwidth. For each incoming message, we delay the sending by a time that is equal to the latency (a fixed cost), and add the time due to the restricted bandwidth. As such, the message is serialized using JSON, which results in a string that has to be transferred over the network. This string has a number of characters, and thus a number of bytes, as we use ASCII encoding. Using this information, and the maximum bandwidth, we can compute the time it takes for the message to pass over the network.

Calibration For the network, calibration is again required, as we need to find values for the latency and the allowed bandwidth. Tools like *ping* can be used to estimate the round trip latency, and tools like *iperf* can estimate the network bandwidth. When two components are ran on the same machine, latency and bandwidth can be abstracted to zero and positive infinity, respectively. Packet loss is implicitly included in the round trip latency measurements.

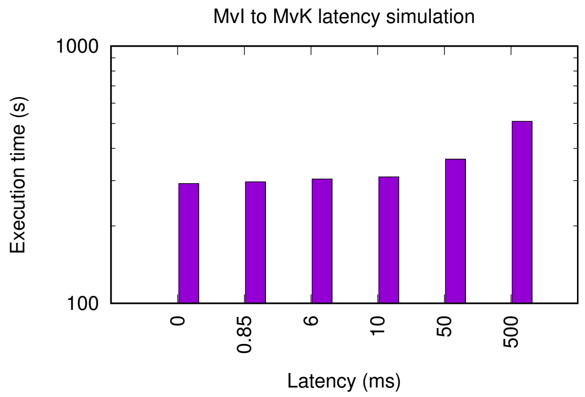

Figure 2: Influence of MvI - MvK latency.

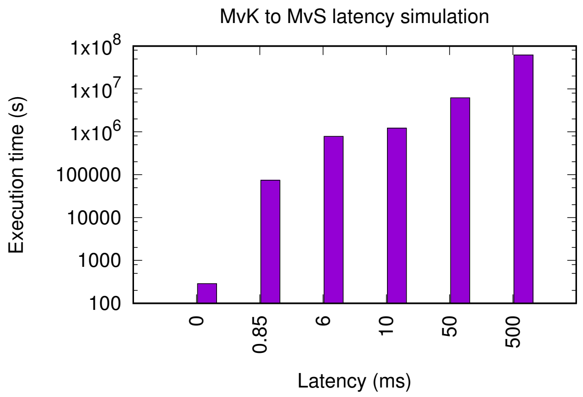

Figure 3: Influence of MvK - MvS latency.

4 PERFORMANCE

We now apply this model for our first purpose: performance evaluation. We consider four aspects of performance analysis that are simplified by modelling and simulation using the Parallel DEVS formalism. These aspects are what-if analysis, benchmark automation, determinism, and benchmark performance.

4.1 What-if Analysis

First is the possibility for what-if analysis. While with usual benchmarks, we need to have access to the benchmarking hardware to generate meaningful results, a what-if analysis allows us to use a hypothetical setup. This makes it possible to prototype specific hardware, such as a low-latency network, without actually having to invest in it beforehand. Indeed, it might not be worthwhile the investment if the benchmark results prove insatisfactory. Additionally, we can explore a set of possibilities, optimizing the total cost.

With a DEVS model, what-if analysis becomes possible. For example, we can easily change the network latency parameter from $0ms$ (e.g., local execution) to $500ms$ (e.g., satellite connection), and see the effect. Figure 2 and Figure 3 present results for varying network latencies between the MvI and MvK, and the MvK and MvS on the total execution time, respectively. These results indicate that we can increase the latency between the MvI and MvK without too many problems, which is one of the requirements of the Modelverse: it must be able to run distributed, with MvIs running on various machines, possibly even over the internet (i.e., relatively high latencies). Although the execution time does increase, it less than doubles when going from an instantaneous connection ($0ms$ latency) to a high latency connection ($> 100ms$ latency). The MvK and MvS, however, should ideally be ran with very low latencies, as the results indicated: execution time is highly dependent on this latency. While not ideal, this does not form a significant problem: the MvK and MvS are two server-side components, and are likely hosted close to one another and using a low-latency connection.

These results are as expected, though they still provide added value: we can quantify how long it will take, up to some degree of certainty. For example, we can now state that the performance is tolerable, even when using a high-latency connection between MvI and MvK. For performance critical software, hard restrictions are often imposed on the acceptable delays, rendering quantification important. Similarly, for the MvK and MvS we intuitively know that splitting them is not a good idea performance-wise, as they communicate a lot. Nonetheless, through simulation we can quantify the expected execution time, allowing for a trade-off. Constraints and costs can be combined with the execution time to find a cost-optimal solution.

Notwithstanding these advantages of the DEVS model, it remains an abstracted simulation. As such, results will not be perfectly accurate, though rather to be within some bounds around the actual values. The more fine grained the model becomes, the less abstractions are made, and the more accurate the results become. However, less abstractions will also increase the simulation time. For our purposes, the current level of abstraction proved sufficient.

4.2 Automation

A second advantage of using simulation is that everything can be fully automated. Going back to our previous example of network latency, it is immediately obvious that we cannot automatically switch the system over to various types of network, even if we were to have all the required hardware. More complex hardware changes, such as disconnecting a network cable, or swapping the CPU, are even more difficult to do automatically. As such, these operations are done manually, introducing manual intervention and thus non-determinism: the point in time when the cable is manually disconnected, will vary between two simulation runs, even if only by a few milliseconds.

With a DEVS model, all these operations become trivial to execute, as all relevant aspects of the system, such as the network, are modelled explicitly. Disconnecting a network cable is then just sending an event to the network model, which subsequently no longer transmits messages. All results presented in this paper were obtained completely automatically, and without the use of any platform-specific operations. Additionally, by disconnecting from the platform, it becomes possible to run multiple simulations simultaneously, as all platform changes are simulated as well.

4.3 Determinism

A third advantage of using simulation is determinism. Normal benchmark execution, on actual hardware, has many causes of non-determinism: a shared-resource machine, a shared network, varying network characteristics, and so on. While it is possible to replicate benchmark results, they are only identical to some level, and it might take several tries. If execution takes a significant amount of time, we often do not want to execute the same program multiple times. Additionally, it makes it difficult for others to replicate results.

With a DEVS model, determinism is achieved by modelling all sources of non-determinism, such as the network and computation times, explicitly. Changes to the setup can be evaluated with a single simulation run, which then automatically reuses *exactly* the same simulation setup, as all data is deterministic. Not only will this result in exactly the same behaviour, but it will also immediately show the impact of changes on performance, without non-deterministic jitter on the results. Of course, the performance effect *in general* still requires multiple simulation runs with varying seeds. Figure 4 presents the result of several actual execution runs of the program. While there is less than 10% difference between all executions, the difference is noticeable. With simulation, results are always equal, given the same random seed.

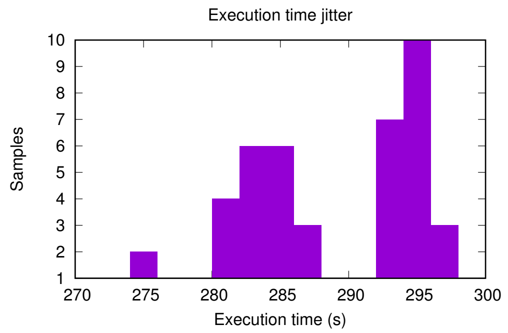

Figure 4: Actual execution results.

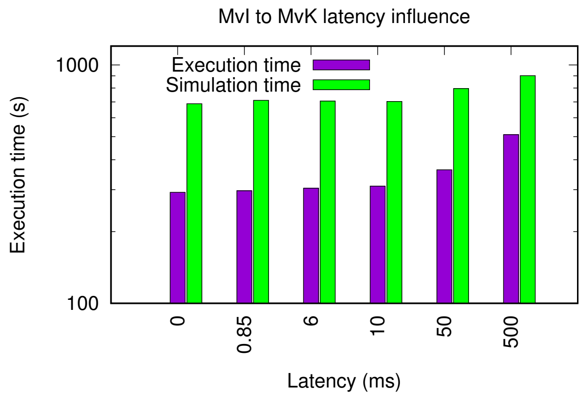

Figure 5: Varying MvI - MvK latency.

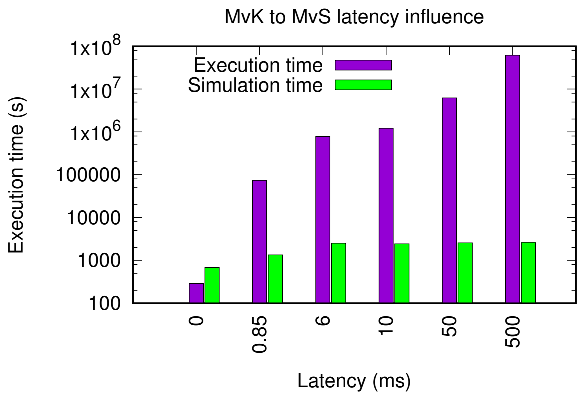

Figure 6: Varying MvK - MvS latency.

4.4 Performance

A fourth advantage of using simulation is that simulation is often faster than executing the actual application. During execution of the program, only a part of it is spent on actual computation, as much overhead exists: task switching, network timeouts, network transfer delays, and so on. During simulation, mostly the same code is still executed as in the original application, though we can skip many of these sources of overhead. For example, network latency only affects simulated time, as the network is not actually used, but only simulated. As such, high network latencies (e.g., 500*ms*) will not take much longer to simulate than no network latency at all (i.e., 0*ms*). Similarly, simulations can happen on fast computers, while using the timings of a slower computer. This is advantageous, as the wall clock time then progresses slower than the simulated time, meaning that simulation becomes faster than actual execution. Notwithstanding these speedups, simulation also introduces its own types of overhead. While this is not to be ignored, the overhead induced by the network is generally larger than the simulator overhead.

Figure 5 presents the difference between the time needed for simulation, and the simulated execution time for a range of different latencies between the MvI and MvK. In this case, simulation is always slower than actual execution, as this network connection is not frequently used. This is comparable to the results we had before, where we saw that the influence of this latency on execution time is minimal. We do, however, notice that the simulation time also increases, which was against expectations. It seemed that the JIT compiler, which is a core component of the MvK, reacts differently depending on how much time it takes to execute some functions.

Figure 6, on the other hand, shows the results for varying the MvK to MvS latency. Here, the simulation time also increases, though far less than when executing the actual system. In this case, simulation clearly outperforms actual execution: for 500*ms* latency, simulation is more than $24,000$ times faster.

5 DEBUGGING

Apart from performance analysis, DEVS modelling and simulation also helps debugging the application. While the original application can be debugged using commonplace code debuggers, we claim that this is not ideal for distributed and performance-critical applications, such as the ones considered in this paper. Code debuggers often rely on instrumentation of the source code or binary, thereby altering the code that is executed. While instrumentation does not modify the semantics of the code directly, it might indirectly influence the semantics by increasing the execution time of some code fragments, thereby influencing various interleavings. Attempts have been made to minimize the impact of instrumentation, though advanced features, such as omniscient debugging, always have a noticable impact. We consider two shortcomings of code debuggers in our context: lack of deterministic debugging and advanced debugging features. In the remainder of this section, we elaborate on each problem, and mention how DEVS modelling and simulation addresses this.

5.1 Deterministic Debugging

The first problem is the lack of deterministic debugging. When debugging, the code is almost always instrumented in one way or the other. The simplest way of debugging, adding print statements, obviously changes the source code. Even worse, it also changes the timing behaviour: the print statement has to be executed, causing all operations after it to start later, possibly changing interleavings. All approaches monitor the application in one way or the other, resulting in ever so slight changes in execution behaviour. This makes it difficult to replicate bugs while using a debugger, or even on a second try. As a result, the same execution might have to be replicated many times, just to make sure that the bug manifests itself. But even when the bug can be reproduced frequently, the patches that are applied to further track down the bug (e.g., more print statements), or even to fix the bug (e.g., change some algorithm), can merely mask away the bug, instead of actually solving it. When non-determinism is involved in one way or the other, we often cannot be certain whether a bug just no longer manifests itself, or was completely fixed.

With the DEVS model, we have previously mentioned that we achieved determinism for performance benchmarks. But this determinism is also present during debugging: when nothing is altered in the simulated time, any change to the algorithms has no effect whatsoever on the simulation results. Therefore, print statements can be liberally added, and we can be sure that the bug is reproduced each and every time.

5.2 Advanced Debugging Features

The second problem is related to debugging features. While code debuggers have lots of features, some intrusive features, such as omniscient (reversible) debugging, are only selectively enabled, as they have a huge performance impact (e.g., up to $50,000$x for omniscient debugging in GDB according to users[1]). Enabling such features aggravates non-determinism and makes it even more difficult to replicate the bug. Nonetheless, even when running the code debugger on our DEVS simulation, the huge performance penalty makes it difficult to use, and it then works at the wrong level of abstraction.

With the DEVS model, however, the level of abstraction can be raised by using a DEVS debugger. While they both debug the same application, they do so at a different level of abstraction: code debuggers debug each line of code, and for omniscient debugging, this means that each line of code can be "reverted". DEVS debuggers, however, debug transitions of the DEVS model, which are composed of hundreds or thousands of lines of code. Naturally, the performance impact is decreased significantly. Additionally, advanced debugging features that are already implemented for the specified DEVS simulator can be used as-is, such as all the features implemented for PythonPDEVS (Van Mierlo et al. 2017). This opens up many new dimensions to the debugging of our application, while limiting the performance impact.

6 RELATED WORK

We considered two aspects in this paper: performance evaluation and debugging. As such, related work is also split up in these two dimensions.

[1] https://softwareengineering.stackexchange.com/questions/181527/why-is-reverse-debugging-rarely-used

For performance evaluation, the traditional approach is the creation of benchmarks, which execute the application and measure how long it takes. While this yields trustworthy results, there were many potential problems, as mentioned in this paper. DEVS modelling has been used before to measure the performance of software applications, for which we now give some examples. One DEVS model was for a distributed DEVS simulator (Syriani et al. 2011), where the DEVS abstract simulator was modelled using DEVS itself. This was primarily done to monitor the behaviour of the simulator in exceptional cases, such as when a disconnect happens. Not many details were given on the calibration of the model, and the DEVS simulator itself was not performance-critical in this paper: it was a minimal distributed DEVS simulator without advanced synchronization protocol. As such, no attention was payed to the performance of the application. Another DEVS model is that for a new algorithm for property-based locking in collaborative modelling (Debreceni et al. 2017). Here, DEVS was also used for the context of (distributed) execution, though the focus was not on execution performance, but on deterministic simulation as to whether a lock was granted or rejected.

For distributed debugging, the traditional approaches have troubles coordinating different systems and platforms. Some code debuggers do exist, though they have a limited set of functionality in a distributed setting. Even for non-distributed programs, code debuggers suffer from the difficulty of reproducing a bug, often called intermittent bugs, or “Heisenbugs” (Gray 1985). Heisenbugs are bugs which “go away when you look at them”, and could elude programmers for years of execution (Gray 1985). In non-distributed programs, Heisenbugs are much less frequent, as they are often caused by hardware faults. Much work has been spent on decreasing the overhead of instrumentation, such as using dedicated hardware (Stollon 2011) or dynamic instrumentation (Zhao et al. 2008). Nonetheless, an overhead still exists, thereby altering the behaviour between runs. Related to our work, full system simulation (Albertsson and Magnusson 2000) can be used to completely simulate the hardware on which we are executing. Compared to our approach, full system simulation is more general, as it allows all types of applications to be executed. Nonetheless, due to the very low level of abstraction, that of the CPU instructions, debugging is made difficult again, as we are debugging using low-level concepts, instead of programming language or modelling concepts. As such, this approach only seems useful when debugging parts of the operating system. Additionally, performance is reduced even further, as execution is estimated to take about 50-200 times as long (Albertsson and Magnusson 2000). No mention is made about using this approach for distributed applications. For the previously mentioned DEVS models, no mention was made about debugging using the DEVS model. A distributed version of PythonPDEVS (Van Tendeloo and Vangheluwe 2015) was modelled using DEVS, and was used for debugging (Van Tendeloo 2014). While no extensive DEVS debugger existed back then, the possibility to alter the code (e.g., add print statements) and reproduce the bug, were immensely helpful in debugging problems with the distributed synchronization algorithms.

7 CONCLUSIONS

Today’s software has to answer to many requirements, such as distributed execution, support for multiple simultaneous users, and have acceptable performance as well. In particular, our tool for Multi-Paradigm Modelling (MPM), called the Modelverse, has to fulfill such requirements due to the nature of MPM. During its development, problems arise due to the distributed and concurrent nature of the application, which cannot be easily debugged with today’s code debuggers. After the initial development effort, performance optimization is made difficult due to the tight relation with hardware and the non-deterministic nature of the communication medium.

In this paper, the Modelverse was modelled in the Parallel DEVS formalism, thereby giving us full control over time: wall clock time and simulated time are effectively split. This split gives us the ability to debug the application without interfering with the delicate timing mechanics underlying the system, and due to the higher level of abstraction, we can reuse advanced debugging operations, such as omniscient debugging, without excessive overhead. Apart from debugging, performance analysis and optimization become easier and less ad-hoc than usual, as we can do what-if analysis, and deterministically get benchmarking results. For each of these aspects, we have described how our DEVS model addresses these problems, resulting in the aforementioned advantages.

While these results were obtained in the context of our prototype MPM tool, we believe that this approach is applicable to many other complex, distributed applications. Advantages of using DEVS modelling and simulation for tool development are relatively easily achievable, given that much code can be reused, as was the case for our tool. In future work, we intend to further optimize the application with the use of our DEVS model.

REFERENCES

Albertsson, L., and P. S. Magnusson. 2000. “Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads”. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 191–198.

Chow, A. C. H., and B. P. Zeigler. 1994. “Parallel DEVS: a parallel, hierarchical, modular, modeling formalism”. In *Proceedings of the 1994 Winter Simulation Conference*, edited by J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, pp. 716–722, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Debreceni, C., G. Bergmann, I. Ráth, and D. Varró. 2017. “Property-Based Locking in Collaborative Modeling”. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pp. 199–209.

Goldstein, R., and A. Khan. 2017. “A Taxonomy of Event Time Representations”. In *Proceedings of the Spring Simulation Conference*, pp. 6:1–6:12.

Gray, J. 1985. “Why Do Computers Stop and What Can Be Done About It?”. Technical report, Tandem Computers.

Mosterman, P. J., and H. Vangheluwe. 2004. “Computer Automated Multi-Paradigm Modeling: An Introduction”. *SIMULATION* vol. 80 (9), pp. 433–450.

Mustafiz, S., J. Denil, L. Lúcio, and H. Vangheluwe. 2012. “The FTG+PM Framework for Multi-Paradigm Modelling: an Automotive Case Study”. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling*, pp. 13–18.

Stollon, N. 2011. *On-Chip Instrumentation*. 1st ed. Springer.

Syriani, E., H. Vangheluwe, and A. Al Mallah. 2011. “Modelling and simulation-based design of a distributed DEVS simulator”. In *Proceedings of the Winter Simulation Conference*, pp. 3007–3021.

Van Mierlo, S., Y. Van Tendeloo, B. Meyers, J. Exelmans, and H. Vangheluwe. 2016. “SCCD: SCXML Extended with Class Diagrams”. In *Proceedings of the Workshop on Engineering Interactive Systems with SCXML*, pp. 2:1–2:6.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2017. “Debugging Parallel DEVS”. *SIMULATION* vol. 93 (4), pp. 285–306.

Van Tendeloo, Y. 2014. “Activity-aware DEVS simulation”. Master’s thesis, University of Antwerp, Antwerp, Belgium.

Van Tendeloo, Y. 2015. “Foundations of a Multi-Paradigm Modelling Tool”. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, pp. 52 – 57.

Van Tendeloo, Y., and H. Vangheluwe. 2014. "The Modular Architecture of the Python(P)DEVS Simulation Kernel". In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, pp. 387–392.

Van Tendeloo, Y., and H. Vangheluwe. 2015. "PythonPDEVS: a distributed Parallel DEVS simulator". In *Proceedings of the 2015 Spring Simulation Multiconference*, SpringSim '15, pp. 844–851, Society for Computer Simulation International.

Van Tendeloo, Y., and H. Vangheluwe. 2017a. "An evaluation of DEVS simulation tools". *SIMULATION* vol. 93 (2), pp. 103–121.

Van Tendeloo, Y., and H. Vangheluwe. 2017b, December. "The Modelverse: a Tool for Multi-Paradigm Modelling and Simulation". In *Proceedings of the 2017 Winter Simulation Conference*, WSC 2017, pp. 944 – 955, IEEE.

Vangheluwe, H., J. de Lara, and P. J. Mosterman. 2002. "An Introduction to Multi-Paradigm Modelling and Simulation". In *Proceedings of the AIS'2002 Conference (AI, Simulation and Planning in High Autonomy Systems)*, pp. 9 – 20.

Zeigler, B. P., H. Praehofer, and T. G. Kim. 2000. *Theory of Modeling and Simulation*. 2nd ed. Academic Press.

Zhao, Q., R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. 2008. "How to Do a Million Watchpoints: Efficient Debugging Using Dynamic Instrumentation". *Lecture Notes in Computer Science* vol. 4959.

ACKNOWLEDGEMENTS

This work was partly funded by a PhD fellowship from the Research Foundation - Flanders (FWO). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

AUTHOR BIOGRAPHIES

YENTL VAN TENDELOO is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). In his Master's thesis, he worked on MDSL's PythonPDEVS simulator, a simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS, grafted on the Python programming language. The topic of his PhD is the conceptualization, development, and distributed realization of a new (meta-)modelling framework and model management system called the Modelverse.

HANS VANGHELUWE is a Professor in the department of Mathematics and Computer Science at the University of Antwerp (Belgium) and an Adjunct Professor in the School of Computer Science at McGill University (Canada). He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributor to the DEVS community of fundamental and technical research results.