ERS - Enterprise Resource Simulator: a New Simulation Platform

Mark Oostveen, Michel Hofmeijer*, Fred Jansma

ERS development team, Incontrol Enterprise Dynamics, Jaap Bijzerweg 21 A, 3446 CR Woerden, The Netherlands; **Michel.Hofmeijer@incontrolsim.com*

SNE 33(3), 2023, 125-132, DOI: 10.11128/sne.33.sw.10654
Received: 2023-10-18
Accepted: 2023-10-22
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. ERS is a new simulation platform that allows you to develop and run simulations fully utilizing modern hardware. ERS supports multi-formalism in a simulation model and utilizes a new mechanism to leverage new techniques to scale models without fundamental size limits. The ERS Platform provides development tools alongside the simulation engine. ERS aims to integrate with other tools and platforms. ERS allows the development of tailor-made applications and libraries based on the engine. Those libraries are, in principle, interoperable unless specified. This allows experts in the field to create plug & play applications and libraries to share inside the ERS ecosystem, including in specialized fields like Material handling, logistics, crowd management, chemical materials, etc.

Introduction

Enterprise Resource Simulator (ERS) is the new simulation platform developed by InControl (Enterprise-Dynamics). ERS provides an environment to develop, maintain, and run a significant variation of custom state-of-the-art simulation applications. ERS provides these applications with a new powerful engine that allows them to simulate what they need without worrying about the most technical aspects of building a simulation application. Using ERS, users can build applications with their expertise while using InControl's simulation expertise. By building your own application on top of the ers-core, you can have a large degree of freedom in how your application simulates and runs.

ERS does not just allow the applications to be built with the current state-of-the-art simulation capabilities but advances that state-of-the-art based on the demands of industry and science alike. ERS does this by enabling a skilled developer to create large-scale applications that can run with the proper computational resources. The new platform does this by enabling users to efficiently split Models so that computational resources can be used to their full potential.

ERS provides access to state-of-the-art simultaneous computation and multi-formalism in one Model. The broad possibilities of ERS allow the developer to build applications that can intuitively simulate the user's problems without worrying if it fits the formalism chosen by the platform.

This paper will explain how ERS works and how using ERS can benefit the developer of simulation applications and the user. We will explain the use case of ERS. Also, we will explain how ERS works from a technical point of view and why we choose the design of ERS. Later in the paper, we will explain how this setup allows us to develop applications that can have Models with multiple formalisms within one Model. Lastly, we will explain why the technical setup will lead to good performance and scalability.

1 Technical Overview

In this chapter, we will discuss the technical architecture of ERS. While ERS as a platform has many features and abilities, the most important for this paper is the core simulation abilities of ERS. ERS does not define the logic of the Model but does still run the Model. ERS provides specific built-in tools that enable ERS to run complex Models efficiently.

1.1 Model Structure

Before we discuss how ERS works, we need to introduce four core concepts of the architecture of ERS. The first of these concepts is the Model.

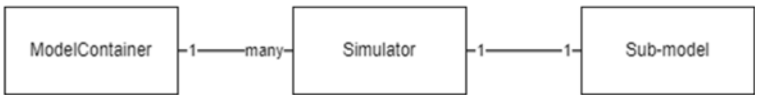

Figure 1: View of relationships between concepts, specifically between Sub-Model, Simulator, and Model.

The **Model** is the environment that ensures that all parts of the simulation-model are synced, it handles the run and the communication within the simulation-model. If two Models are loaded into ERS at the same time, they function completely separately. The Model is essential for implementing multi-formalism through a Lookahead-Table.

All data needs to be inside the Model or have a connection explicitly defined in the Model. One default connection mandatorily created is with the Shared Space. The **Shared Space** is unique per Model which can contain predefined objects accessible in all Sub-Models. The Shared Space can also contain assets that the Sub-Models share. This can be, for example, 3D models, shared functions, or shared static data. The only limitation is that objects in the shared space should not be changed during run time. The limitation on changing objects in the Shared Space at runtime is to enable ERS to parallelize execution automatically. A Model can be multi-formalistic and can be very complex. The smaller constituent parts of the Model are called Sub-Models. The Model is also where the initial random number generator is responsible for creating a valid state in the Sub-Models, ensuring determinism.

A **Sub-Model** is a largely independent uni-formalistic part of the Model. What would be considered a complete Model in most software applications would be considered a Sub-Model in ERS. Sub-Models are extremely flexible, almost anything could be in a Sub-Model, and their primary purpose is to use the efficient computation possibilities and multi-formalism built into ERS. Sub-Models implement their random number generator to ensure determinism in the Model. Sub-Models can contain entire physical environments, or they could simply contain a single algorithm. The decision to create a Sub-Model should primarily depend on how separate the new Sub-Model will be from the overall Model.

The **Simulator** is the object that runs a Sub-Model and manages the sync-events and time within its Sub-Model. Each Simulator has one unique Sub-Model. Each Simulator is uni-formalistic and communicates with the Model to sync the simulation-model. The Simulator has its own internal time system, which regulates the uni-formalistic Sub-Model.

For a Simulator to be used in a Model, it is tuned to a specific formalism to handle time correctly. The Simulator can run largely independently and only needs to stop when it syncs with the rest of the Model. This makes multi-threading more efficient, as when the Sub-Models are mostly separate, they can work on separate threads efficiently, reducing overhead, and conserving data bandwidth.

1.2 Jobsystem, Lookahead-Table and Syncing

In ERS, the Models can be extensive, and many processes can be scheduled and executed simultaneously. To manage all these different tasks, possibly at the same time, we have implemented a JobSystem. This system schedules all the tasks that follow the user's logic so that the computation resources are kept sufficiently busy. The system can be divided for ease of understanding into two parts: jobs within a Sub-Model and jobs that do not belong to one Sub-Model.

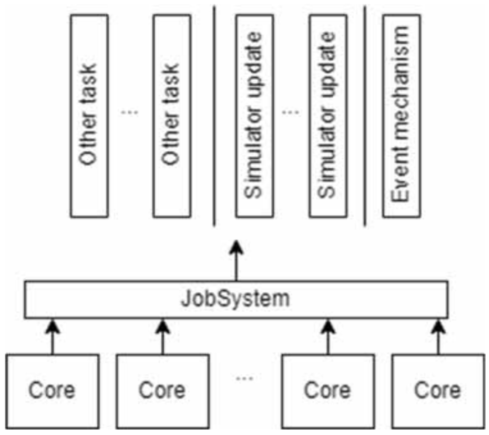

Figure 2: The JobSystem, where multiple threads are working together to execute five active jobs of the JobSystem simultaneously.

The JobSystem generally aligns with other simulation software when working in a single Sub-Model.

When scheduling jobs generated within a single Sub-Model, the JobSystem follows the order defined by the user and the formalism of that Sub-Model. The basic properties of Sub-Model logic are sequential, meaning that the order of jobs in time is absolute.

However, as soon as jobs must be scheduled that involve multiple Sub-Models, this becomes more difficult. We must solve the problem of data that is needed by multiple threads that aren't necessarily at the same point in (simulation) time because we must be certain that all previously scheduled jobs must be completed and cannot change the data and structures. There are multiple ways of handling this problem.

The first possible solution to synchronization would be optimistic synchronization (Jafer, 2010), which allows scheduling jobs that need Sub-Models that are not necessarily aligned in time and saving the Sub-Models before we execute the code. If it then turns out this was not possible, we restore the Models to their pre-calculation state. However, because the size of the Models in ERS can be enormous, making frequent backups is very memory expensive.

Instead, we use conservative synchronization, where we only schedule the job when we know that both Sub-Models are aligned. Of course, this means we may have idle processor time since we might have to wait on the slowest Model to realign in time. However, it can be shown that conservative synchronization does outperform serial computation (Nicol, 1993) even in the worst case. There are also examples of conservative synchronization outperforming optimistic synchronization in every metric (Jafer, 2010). One of the reasons for the good performance of conservative synchronization is that we can use the resources, not used for the calculation of possible future states, to do background tasks. In light of that evidence and the belief that we have found a way to minimize the waste of computational resources, we have chosen conservative synchronization.

To solve the issue of processor time being wasted, we use a Lookahead-Table in combination with the JobSystem that determines the order of synchronization and at what time the synchronization job takes place for the Sub-Models in their local time.

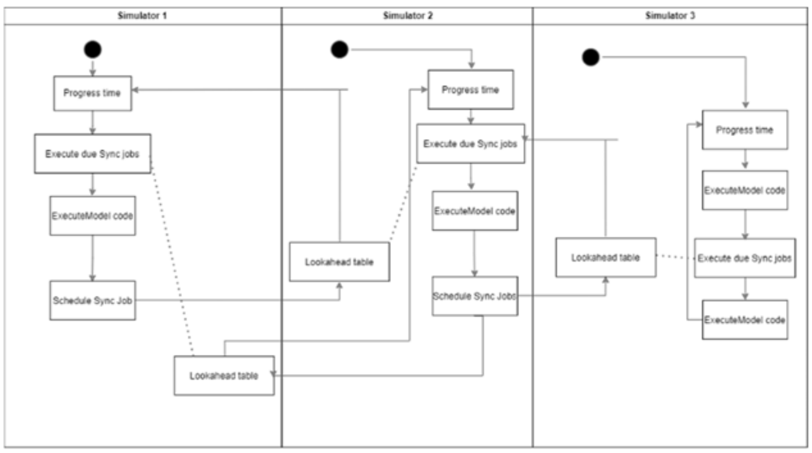

Figure 3: sync-events in the relation between Simulators.

The Lookahead-Table is a record of when each Simulator needs to be synced with each other Simulator. Each Simulator has its own Lookahead-Table. The Simulator generates the sync-events by processing each action in a Sub-Model (Figure 3). The Lookahead-Table lets the Simulators run independently until they have to sync. The user has to define a table that identifies these moments. The sync can be run during the simulation by copying the sync data to prevent modifications during the continued execution of the Model (Figure 4).

Allowing application developers to determine the sync-event time schedule makes it possible to have order-of-events-violating Models without compromising causality. The events will always obey relative causality if a sync-event is scheduled between events. The Lookahead-Table enables the application developer to determine how strict causality is enforced without leading to issues in the results.

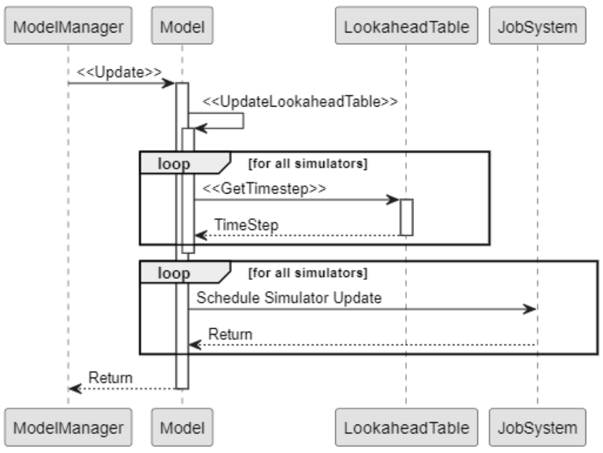

Figure 4: Update function of a Model calculating a new destination time for each Simulator, and then starting a job to update in parallel if it wasn't already.

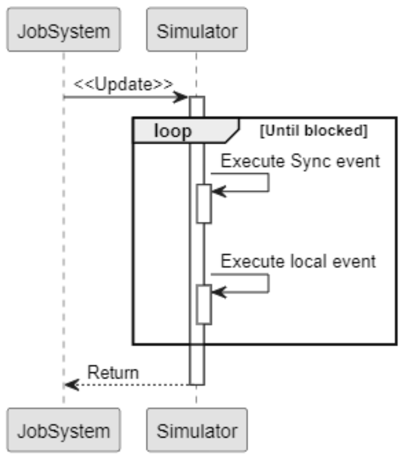

Figure 5: Update function of the Simulator, which keeps executing events until causal rules block it.

The Lookahead-Table is different between simulation runs with the same parameters, but this doesn't impact the simulation results. With the efficient splitting of the Model into Sub-Models, there should not be a constant stream of data going from one Sub-Model to another since this will make overhead much larger and reduce the benefits of multi-threading.

Another key observation is that the Lookahead-Table cannot predict if a conditional data exchange is necessary (unless explicitly given this possibility). So all possible data exchanges need to be included in the Lookahead-Table and can cause threads to wait on each other. However, ERS allows advanced users to interact with the Lookahead-Table directly by scheduling sync-events and modifying promises made to the Lookahead-Table by the Sub-Model's content.

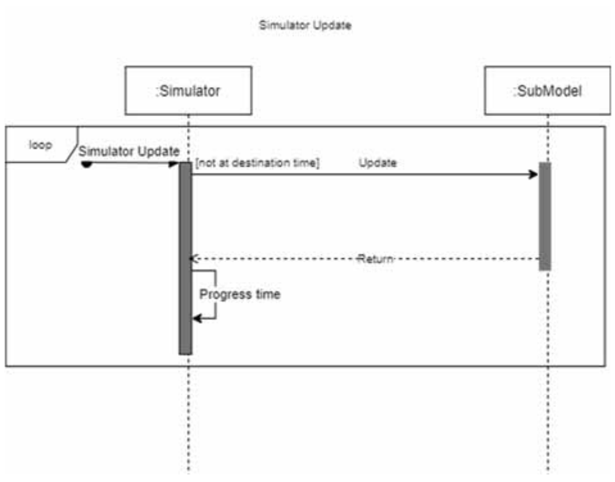

Figure 6: Update loop of a Simulator in relation with a Sub-Model.

1.3 Event Mechanism

In ERS, we found that some terminology did not suit our needs perfectly. Because of this, we will define additional terminology below.

Model-orchestrator is the time mechanism influenced by the Lookahead-Tables to orchestrate sync-events between Simulators.

Our new mechanism uses a pessimistic local hybrid-DE-orchestrator as in (Gomes, 2018) sub-section 5.1 without rollbacks on each Sub-Model. The Model uses conservative synchronization using sync-events that run in parallel time-flows and conditionally converge, which are orchestrated by a Model-orchestrator that can be distributed. The pessimistic nature of the orchestrator is more complex than traditional orchestrators. In essence, some events are executed in an optimistic fashion relative to each other, but they cannot interfere with the states relevant to each other. The "worst case" behavior is purely pessimistic but can be avoided in almost all situations.

events	Sub-Model a	Sub-Model b
Local	X_a	X_b
Non-local	Y_a	Y_b

We define the relative temporal restrictions between events in two ways. First, we categorize them as either pessimistic or type 1 optimistic based on whether an earlier event can be executed after a later event. If an earlier event can be executed later, the restriction is type 1 optimistic. The restriction is pessimistic if the later event waits until the earlier event is finished. If two events happen at the same time, there are slightly different concerns. When two events happen at the same time, the consideration is whether one event can start without the other and if the events can progress independently.

For this classification, we consider sync-events as two events, one in each Sub-Model. In this setting, the relation is type 2 optimistic when one event can start without having to wait until the other event starts and runs independently, and the restriction is conservative if both Sub-Models have to wait on each other and run together. We follow a somewhat different naming convention as syncing is more typically categorized as conservative or (type 2) optimistic. In ERS, this relation is unique to sync-events, as those are the only events that can be required to be executed simultaneously.

It is important to note that the conditions are not equivalent. Pessimistic restrictions imply that a previous event must be finished, while conservative restrictions imply that the two events have to start at the same time and run together.

These restrictions are important to allow whether we can calculate the events separately, increasing the efficiency of the calculations.

Relative temporal restrictions	X_a	X_b	Y_a	Y_b
X_a	Pessimistic	Optimistic	Pessimistic	Optimistic
X_b	Optimistic	Pessimistic	Optimistic	Pessimistic
Y_a	Pessimistic	Optimistic	Pessimistic	Pessimistic & Conservative
Y_b	Optimistic	Pessimistic	Pessimistic & Conservative	Pessimistic

Local Sub-Model Events: are events entirely inside a Sub-Model.

Model-Global-Events: are events involving multiple Sub-Models.

Pessimistic relationship: requires that events are executed in order. Otherwise, causality is broken. We do not have to specify whether it is also Conservative because two events of this type cannot be executed at the same time.

Optimistic relationship: does not require that the event is executed in the same order because events do not impact one another in any way. This relation is both type 1 and 2 optimistic since these events are fully independent of each other.

Pessimistic & conservative relationship: does require events to be executed in order and can have a shared state.

In the following table, we categorize each type of event pair as either pessimistic or optimistic. For the pairs (Y_a, Y_b) and (Y_b, Y_a) we also included whether they are conservative or optimistic. Note that In ERS, simultaneity cannot occur for two events in the same Sub-Model, so we have not categorized those events. Between Sub-Models, simultaneity can occur, but not coordinated by the Model directly, save for sync-events. So while we categorize these pairs of events as optimistic, they never interact, so there is no danger in doing so.

In the table, we show the relative temporal relations between different events inside a Model (as defined above).

2 Multi-formalism

In an ERS-based application, an application-builder is no longer restricted by the formalism dictated by the software he uses; instead, the application-builder can choose the optimal formalism for the problem.

The freedom of choice for different formalisms is achieved due to the new architecture of ERS, where the engine can handle simulations running in multiple time signatures simultaneously. This allows the user to Model in several formalisms and allows different Sub-Models to have different formalisms. These Sub-Models can interact and form a larger whole, allowing even the most complex systems to be simulated.

The strength of this way of modeling is how previously separate disciplines can be united in one Model. This allows multiple teams to work in one integrated Model. This can best be demonstrated in example case 1.

3 Example Case

Consider the case of a large international airport. The managers of this airport want to know if their airport can handle an expected increase in customers and flights. The increased number of passengers can give issues to two separate systems: terminal operations and baggage handling. Of course, these could affect each other, but only at specific points.

If the passengers are delayed checking in their baggage, they will arrive later at the security lanes. Conversely, if the security lanes become too long, this might delay flights, which will change the timing of the baggage system.

Crowd dynamics, as is needed for the security lanes, would typically be done in an agent-based simulation, while the baggage system will typically be modeled in a discrete-event based simulation.

Currently, these simulations would be made in different applications that best handle their specific situations. Resulting in two Models that would be run statically in relation to each other. In ERS, these two Models could be one Model which could correctly identify the effect the two Sub-Models would have on each other.

This would not require reading in arrival lists, and the Models can also dynamically be linked with each other and, depending on the number of syncs needed, would run in less than double the time of running the slowest algorithm. However, because it only must run once and not iteratively, it will result in much faster runs. In addition, it will result in more reliable results because the entire Model can be created in a single application running on the ERS platform, and it will allow faster debugging and working in the Model because the connections between the various parts are more intuitive.

4 Splitting Models Efficiently

One of the core concerns of any program is that it runs fast enough for its given purpose. For simulation software, this means that the program needs to scale well and be able to run within a reasonable time frame, given the proper computational resources. In ERS, the application-builder can significantly increase the application's efficiency since the application-builder can divide mostly separate processes into separate Sub-Models. These Sub-Models then can do most of their computations separately because of the JobSystem and the Lookahead-Table. The separation of these Sub-Models can speed up computations if the Sub-Models don't need to communicate too often, resulting in less overhead. The independence of these Sub-Models also allows the user to define Sub-Models so that the calculations can be distributed over multiple threads.

5 Additional Challenges

In literature, specifically in (Gomes, 2018), and (Taylor, 2019) some challenges that have not yet been explicitly discussed are identified. This part of the paper will discuss these challenges and their application to ERS.

Latency: It is recognized that latency is challenging for synchronizing multiple computers to work on 1 task. However, in most places where ERS will be used, e.g., data centers or local networks, latency will naturally be minimized due to the scaling when splitting Sub-Models. This latency problem grows smaller with the number of Sub-Models. The Model can be split into other Sub-Models. We do not need to replicate the entire Model consensus across all computers, only Lookahead-Tables for Sub-Models that influence the Sub-Model running on a computer.

Modular Composition—Algebraic Constraints: the authors identify the need for some (continuous) Models to enforce algebraic constraints at all times on several Sub-Models, making them depend on each other. This dependency can cause (near infinite) feedback loops. This is unavoidable because ERS is a platform, so we do not restrict the relations that can be defined between Sub-Models. However, the worst case does not happen as an infinite loop is not possible in ERS in that way, so it will, at some point, resolve. In general, these kinds of errors cannot be prevented by a simulation platform because it is caused by inter-Sub-Model relations, which we cannot regulate if we want to give application builders sufficient freedom. In general, this is a concern, but this is not applicable to the ERS engine.

Algebraic Loops: algebraic loops are loops created by the indirect dependence of variables on themselves. They are very similar to *Modular Composition—Algebraic Constraints*, and we accept them as possible problems because we do not limit the ability of application builders to make connections between Sub-Models.

Consistent Initialization of Simulator state: in some Simulators, the input data has to obey certain conditions to be valid. This can be seen as a sub-problem of the problem with Modular composition-algebraic constraints, in the sense that this constraint is only necessary at the start of the simulation. The argument is the same for the overall problem. At the same time, it is a problem; it is not a problem that a simulation platform can solve and instead should be handled by the application developer or the model builder.

Compositional Convergence—Error Control: in many simulation Models, there is the desire to estimate the errors related to the underlying theoretical solution. In ERS, we do not calculate this error since this is too specific to be built into a platform. Instead, this can be best handled by an application builder.

Compositional Stability: Similar to the last point, many simulations might also want to estimate the stability of the error compared to the theoretical solution. However, this problem is too specific to be handled at a platform level and should instead be handled on an application or user level.

***Compositional continuity*:** for continuous Simulators that are connected to non-continuous Simulators, it can be an issue to retain the continuity in the connection. In ERS, we allow almost arbitrarily small-time steps (up to a single Planck time).

This combats this issue as far as possible on a platform level. Special measures can be taken on an application or Model level, but a platform should not enforce these.

Real-Time Constraints, Noise, and Delay: For continuous time simulation, whether it is completely internal or part emulation, it is important to be able to support the right frequency of updating (micro-step). ERS takes three measures to support the right frequency.

First of all, as mentioned earlier, the platform does not enforce a step size limit that can be physically restrictive as time steps can be as small as a single Planck time. Secondly, ERS allows the total Model to be split into many Sub-Models so that an application can take advantage of simultaneous calculations as much as possible. Lastly, ERS can support many different types of simulations simultaneously, removing the need to model in several applications and thus eliminating the issue of bad communication, as long as no emulation is included.

However, even with these measures, implementing the right frequency is not always possible, and dealing with this remaining issue will have to be handled on an application-to-application basis.

Discontinuity Identification: In communication with continuous simulations, it is beneficial to identify discontinuities. However, the core cause of the discontinuity lies in the continuous Sub-Model or the communication between Sub-Models.

In either case, the application developer is responsible, so it should be solved on an application or model level and not on a platform level.

Discontinuity Handling: Once a discontinuity is identified, it would be beneficial to handle that discontinuity in a particular way so that the simulation can continue with reasonable accuracy.

However, because the cause of the discontinuity is in the Model, it is best to handle this on an application level and not on a platform level, because different types of discontinuities might be handled differently.

Algebraic Loops, Legitimacy, and Zeno Behavior: as we discussed earlier with algebraic loops, a more general question can be asked about whether we should detect potentially infinite event chains that either keep the simulation at the same time or represent an increasingly minor progression of time in such a way that the simulation never reaches the designated endpoint. In general, we cannot detect this behavior in ERS since it is Sub-Model specific and cannot be identified with certainty without knowledge of the internal working of the Sub-Model.

However, In ERS, loops cannot extend indefinitely since the engine will not allow sub-Planck time increments (that by definition will not be physically consequential) nor infinite scheduling on a single point in time. This means that this behavior will always end in ERS – but this will take a very long time to materialize as ERS is designed for microscopic time-scale simulation.

Stability Under X: a concern for co-simulation (Gomes, 2018), in general, is that the entire Model might not be stable, even if all the Sub-Models are stable. This is specific to a Simulator and should be handled by the application or on the model level.

Some issues might cause instability, such as noise caused by the communication between continuous time and discrete time Models. However, the instability of the whole system is still inherently caused by the design of the Sub-Models, and so this issue should be checked and corrected for by applications for which this behavior could occur.

Theory of discrete event Approximated States: With multi-formalistic co-simulation becoming commercially viable, there is a need to develop a theoretical framework for the quality of communication between continuous-time and discrete-time simulation Models. We acknowledge this, but this goes beyond the scope of this paper. InControl will engage with the science community to start the development of such a theoretical framework.

Standards for Hybrid Co-Simulation: Besides theory, a new standard for co-simulation should be established so that new Models can be developed on a solid foundation. This is, in a sense, what ERS as a platform does since it offers a way of building a wide variety of co-simulation scenarios on one simulation platform.

This means that all applications build to solve these scenarios can build on the solid foundation that ERS has established.

Semantic Adaptation and Model Composition: A central question in co-simulation is what information needs to be included in the wrapper of a simulation. In our case, this would be the Sub-Model or the Simulator. In (Gomes, 2018) it is argued that this should be specific to the Model, in contrast to how this is handled in ERS. This paradox is solved by considering that the paper includes data transfer as part of the wrapper, while in ERS, this is part of the sync event, which an application builder can alter.

Thus ERS can have a single implementation of the Sub-Model and Simulator concepts without running into problems identified in (Gomes, 2018) .

Predictive Step Sizes and Event Location: If the core time concept of a simulation engine is based on discrete-time (like in ERS), there is a question of how large the time steps should be. In ERS, application builders primarily regulate this since they can schedule their own events. Meaning that the application builder can decide the precision required.

While the precision might be sufficient, this approach might still have efficiency concerns. For example, if very high precision is used, this high precision can lead to a large number of sync-events that do not transfer data. These sync-events are unavoidable in ERS on a platform level because we use conservative synchronization, so we do not use the Lookahead-Table to resolve these sync-events. This efficiency problem is of limited severity because sync-events that transfer no information only use a tiny amount of computation resources.

In addition, the application builder can mostly prevent unnecessary sync-events, so efficiency can be high with the right implementation. Also, each Simulator allows a specific step size to incorporate high precision in sub-subsections of the entire simulation Model.

6 Conclusion

ERS is a new simulation platform for application builders who want more freedom, power, and possibilities than other simulation packages can offer. It allows for applications that model reality closely, even if reality is complex and does not follow the constraints that any specific formalism requires.

ERS works by splitting a complete Model into Sub-Models whereby each Sub-Model uses its formalism and can communicate and exchange data with other Sub-Models through sync-events. The Model causality is maintained by Lookahead-Tables, which create and maintain a time consensus that determines the latest point to which a Sub-Model can independently run. Sub-Models run concurrently or even remotely, allowing ERS to scale well.

ERS allows application-builders to reach their full potential and connect all relevant systems with reasonable run times in one Model. It does this by allowing the modelers and application builders to build a platform that can support massive models and use third-party libraries in the languages they know best.

ERS can support the needs of the application the modeler wants.

References

[1] Gomes C. T. Co-simulation: a survey. ACM Computing Surveys (CSUR). 2018 May; 51(3): 1-33.

[2] Jafer S, Wainer S. A. Conservative vs. Optimistic Parallel Simulation of DEVS and Cell-DEVS: A Comparative Study. Proceedings of the 2010 Summer Computer Simulation Conference. 2010 July; (pp. 342-349.).

[3] Nicol, D. M. The cost of conservative synchronization in parallel discrete event simulations. 1993 April; J. ACM, 2(40), 304–333. doi:https://doi.org/10.1145/151261.151266

[4] Taylor, S. J. Distributed simulation: State-of-the-art and potential for operational research. European Journal of Operational Research, 2019 October; 237(1), 1-19.