Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS

C. Thule <casper.thule@eng.au.dk>
P. G. Larsen <pgl@eng.au.dk>
Aarhus University, Department of Engineering,
Inge Lehmanns Gade 10, 8000 Aarhus C, Denmark

Abstract. The development of Cyber-Physical Systems often involves cyber elements controlling physical entities, and this interaction is challenging because of the multi-disciplinary nature of such systems. It can be useful to create models of the constituent components and simulate these in what is called a co-simulation, as it can help to identify undesired behaviour. The Functional Mock-up Interface describes a tool-independent standard for constituent components participating in such a co-simulation and can support different formalisms. This paper describes an exploration of whether different concurrency features in Scala (actors, parallel collections, and futures) increase the performance of an existing application called the Co-Simulation Orchestration Engine performing co-simulations. The investigation was conducted by refactoring the existing application to make it suitable for implementing functionality that takes advantage of the concurrency features. In order to compare the different implementations testing was carried out using four test co-simulations. These test co-simulations were executed using the concurrent implementations and the original sequential implementation, verifying the simulation results, and retrieving the execution times of the simulations. The analysis showed that concurrency can be used to increase the performance in terms of execution time in some cases, but in order to achieve optimal performance, it is necessary to combine different strategies. Based on these results, future work tasks has been proposed.

Keywords: Co-simulation; concurrency; INTO-CPS; cyber-physical systems; FMI.

DOI: 10.15514/ISPRAS-2016-28(2)-9

For citation: Thule C., Larsen P.G. Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS. Trudy ISP RAN/Proc. ISP RAS], vol. 28, issue 2, 2016, pp. 139-156. DOI: 10.15514/ISPRAS-2016-28(2)-9

1. Introduction

Cyber-Physical Systems (CPSs) need to have close interaction between computer-based cyber parts controlling physical artefacts in a dependable way. In order to develop CPSs in a dependable manner it can be useful to create models of constituent

components that jointly form the system. A constituent model is an abstract description of a constituent, where the irrelevant details are abstracted away. Constituent models can be described in very different forms depending upon their nature, but here we will restrict ourselves to Discrete Event (DE) and Continuous-Time (CT) models representing very different disciplines. Such constituent models can then be used in a collaborative simulation (a co-simulation), which is able to couple models created in different formalisms. Thereby it is possible to simulate the entire system by simulating the components and exchange data as the common simulated time is progressing.

Typically such co-simulations are organised with a master-slave architecture where a Master Algorithm (MA) is used to manage the simulation. Fig 1 shows an example of four slaves, their dependencies, and input/output ports. It is the responsibility of the MA and thereby the master to orchestrate the simulation. This means to allow the different slaves to progress for determined time steps and resolve the dependencies between steps. A co-simulation often consists of three phases: Initialisation, simulation, and tear down. In the initialisation phase the master gets the properties of the slaves, chooses an MA, initialises the slaves, and establishes the communication channels. Next, in the simulation phase the master retrieves output values from the slaves, sets input values on the slaves, and invokes them to run a simulation step with a specific time step size. The slaves must respond with a status whether the step was accepted. In this phase, it can be necessary to perform a rollback[1] (if possible) for the relevant slave and run the simulation again with a different step size. Lastly, the outputs from the slaves are retrieved and the process repeats until a configured end time is reached. The final phase is tear down, where the slaves are shut down, memory is released, results are reported, and so forth.

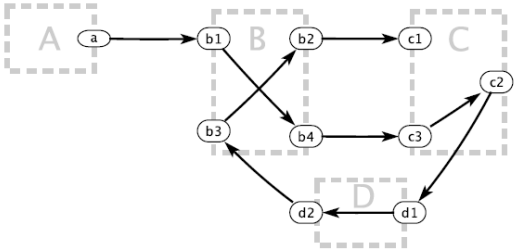

Fig. 1. Example of a simulated CPS with dependencies between slaves (the grey boxes) via their respective ports (the black ellipses) [1].

A challenge in using co-simulation as part of developing CPSs is that many complex multi-disciplinary systems cannot be modelled naturally in one simulation tool alone, but require several specialised simulation tools, that each do their part

[2]. This makes it necessary to develop solutions tailored for a specific purpose instead of generalised solutions, which is expensive. The Functional Mock-up

[1] A rollback can be necessary e.g. if a slave rejects a step size.

Interface (FMI) was created to solve these challenges, as it is a tool-independent standard for co-simulation [3]. The standard provides and describes C interfaces that can be partly or fully implemented by a component, which is then called a Functional Mock-up Unit (FMU). This makes it possible to create generalised solutions, as the components can contain their own solvers, and still adhere to FMI. The INTO-CPS project[2] [4] makes use of FMI for a simulation kernel of a tool suite ranging from original requirements expressed in SysML over heterogeneous constituent models that can be co-simulated and gradually moved down to their corresponding realisations. When developing CPSs using co-simulation, it is desirable to execute the simulations as fast as possible to enable the use of increasingly complex models and try a greater range of test scenarios. As many processors today have multiple cores [5] concurrency may increase the performance of an application, but it also introduces overhead. It is therefore of keen interest to determine, how concurrency can be used to potentially improve the performance. The performance in this context is considered to be how fast a co-simulation is performed, and is therefore measured in terms of time. This paper describes how the usage of concurrency was implemented in an existing application called the Co-Simulation Orchestration Engine (COE), which orchestrates co-simulations using FMI. Different implementations were performed in Scala using three different concurrency features: Akka Actors [6], futures [7], and parallel collections [8]. These were chosen because they offer different capabilities that can be taken advantage of in the COE, and therefore the trade-off between features and performance is interesting. One of the most important capabilities is composability, because FMUs can have different step sizes and rollbacks can be necessary, which can lead to complicated scenarios. Following is a short description of the concurrency features:

Parallel Collections: The motivation behind adding parallel collections to Scala was to provide a familiar and simple high-level abstraction to parallel programming [8]. Parallel collections are conceptually simple to use, as a regular collection can be converted to a parallel collection by invoking the function “par”. Once it is a parallel collection, functions such as map and filter are executed concurrently. Parallel collections are considered less composable than the other implementations, as the results are gathered in a blocking fashion.

Futures: A future is a placeholder for a value, that is the result of some concurrent calculation, and it can be accessed synchronously or asynchronously. The term “future” was originally proposed by Baker and Hewitt [9] in the context of garbage collection of processes. As opposed to parallel collections, it is possible to chain futures, such that when a future has been computed, the computed value is passed to the chained future.

[2] Public deliverables and more information regarding the INTO-CPS project can be retrieved from http://into-cps.au.dk.

Actors: The Actor Model was introduced as an architecture to efficiently run programs with a high degree of parallelism without the need for semaphores [10]. An actor is an autonomous object that encapsulates data, methods, a thread, a mailbox, and an address [11]. Actor methods can return futures, and therefore offer the same composability as futures in this regard. Actors also provide additional composable features, such as hierarchical structures, remote capabilities, message parsing, and so on.

The paper is structured as follows: Section 2 describes the initial implementation and the implementations using concurrency. Afterwards, Section 3 describes how the implementations were tested and presents the results. Then related work is treated in Section 4. Lastly, the work is summarised in Section 5 and future work is outlined in Section 6.

2. Co-Simulation Orchestration Engine Implementations

This section concerns the implementations of the COE application[3]. It focuses on the MA part of the implementations, as the initialisation and tear down phases are unaltered for the implementations described below.

The COE application runs as a web server using HTTP. The following HTTP requests are performed in the given order to run a simulation:

1. **Initialise:** A configuration file is sent to the web server. The configuration file contains the FMUs to be used in the simulation, the mapping between input and output values, and whether to use a fixed or variable step size.
2. **Simulate:** This request starts a simulation.
3. **Results:** This request returns the result and duration of a given simulation.

There are different implementations of the MA in the COE: A sequential implementation, and three implementations that execute concurrently, following the principles described above. These different implementations were developed in order to test and compare the performance of the COE in a sequential/concurrent setting and determine whether using concurrency could improve the performance.

2.1 Sequential Implementation

The sequential implementation of the MA consists of the following steps in the given order:

4. **Resolve inputs:** This step consists of mapping the outputs of the FMUs to the inputs of the other FMUs.
5. **Set inputs:** The input values determined in the previous step are passed to the FMU instances in this step.
6. **Serialize state:** In this step the states of the FMUs are serialized, so it is possible to perform a rollback in case of an error.

[3] See [12] for further details on the implementation.

7. **Get step size:** If variable step size is supported by the FMUs, then the maximum step size is retrieved in this step. Otherwise a configured fixed step size is used.
8. **Do step:** The FMU instances are invoked to perform a step with the step size determined in the previous step. This function contains the most extensive calculations performed by the FMUs.
9. **Process result:** The return values from the previous invocations are analysed and in case of any errors a rollback is performed or the simulation is terminated.
10. **Get state:** The state in terms of output values is retrieved in this step, and thereby the next iteration can begin.

In the sequential implementation a mapping operation is performed over the FMU instances in every step except the "Process result" step, where it depends on whether errors are encountered and if so which errors. This sums to six, possibly seven, mapping operations over the FMU instances.

2.2 Implementations with Concurrency

When implementing concurrency in the COE it is desirable that as much work as possible is performed in every concurrent invocation. To allow for a better usage of concurrency some functions should be grouped, such that a group of functions can be invoked concurrently. If concurrency was used in the sequential implementation to invoke the FMUs without refactoring the implementation, it would be necessary to invoke every step in different concurrent invocations. This would result in several thread initialisation and synchronizations per simulation step, where a synchronization is a waiting operation until all threads have finished computing. An example of this is shown in Fig. 2. The figure shows a possible usage of concurrency based on the sequential implementation with four FMUs (horizontal frames), where the functions "Set inputs", "Serialize state", "Do step", and "Get state" are invoked in different concurrent invocations. The realised implementation (vertical frame) invokes the functions using the same concurrent invocation for a given FMU. This will be described further below.

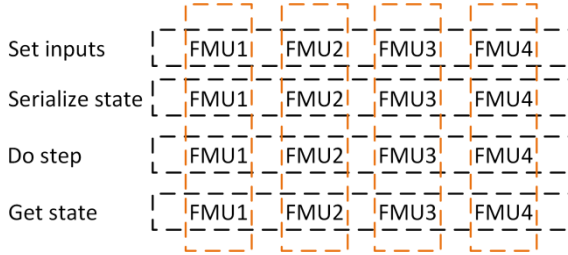

Fig. 2. The horizontal frames represent a possible usage of concurrency based on the sequential implementation. The vertical frames represent the usage of concurrency based on the implementations.

By refactoring and grouping these functions, it is possible to reduce the thread initialisations and synchronizations. This leads to more work performed by every spawned thread and fewer synchronizations, which minimizes the overhead of using concurrency. It is not possible to eliminate synchronization completely, because it is necessary to resolve the inputs for the FMUs before progressing, which requires retrieving the outputs from other FMUs, and therefore the simulation cannot continue until this has been performed. Besides minimizing the overhead of using concurrency, this grouping will also help to minimize the number of mapping operations performed in the steps in the sequential implementation, which is desirable to improve the performance.

The grouping and flow of a simulation step for the implementation using concurrency is shown in Fig. 3. The grouping was implemented in a separate and encapsulated function that exhibits referential transparency to prevent the necessity of locking mechanisms. This grouping will be referred to as the concurrent entity below.

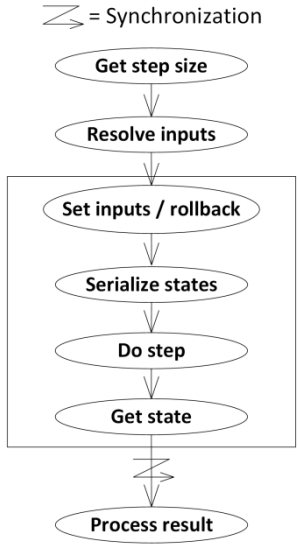

Fig. 3. Simulation step flow in the implementation using concurrency. The box represents the functions grouped together.

By creating these concurrent entities, it was a conceptually simple task to take advantage of the concurrency features. Furthermore, it effectively reduced the mapping operations from six, possibly seven depending on the step "Process result", to three. This implementation also makes it possible to include "assignment functions" such as "Set inputs" in the concurrent entity without lowering performance. Including "Set inputs" as its own concurrent invocation (as shown in the horizontal frames in Fig. 2) would lower the performance, because the overhead of using concurrency is too high compared to invoking the function sequentially.

Using the grouping (the vertical frame in Fig. 2) it improves performance to include "Set inputs", because it can be grouped with the other functions, e.g. "Do step", without additional overhead. However, the grouping also came with a trade-off: In the sequential implementation, the state would not be retrieved, if one or more FMUs fail in the step called "Do step", because it would be wasteful due to the error(s). But in the implementation using grouped functions, the state of the FMUs not failing in the step "Do step" would still be retrieved, because the entities responsible for the FMU simulation step are unaware of the state of other entities until the synchronization phase[4]. This can therefore lead to unnecessary retrieval of states.

In the sequential implementation, the flow is to calculate the parameters necessary for the next immediate function to be invoked on the FMUs, and then calculate the parameters again. In the implementations with concurrency this is changed to calculate the parameters necessary for an entire simulation step, and invoke the concurrent entity for each FMU concurrently. This makes it possible to maximize the workload for each concurrent invocation.

3. Testing

This section presents the evaluation of the COE described in Section II. The purpose is to gain data that can be used to compare performance of the sequential implementation and the implementations using concurrency. Furthermore, as concurrency can lead to non-determinism, it is important to verify the simulation results, which are the output values of the FMUs at different points in time relative to the step sizes. For this purpose, the sequential implementation was considered an oracle, and therefore simulation results of the concurrent implementations were compared against simulation results from the sequential implementation. In the longer term the plan is to use a representation of the FMI semantics as the ultimate oracle [13]. Here semantics is provided using the Communicating Sequential Processes [14] and this has been used to model check FMI for deadlock and livelock properties using the FDR tool [15].

The following test principles were followed during testing:

Test environment: A test consisting of multiple simulations should be performed on the same hardware with approximately the same processes running during the test. The reason for stating "approximately the same processes" is that the tests were run in a Windows environment, where it is not possible to completely control the running processes from the Operating System. All processes irrelevant to the execution of tests should be disabled during the tests.

Test functions: To limit inconsistencies in the processes running between simulations, each test should be implemented as a single test function. This means that a test performing simulations using the sequential implementation and the three

[4] Several programming languages offer the possibility to abort threads in a case like this. However, that increases the complexity and is not considered applicable in general.

concurrent implementations should be implemented in one test function to avoid undesirable interaction required to start other tests. To further ensure usable results the COE application should be restarted for every simulation.

Correct simulation results: The sequential implementation is considered to be an oracle and it is assumed that it calculates the "correct" simulation results. It should be verified that the concurrent implementations calculate the same simulation results as the sequential implementation.

Automation: The tests should be automated so they are easy to replicate and less prone to manual errors. This will also make them usable in the future development of the COE.

3.1 Test Setup

To enable automatic testing a framework was developed. This enabled testing of different concurrent implementations, evaluation of performance, and verification of consistency between the sequential simulation results and the concurrent simulation results. Implementation-wise this required support for launching the different implementations with different arguments, invoking the web servers using HTTP requests along with gathering, and verifying the consistency of results. To verify the consistency of results, the simulation results of the implementations using concurrency are automatically compared to the simulation results of the sequential implementation, as this is considered an oracle.

Different FMUs were used in the tests to investigate the performance, including a configurable FMU that was developed to control the level of computations, which will be described below. The tests and their corresponding FMUs are the following:

Heating, Ventilation, and Air Conditioning (HVAC) test: This test uses FMUs that perform the most extensive computations available in the project. The simulation consists of five FMUs: one controller FMU and four Fan Coil Unit FMUs. A test, which will be referred to as HVAC #1, was set up with an end time of 1000 seconds and a step size of 0.1 seconds. This is inspired by the case study undertaken by Unified Technologies Research Center inside the INTO-CPS project [16].

Sine Integrate Wait tests: These tests consist of three different FMUs, that perform limited computations, and therefore one has been modified. The FMUs are: a sine FMU generating a sine wave, an integrate FMU that integrates the sine values, and a modified integrate FMU. It is possible to configure the modified integrate FMU, such that it performs busy waiting in the "Do step" function for a given number of microseconds. It makes use of "QueryPerformanceCounter" recommended by Microsoft to use when high-resolution time stamps are required with microsecond precision [17]. The configuration of the busy wait does not have any impact on the performance of the FMU, because it happens in the initialisation phase, which is not part of the performance measurement. These FMUs were used to set up three tests, referred to as SI #1/2/3, where each simulation in the tests have an end time of 100 seconds and time step size of 0.1 seconds. The tests are the following:

SI #1 consists of one sine FMU, one modified integrate FMU, and three simulations: In the first simulation, the modified integrate FMU has a wait time of zero milliseconds, then 0.5 milliseconds, and lastly 1 millisecond.

SI #2 uses one sine FMU and five modified integrate FMUs with the same simulation setup as SI #1.

SI #3 uses one sine FMU and 100 integrate FMUs.

3.2 Test Results

This section contains the results of the tests described in Section 3.1. The results are presented in tables, where the unit of the numbers is milliseconds, and the table columns represent the following: Sequential refers to the sequential implementation, “Future” refers to the concurrent implementation using futures, “Par” refers to the concurrent implementation using parallel collections, and “Actor” refers to the concurrent implementation using actors. The result for the HVAC test is presented in Table. 1, and the results for the SI tests are presented in Table. 2, 3, and 4.

Based on these tests it is possible to draw some conclusions:

Executing simulations concurrently can be faster than executing them sequentially: The results for HVAC #1, SI #2, and SI #3 show that concurrent execution can be faster than sequential execution.

Executing simulations sequentially can be faster than executing them concurrently: The results for HVAC #1, SI #1, SI #2 and SI #3 show, that sequential execution can be faster than concurrent execution. Some of these test results contradict the previous conclusion, and therefore it is necessary to pay attention to the concurrency feature used.

Trade-off: An interesting discovery is that parallel collections perform worse than futures and actors. This indicates that even though parallel collections offer fewer capabilities than the other concurrency features, it does not perform faster.

Table. 1. Results from HVAC #1.

Sequential	Future	Par	Actor
31256	29822	31980	30919

Table. 2. Results from SI #1.

Wait	Sequential	Future	Par	Actor
0.0	195	330	656	374
0.5	4468	4635	5161	4715
1.0	8758	8938	9545	9032

Table. 3. Results from SI #2.

Wait	Sequential	Future	Par	Actor
0.0	355	434	834	622
0.5	21904	4679	5042	4746
1.0	43356	8970	9348	9184

Table. 4. Results from SI #3.

Sequential	Future	Par	Actor
355	434	834	622

4. Related Work

In order to make use of the improvements in hardware, it is necessary to improve the software. An adage known by "Wirth's law" goes: "Software is getting slower more rapidly than hardware becomes faster" [18][5]. He argues that methodologies are important in order to take full advantage of the improvements in hardware. Sutter urges application developers to take a hard look at the design of their applications and identify places that could benefit from concurrency [20]. This is necessary to exploit hardware capabilities, as processor manufacturers are turning to multicore processors. Harper et. al. conducted a study on a large-scale Publish/Subscribe bus system, and found an overall performance of 80 percent based on concurrency experiments [21]. Additionally, they surveyed concurrency design patterns with the purpose of helping developers towards the "right" patterns.

As mentioned previously, it is important to reduce communication and synchronization overhead between processes to achieve a fast simulation. Agrawal et. al. have implemented and evaluated three communications primitives for hardware/software co-simulation and found that a message-queue based communication backplane is preferable [22]. The other two primitives evaluated were shared memory and file-based sockets. Strategies that address the issue of synchronization are also introduced by Bishop et. al., and these strategies also deal with time management [23]. They conclude that using the design strategies discussed can enable the development of high-performance application-specific co-simulations. Kim et. al. consider synchronization between components simulators as the main reason for poor performance of HW/SW co-simulation [24]. They propose a novel technique based on virtual synchronization, which improves the simulator speed and minimizes the synchronization overhead. Becker et. al. describes an approach, where distributed communicating processes are used for the interaction between software and hardware using Unix interprocess communication mechanisms [25]. The

[5] Wirth attributes this to a different saying by Reiser [19].

approach does not accurately simulate the relative speeds of the hardware and software components, but the authors found this to be acceptable in their case.

The articles above consider synchronization, communication between simulators, and concurrency as a bottleneck in achieving fast co-simulations. It is therefore of keen interest to minimize the communication and synchronization along with taking advantage of concurrency. This work addresses these issues as well, as it is an attempt to limit synchronization and take advantage of concurrency. Furthermore, it is an attempt to avoid unnecessary inter-thread/inter-process.

5. Conclusion

Using FMI it is possible to develop a generalised application capable of performing co-simulation, thereby avoiding the need for tailored solutions developed to support the co-simulation of specific systems. It is desirable to perform a co-simulation as fast as possible, as it can help to verify the behaviour of systems or lead to the discovery of undesired behaviour. It was therefore investigated whether concurrency could be used to improve the performance of an application performing co-simulation. In some cases the usage of concurrency resulted in faster co-simulations, whereas in other cases sequential computation offered better performance. Because of this it is reasonable to conclude, that it is necessary to allow for different simulation strategies to achieve the fastest simulation. These strategies should support running simulations sequentially, concurrently, or a mix of these. For example, if an FMU that performs long-lasting computations is to be simulated with three FMUs that perform fast computations, then it could be optimal to run this simulation in a hierarchical structure using two threads as shown in Fig. 4.

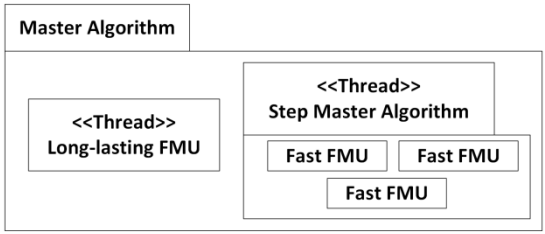

Fig. 4. Master Algorithm simulating four FMUs using an additional step Master Algorithm

Allowing for different strategies inevitably involves computing which strategies to use. A way of assisting the choice of strategy is to include a measure of how long-lasting the computations performed by an FMU are within the properties of the given FMU. However, this might be difficult to realise in a practical manner, where different hardware is used. An alternative approach is to use meta data for a given simulation. This can be configured beforehand, or the COE can determine it, when running the first co-simulation using the given FMUs.

6 Future Work

In order to improve the performance of the COE and choose when to use concurrency, there are several tasks to undertake:

Testability: Currently, the COE supports reporting the duration of an entire simulation without initialisation and reporting of results. As these steps inevitably are part of a simulation, they should be part of the performance tests. Additionally, the COE should offer better granularity for performance measurements. Better granularity will make it possible to examine the performance of different parts of the application, which can aid in finding bottlenecks and help target the development effort.

Investigate concurrency: Besides concluding that concurrency can/cannot improve the performance of the application in some cases, it is interesting to investigate when concurrency can improve the performance. Part of this investigation is to determine, whether an increase of performance is achievable by enabling sequential, concurrent, and mixed processing, as mentioned in the previous section. The approach is to implement nested COEs that appears as FMUs externally as shown in Fig. 5.

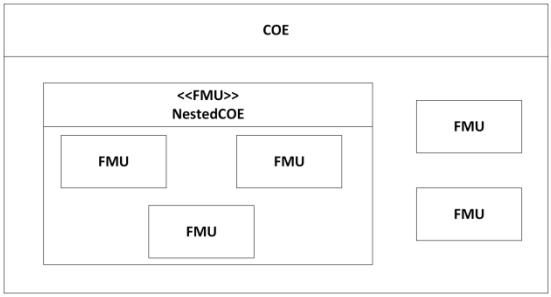

Fig. 5. Example of a nested COE participating in a co-simulation

This approach would allow for compositional co-simulation, as the nested COE will exhibit the same behaviour as an FMU externally and therefore be closed under composition. This allows for an elegant representation of complex systems, as it can be considered a co-simulation of co-simulations. Furthermore, it allows for hierarchical co-simulations, which can contribute to the reusability of co-simulations. The implementation will also support distributed scenarios, where nested COEs can be executed on different machines/operating systems and therefore allow for a greater range of co-simulation scenarios.

Guidelines: Since the future work concerns investigation of concurrency, it is compelling to attempt to generalise the lessons that will be learned and apply them on different case studies. The hope is that this can contribute to existing methodologies and guidelines on using concurrency efficiently.

Semantics alignment: The continuation of the FMI semantics work referred to above will also involve theorem proving using the Isabelle theorem prover [26] and we hope that it will be possible to align that with the COE work in order to use the semantics

directly as an oracle of checking conformance. This also involves examining the semantic properties of the concurrency features.

Graphical user interface: A Graphical User Interface (GUI) for the tools in the INTO-CPS project is currently being developed. This contains functionality to configure and interact with the COE. Furthermore, it will be possible to configure which simulation strategies to use and how the FMUs should be organised. The GUI application is being developed as a desktop application, as some of the tools in the tool chain do not support a distributed approach. However, by using Electron [27] it is possible to use web technologies for desktop applications. Therefore the GUI application is cross platform and supports a possible transition to being hosted on a web server in a distributed fashion.

Acknowledgement

The work presented here is partially supported by the INTO-CPS project funded by the European Commission's Horizon 2020 programme under grant agreement number 664047. Furthermore, the authors would like to thank Nick Battle for reviewing and providing input to this paper. Finally, thanks to Alexander Petrenko for translating parts of the paper into Russian.

References

[1]. D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *2013 Proceedings of the International Conference on Embedded Software*, Sept 2013, pp. 1–12. ISBN: 978-1-4799-1443-2.

[2]. J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *Proceedings of the 8th International Modelica Conference*, 2011, pp. 115-120. DOI: 10.3384/ecp11063115.

[3]. FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014.

[4]. J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains," in *FormaliSE: FME Workshop on Formal Methods in Software Engineering.* Florence, Italy: ICSE 2015, May 2015, pp. 40-46. DOI: 10.1109/FormaliSE.2015.14.

[5]. D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005. DOI: 10.1109/MC.2005.160.

[6]. Typesafe Inc, "Akka scala documentation," http://akka.io/docs/, Akka, September 2015, Release 2.4.0.

[7]. P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," http://docs.scala-lang.org/overviews/core/futures.html, (Visited on 05/03/2016).

[8]. A. Prokopec and H. Miller, "Parallel collections – overview - scala documentation," http://docs.scala-lang.org/overviews/parallel-collections/overview.html, 2015, (Visited on 05/03/2015).

[9]. H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming*

Languages. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/800228.806932. DOI: 10.1145/800228.806932.

[10]. C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://worrydream.com/refs/Hewitt-ActorModel.pdf.

[11]. G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online].Available: http://www.sciencedirect.com/science/article/pii/S1383762198000678.

[12]. C. Thule, "Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS," Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep. ECE-TR-26, May 2016. [Online]. Available: http://ojs.statsbiblioteket.dk/index.php/ece/issue/archive.

[13]. N. Amalio, A. Cavalcanti, C. König, and J. Woodcock, "Foundations for FMI Co-Modelling," INTO-CPS Deliverable, D2.1d, Tech. Rep., December 2015.

[14]. T. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.

[15]. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3 — A Modern Refinement Checker for CSP," *in Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 8413, 2014, pp. 187–201.

[16]. J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E. D. Mady, "Collaborative Model-based Systems Engineering for Cyber-Physical Systems - a Case Study in Building Automation". *INCOSE*. Edinburgh, Scotland. July 2016.

[17]. Microsoft, "Acquiring high-resolution time stamps (windows)," https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx, 2015, (Visited on 05/03/2016).

[18]. N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995.

[19]. M. Reiser, *The Oberon System: User Guide and Programmer's Manual*. New York, NY, USA: ACM, 1991.

[20]. H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16–23, 2005.

[21]. K. E. Harper, J. Zheng, and S. Mahate, "Experiences in initiating concurrency software research efforts," *in Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering – Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810316. DOI: 10.1145/1810295.1810316.

[22]. B. Agrawal, T. Sherwood, C. Shin, and S. Yoon, "Addressing the challenges of synchronization/communication and debugging support in hardware/software cosimulation," in *VLSI Design, 2008. VLSID 2008. 21st International Conference on VLSI Design (VLSID 2008)*, Jan 2008, pp. 354–361.

[23]. W. Bishop and W. Loucks, "A heterogeneous environment for hardware/ software cosimulation," *in Simulation Symposium, 1997. Proceedings., 30th Annual*, Apr 1997, pp. 14–22.

[24]. D. Kim, Y. Yi, and S. Ha, "Trace-driven hw/sw cosimulation using virtual synchronization technique," in *Design Automation Conference, 2005. Proceedings. 42nd*, June 2005, pp. 345–348.

[25]. D. Becker, R. K. Singh, and S. G. Tell, “An engineering environment for hardware/software co-simulation,” in In 29th *ACM/IEEE Design Automation Conference*, 1992, pp. 129–134.
[26]. T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.
[27]. Github, “Electron – Build cross platform desktop apps with JavaScript, HTML, and CSS”, version 1.2.2, http://electron.atom.io/, (Visited on 10/06/2016).

Исследование влияния использования параллелизма на производительность движка косимуляции в проекте INTO-CPS

К. Тул <casper.thule@eng.au.dk>
П. Г. Ларсен <pgl@eng.au.dk>
Орхусский университет, Технический факультет,
ул. Инге Лехманс, 10, 8000 Орхусс С, Дания

Аннотация. Кибер-физические системы часто включают в себя управляющие кибер-элементы, контролирующие физические объекты и взаимодействующие с ними. Анализ процессов в таких системах является сложной задачей из-за междисциплинарного характера этой области исследований. Моделирование и симуляция поведения составляющих систему компонентов, так называемая косимуляция, позволяет выявлять возможность нежелательного поведения. Интерфейс FMI (Functional Mock-up Interface) описывает стандартный интерфейс взаимодействия с составляющими компонентами, участвующими в такой косимуляции, и может поддерживать различные формализмы. Статья описывает исследование того, насколько различные возможности параллелизма в Scala (акторы, параллельные коллекции и футуры) увеличивают производительность существующего движка Co-Simulation Orchestration Engine, выполняющего косимуляцию. Исследование сопровождалось рефакторингом имеющегося кода с тем, чтобы реализация могла использовать преимущества параллельных возможностей. Для того, чтобы сравнить различные варианты реализации выполнялось по четыре тестовых косимуляции. В тестовых косимуляциях сравнивались параллельные реализации и исходная последовательная реализация, верифицировались результаты моделирования и получались оценки времени моделирования. Анализ показал, что в некоторых случаях параллелизм может использоваться для повышения производительности, но для того, чтобы достичь оптимальной производительности, необходимо комбинировать различные стратегии. На основе полученных результатов предлагаются будущие направления исследований.

Ключевые слова: Косимуляция; параллелизм; INTO-CPS; кибер-физические системы; FMI

DOI: 10.15514/ISPRAS-2016-28(2)-9

Для цитирования: Тул С., Ларсен П.Г. Исследование влияния использования параллелизма на производительность движка косимуляции в проекте INTO-CPS. Труды ИСП РАН, том 28, вып. 2, 2016 г., стр. 139-156. DOI: 10.15514/ISPRAS-2016-28(2)-9

Список литературы

[1]. D. Broman, C. Brooks, L. Greenberg, E. Lee, M. Masin, S. Tripakis, and M. Wetter, "Determinate composition of fmus for co-simulation," in *2013 Proceedings of the International Conference on Embedded Software*, Sept 2013, pp. 1–12. ISBN: 978-1-4799-1443-2.

[2]. J. Bastian, C. Clauss, S. Wolf, and P. Schneider, "Master for Co-Simulation Using FMI," in *Proceedings of the 8th International Modelica Conference*, 2011, pp. 115-120. DOI: 10.3384/ecp11063115.

[3]. FMI development group, "Functional mock-up interface for model exchange and co-simulation 2.0," Modelica, Tech. Rep. Version 2.0, July 2014.

[4]. J. Fitzgerald, C. Gamble, P. G. Larsen, K. Pierce, and J. Woodcock, "Cyber-Physical Systems design: Formal Foundations, Methods and Integrated Tool Chains," in *FormaliSE: FME Workshop on Formal Methods in Software Engineering*. Florence, Italy: ICSE 2015, May 2015, pp. 40-46. DOI: 10.1109/FormaliSE.2015.14.

[5]. D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005. DOI: 10.1109/MC.2005.160.

[6]. Typesafe Inc, "Akka scala documentation," http://akka.io/docs/, Akka, September 2015, Release 2.4.0.

[7]. P. Haller, A. Prokopec, H. Miller, V. Klang, R. Kuhn, and V. Jovanovic, "Futures and promises - scala documentation," http://docs.scala-lang.org/overviews/core/futures.html, (Visited on 05/03/2016).

[8]. A. Prokopec and H. Miller, "Parallel collections – overview - scala documentation," http://docs.scala-lang.org/overviews/parallel-collections/overview.html, 2015, (Visited on 05/03/2015).

[9]. H. C. Baker, Jr. and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: ACM, 1977, pp. 55–59. [Online]. Available: http://doi.acm.org/10.1145/800228.806932. DOI: 10.1145/800228.806932.

[10]. C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://worrydream.com/refs/Hewitt-ActorModel.pdf.

[11]. G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing," *Journal of Systems Architecture*, vol. 45, no. 15, pp. 1263 – 1277, 1999. [Online].Available: http://www.sciencedirect.com/science/article/pii/S1383762198000678.

[12]. C. Thule, "Investigating Concurrency in the Co-Simulation Orchestration Engine for INTO-CPS," Department of Engineering, Aarhus University, Finlandsgade 22, Aarhus N, 8200, Tech. Rep. ECE-TR-26, May 2016. [Online]. Available: http://ojs.statsbiblioteket.dk/index.php/ece/issue/archive.

[13]. N. Amalio, A. Cavalcanti, C. König, and J. Woodcock, "Foundations for FMI Co-Modelling," INTO-CPS Deliverable, D2.1d, Tech. Rep., December 2015.

[14]. T. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.

[15]. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. Roscoe, "FDR3 — A Modern Refinement Checker for CSP," *in Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, vol. 8413, 2014, pp. 187–201.

[16]. J. Fitzgerald, C. Gamble, R. Payne, P. G. Larsen, S. Basagiannis, and A. E. D. Mady, "Collaborative Model-based Systems Engineering for Cyber-Physical Systems - a Case Study in Building Automation". *INCOSE*. Edinburgh, Scotland. July 2016.

[17]. Microsoft, "Acquiring high-resolution time stamps (windows)," https://msdn.microsoft.com/en-us/library/dn553408(v=vs.85).aspx, 2015, (Visited on 05/03/2016).

[18]. N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, Feb 1995.

[19]. M. Reiser, *The Oberon System: User Guide and Programmer's Manual*. New York, NY, USA: ACM, 1991.

[20]. H. Sutter, "A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 16–23, 2005.

[21]. K. E. Harper, J. Zheng, and S. Mahate, "Experiences in initiating concurrency software research efforts," *in Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering – Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810316. DOI: 10.1145/1810295.1810316.

[22]. B. Agrawal, T. Sherwood, C. Shin, and S. Yoon, "Addressing the challenges of synchronization/communication and debugging support in hardware/software cosimulation," in *VLSI Design, 2008. VLSID 2008. 21st International Conference on VLSI Design (VLSID 2008)*, Jan 2008, pp. 354–361.

[23]. W. Bishop and W. Loucks, "A heterogeneous environment for hardware/ software cosimulation," *in Simulation Symposium, 1997. Proceedings., 30th Annual*, Apr 1997, pp. 14–22.

[24]. D. Kim, Y. Yi, and S. Ha, "Trace-driven hw/sw cosimulation using virtual synchronization technique," in *Design Automation Conference, 2005. Proceedings. 42nd*, June 2005, pp. 345–348.

[25]. D. Becker, R. K. Singh, and S. G. Tell, "An engineering environment for hardware/software co-simulation," in In 29th *ACM/IEEE Design Automation Conference*, 1992, pp. 129–134.

[26]. T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[27]. Github, "Electron – Build cross platform desktop apps with JavaScript, HTML, and CSS", version 1.2.2, http://electron.atom.io/, (Visited on 10/06/2016).