A Model-driven Middleware Integration Approach for Performance-Sensitive Distributed Simulations

Travis Brummett
CS, Tennessee Tech Univ
Cookeville, TN, USA
tbrummett@tntech.edu

Kyoungho An
Real-Time Innovations (RTI)
Sunnyvale, CA, USA
kyoungho@rti.com

Aniruddha Gokhale
EECS, Vanderbilt University
Nashville, TN, USA
a.gokhale@vanderbilt.edu

Sanders Mertens
Zoox
Foster City, CA, USA
smertens@zoox.com

***Abstract*—Complex simulation systems often comprise multiple distributed simulators that need to interoperate and synchronize states and events. In many cases, the simulation logics which are developed by different teams with specific expertise, need to be integrated to build a complete simulation system. Thus, supporting composability and reusability of simulation functionalities with minimal integration and performance overhead is a challenging but required capability. Middleware for game engines are promising to realize both the modular and reusable development criteria as well as the high performance requirements, while data-centric publish/subscribe middleware can support seamless integration and synchronization of the distributed artifacts. However, differences in the level of abstraction at which these middleware operate and the semantic differences in their underlying ontologies make it hard and challenging for simulation application developers and system integrators to realize a complete, operational system. To that end this paper presents a model-driven approach to blending the two middleware, wherein the modeling capabilities provide intuitive and higher-level abstractions for developers to reason about the composition and validation of the complete system, and the generative capabilities address the inherent and accidental complexities incurred in reconciling the semantic differences between the gaming and pub/sub middleware. We present a concrete implementation of our approach and illustrate its use and performance results using simple use cases.**

***Index Terms*—Composable Simulation, Distributed Simulation, Model Driven Engineering, Entity Component System, Data Distribution Service, Automation.**

I. Introduction

Simulation is often the preferred approach when it comes to studying complex cyber physical systems (CPS) or conducting training exercises to use them, e.g., in operating an aircraft. Simulation environments for complex systems often require multiple simulators to interact with each other. For example, to evaluate new smart city traffic management solutions, multiple disparate simulators, such as a vehicular traffic simulator (e.g., SUMO or Carla), a network simulator (e.g., NS3 or OMNeT++) and a physics simulator (e.g., Simulink), must be integrated to realize the complete system simulation. The simulation code for individual system functionalities within an integrated solution are often developed independently by different developers. Consequently, a simulation application must then be realized somehow as a composition of such different simulation functionalities. In these situations, the developer and integrator of the simulation application incur numerous challenges in composing simulation functionalities in the correct manner and deploying the integrated application on distributed resources.

Since the simulation application must be realized through composition, there is a need for an abstraction that permits composition yet does not result in noticeable performance and memory footprint overhead. Further, since several of these simulations may also be interactive (e.g., a flight simulator) and involve one or more collaborating human actors, these abstractions should be responsive and meet the real-time requirements of interactive computing. Middleware solutions designed for high performance gaming engines hold significant promise to meet these needs. Specifically, the Entity Component System (ECS) architectural pattern [12] enables developers to organize code in a way that makes game development to be modular and composable within single address spaces. These properties of ECS have been exploited recently to develop interactive and responsive simulations for military training [6], [13]. Moreover, since ECS favors composition over inheritance in addition to a strict decoupling of data and logic, reusability of simulation code is significantly improved which further helps with independent development of simulation modules by different parties.

Although an ECS-centric solution can help realize a composable and responsive simulation application, the complex and integrated nature of the simulation application implies that the simulation developer must now be responsible for interactions and synchronization between simulation components hosted on their individual simulators that may be deployed across distributed resources, i.e., across different address spaces which is not supported by ECS. These communicating entities rely on events and messages that carry state information. Data-centric publish/subscribe (pub/sub) middleware holds promise in addressing these challenges since it not only provides location transparency thereby relieving the simulation developer from the deployment and configuration challenges but also serves as the highly performant transport mechanism for transferring events and states between the disparate simulators [11].

The ECS and data-centric pub/sub paradigms solve the individual challenges of composition and distribution, respectively, however, only in isolation. None of the paradigms can support both requirements all at once without significant additional

feature support. Blending the two different paradigms thus holds the most promise but still incurs multiple challenges. First, the level of abstraction and the problem addressed by each of the two paradigms is different: ECS handles composition of simulation functions and their computations within a single process while data-centric pub/sub middleware integrates distributed processes by sharing their data with each other. Second, there exists a significant semantic gap between the two paradigms due both to the problem each paradigm tries to address and also due to the differences in their ontologies. For instance, the semantics of a *component* in ECS are vastly different from that in distributed component-based frameworks based on data-centric pub/sub. Thus, there is a compelling need for a systematic approach that can (a) bridge this semantic gap (b) be easy and intuitive to use for the simulation application developer, and (c) provide the responsive behavior expected by distributed simulations.

To address these challenges, we present the *Model-driven Integrated Data Distribution and Entity Component System* (MIDDECS) middleware to bridge the semantic gap between the two different paradigms: ECS and data-centric pub/sub. Our solution is influenced by a concept called *Model-driven Middleware* [4] that propounds the use of intuitive, higher level abstractions in the form of domain-specific artifacts to resolve a number of accidental complexities stemming from the deployment and configuration of composable systems, e.g., modern day microservices applications such as distributed, composed simulations. In contrast to the original MDM idea, we extend it to address both the inherent and accidental complexities outlined above in realizing composable and high performance distributed simulations.

The research we present in this paper is the result of investigations into industrial, real-world problems and a focus on developing solutions that can be easily deployed in practice. To that end, our solution leverages widely used, open source and open standards technologies. Simulation applications developed using MIDDECS can exchange simulation data items via strongly typed and type-extensible interfaces thereby ensuring interoperability between disparate simulators. MIDDECS provides a domain-specific modeling language (DSML) [7], [8] for development, composition, and validation via a Web-based, generic modeling tool. Specifically, this paper makes the following contributions:

- We present our DSML that bridges the semantic gap between the underlying two paradigms while providing intuitive abstractions to the model developer. Its generative capabilities then promote automation to overcome the tedious and error-prone composition and integration challenges.
- We showcase our solution using concrete technologies: WebGME (www.webgme.org), which is a Web-based DSML development environment; Flecs (github.com/SanderMertens/flecs), which is an ECS framework; and RTI Connext DDS (www.rti.com/products/connext-dds-professional), which is an implementation of the OMG Data Distribution Service, a data-centric pub/sub middleware [10].
- We demonstrate our ideas and the performance of the integrated solution using simple but representative use cases that highlight key features of our solution.

The rest of the paper is organized as follows: Section II provides background on the underlying technologies used in this work, and compares our work with prior work; Section III presents challenges and our solution; Section IV presents validation of our ideas in the context of simple use cases; and finally Section V presents concluding remarks alluding to lessons learned and opportunities for future work.

II. Preliminaries and Related Work

This section describes the scope of our work and assumptions we made, background on the underlying technologies used by MIDDECS, and compares our work with prior efforts.

A. Scope of the Work and Assumptions Made

Simulations are widely used in today's world whether it be for training, analysis, or test and evaluation. Often, the different aspects of these simulations are designed by multiple individuals who may have little or no knowledge of the other components designed by others that will be interacting with their own. Thus, it becomes helpful to have a generalized framework and tool for designing and deploying such simulations in distributed environments. Our approach to this problem is to strip away the complexities of designing these components and allow users to design and deploy simulations using a visualization tool based on a domain-specific modeling language and deploy them on infrastructure comprising two different paradigms: ECS and DDS, which are explained next.

In this paper we are not concerned with dynamic resource management or deciding how best to compose and distribute the functionalities. We are also not concerned with specific deployment scenarios, such as a cloud data center or at the edge or a mix of these two and the challenges that stem from doing so. These form dimensions of future work.

B. Background on Underlying Technologies

In the following we provide relevant background on each of the three underlying technologies we used in this work.

1) Overview of the Entity-Component-System Paradigm: ECS is an approach towards organizing code used often in game development. ECS designs favor composition over inheritance, and strict decoupling of data and logic, which improves the reusability of code. ECS applications are constructed out of three concepts: entities, components and systems.

- **Entity:** An entity is a unique identifier, typically of an integer type, that represents an object being simulated. Entities can be created and/or deleted while the simulation is running. A simulation may contain hundreds of thousands of entities.
- **Component:** A component is a plain data type describing a single aspect of an entity. Typical examples of components are "Position", "Velocity", "Rotation" or "Mass". Components do not contain any logic (e.g. methods). An

entity may be composed out of zero or more components. Components may be added and/or removed from entities while the simulation is running. Complex simulations may contain up to hundreds of components. Based on this definition it is obvious that the notion of a component in ECS is quite different from the well-known notion of components or microservices, which are units of functionalities that can be composed.

- **System:** A system is a function that implements a single aspect of the simulation logic. Systems are matched against entities that contain a set of components. For example, a system that moves entities would request to be matched with the "Position" and "Velocity" components. When invoked, the system will iterate over all of the matched entities and perform the simulation logic.

2) Overview of OMG Data Distribution Service: The Object Management Group (OMG)'s Data Distribution Service (DDS) standard defines a data-centric, topic-based publish/subscribe (pub/sub) model for distributed communications [10]. The DDS pub/sub architecture promotes loose coupling between applications with respect to time (i.e., the applications need not be present at the same time) and space (i.e., applications may be located anywhere). DDS also provides more than 20 configurable Quality of Service (QoS) policies. QoS is a concept that is used to specify the behavior of a service (e.g. reliable transmissions and persisting historical data). With configurable QoS policies, developers can easily define and update desired behaviors of applications. It can reduce complexity of application development by separating functional (i.e., business logic) and non-functional requirements (i.e. quality of the logic).

3) Overview of WebGME: Our domain specific modeling language (DSML) is created using the Web-based Generic Modeling Environment (WebGME). WebGME is a Web-based modeling tool. It supports multiple features that include (1) creating meta-models and models, (2) model visualization and (3) model interpreters known as plugins. In our approach, we leveraged WebGME's Unified Modeling Language (UML)-like syntax and constraint specifications to create our DSML. Model interpreters can be associated with the meta-models, which can provide additional semantics to the language that are not captured in a visual form as well as provide the generative capabilities needed for automation.

C. Comparison with Related Work

We now compare our work on MIDDECS with prior efforts. We focus on a sampling of efforts that comprise model-driven techniques to address different challenges encountered in realizing large-scale simulations.

The authors of [6], [13] present a semantics-based approach which facilitates decoupling algorithms provided by high-level tailored simulation modules or engines from the object structure of the low-level entities simulated by those engines. This allows their approach to maintain the benefits of the object-oriented paradigm. These benefits include encapsulation and access control. In contrast, our approach leverages the ECS paradigm with a model driven interface. This allows users to build their simulations and define their entities with the modeling language thereby achieving the ability to reuse entities in a different way. Our approach also focuses on distributing the simulations using the ECS and Pub/Sub paradigm bridging the gap between the two using a modeling language.

In [9], the authors describe a model-driven integration platform for simulation of CPS. The authors recognize the need to execute different heterogeneous simulators since no single simulation framework can simulate all aspects of a CPS. This work demonstrates the use of High Level Architecture (HLA) [3] to serve as the communication substrate to bridge the different simulators while the different APIs of the simulators are bridged using the Functional Mock-up Interface (FMI) [2]. Although this work showcases a nice example of integrating heterogeneous simulators, there are two issues that separate our work from theirs. First, our work requires the integration of two different middleware, i.e., ECS and pub/sub, that co-exist at different levels of abstraction whereas the prior work focuses only on the communication transport between the heterogeneous simulators. Second, their federated (i.e., distributed) approach poses several technical problems that can sometimes severely limit the performance of simulations. HLA was not designed to support low latency data sharing required for large complex simulations. Specifically, the HLA Runtime Infrastructure, which is the implementation of the interface specification of HLA, does not sufficiently support large simulation federations that require frequent data exchange. Additionally, HLA does not have a standard wire-protocol format so interoperability between federated models is challenging with the increasing complexity of simulation software. This is because it delegates the responsibility for serialization/deserialization to the application. Moreover, the lack of security support leaves these simulation systems vulnerable to security attacks. This is the reason why our work chooses data-centric pub/sub as a way to support low latency communication between the disparate simulators.

The effort in [5] describes an extension of the UML/BPMN/SysML tool called Papyrus to support industrial strength co-simulations for which the authors rely on FMI. Although this work also illustrates the need for bringing together heterogeneous simulators and uses FMI to bridge their APIs, this effort is tailored towards making these features available as part of a model-driven development framework, Papyrus. In contrast, our work illustrates how model-driven techniques such as DSMLs can be used to bridge the gap between the middleware that support the simulations. Our MDM approach can be thought off as an alternative to using FMI.

III. MIDDECS Model-driven Middleware Solution

We now present our MIDDECS solution that addresses the challenges in bridging disparate middleware to support an extensible, composable, and distributed simulation environment for complex, real-world systems.

A. Challenges and Requirements to Realize MIDDECS

Before we present our solution, we highlight the challenges in realizing composable and distributed simulations. Addressing each challenge then poses the following solution requirements that MIDDECS must meet.

Requirement 1: Composable and high performance, responsive simulations: The use cases we faced in our industrial research involve hosting simulations in which the simulation logic for different parts could be developed by different developers, possibly, collaboratively. Thus, a critical need for us was to ensure modularity and composability of the simulation logic to form a complete simulation application. Moreover, such a composed simulation must be highly performant and responsive, particularly when humans interact with the system. Thus, there is a need for a capability that can allow simulation functionalities to be developed by different developers yet can be composed and assembled into a simulation that can execute and even provide real-time performance, particularly for simulations involving interactive training.

Requirement 2: Interoperable and extensible interfaces: Adapting individual simulations to integrate with other simulations requires significant software development and maintenance efforts, which can be very costly. To integrate heterogeneous and distributed simulators requires an interoperable connectivity framework that allows simulations to exchange state data and mutual interactions among each other. Interface extensibility is also critical to facilitate system evolution by enabling seamless integration of distributed simulations that may evolve in terms of interface definition. To support this need, well-defined standard extensibility rules for interface definition as well as a wire-protocol format are required.

Requirement 3: Intuitive and higher level of abstraction: Satisfying requirements 1 and 2 is a necessary but not a sufficient condition to realize the end goals because in trying to address these requirements, the responsibilities of the simulation application developer will certainly increase. For instance, developers must now learn how to encapsulate the simulation logic to be reused and composed. They must then be able to compose the logic to build a simulation. Moreover, depending on the performance demands and availability of resources, the simulation may have to be distributed across networked resources, which in turn will require the developer to provide glue code to interconnect the distributed elements of the simulation not to mention also taking care of serialization/deserialization issues. Furthermore, it is often the case that the technology used for encapsulating the logic and composing the independent units has no support for distribution, and hence its semantics must be reconciled with that of the distribution technology.

In summary, the simulation application developer faces a plethora of inherent and accidental complexities. Thus, there is a critical need to provide the developer with higher levels of abstraction to reason about the system composition and leave it to the tooling to decide how best to decompose, distribute and deploy the functionality across the available resources.

B. Meeting the Requirements: The MIDDECS Approach

The MIDDECS approach illustrated in Figure 1 leverages three key technologies as follows: in order to build a modular simulation logic (Requirement 1) we adopt the Entity Component System (ECS) approach, which allows us to build simulation entities that flexibly compose simulation logic (i.e., systems in ECS) built by other developers. ECS applications are deployed in a distributed way and use OMG DDS as the interface for communications (Requirement 2). For raising the level of abstraction and bridging the semantic gap between ECS and DDS (Requirement 3), we relied on WebGME (www.webgme.org). The model we created in WebGME acts as the abstraction layer to develop an ECS-based distributed simulation application model. In the rest of this section we describe how each of the requirements is satisfied by the technology choices we made.

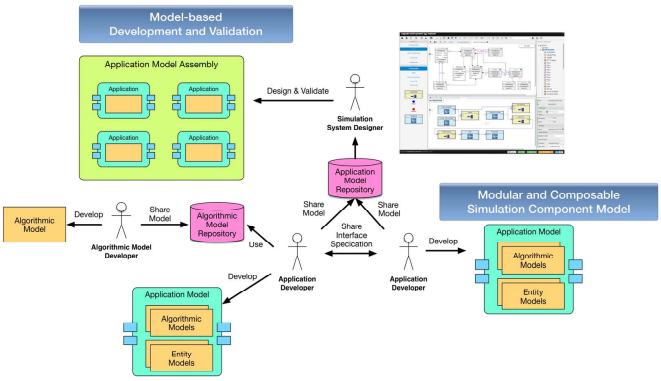


Fig. 1. Architectural Overview of the MIDDECS Solution

1) Meeting Requirement 1: Use Entity Component Systems (ECS): Requirement 1 calls for a highly performant solution that enables composable simulation logic – a property that is satisfied by the Entity Component System (ECS) architecture pattern [1], [13]. An efficient ECS implementation stores component data in one or more arrays. When systems process entities, they iterate over the component arrays allocated in contiguous memory. As a result, ECS applications typically have less cache misses and improved CPU cache utilization, which improves system performance. ECS code is therefore often able to process more data (e.g. simulated objects) than code that adheres strictly to object orientation. To that end, we have leveraged a specific open-source implementation of the ECS paradigm called Flecs (https://github.com/SanderMertens/flecs).

As an example, consider a *Gravity* system in ECS. In this example, an entity's position component may contain X and Y coordinates. By attaching that position component to an entity, we give it those coordinates and therefore a position. But positions can change over time. To change components or attach or remove them from entities, the notion of systems are used. Systems analyze every execution loop of all the entities, and perform any necessary changes. For example, if we attach a gravity component to an entity, it will start being pulled along

the y-axis. A position system might check for the existence of a gravity component on every entity, and if it has one, update the entity's Y coordinate value in its position component. In this way, the entity will "fall" down the y-axis.

2) Meeting Requirement 2: Use OMG Data Distribution Service (DDS): Requirement 2 calls for a type-safe approach to distribute the simulation units by providing location transparency, disseminating events, and taking care of (de)serialization issues all while providing responsible behavior – properties supported by the OMG DDS standard. When ECS data is distributed between applications, component arrays are exchanged directly between the ECS framework and DDS middleware. This minimizes protocol overhead of DDS as data is sent in large batches, while at the same time avoiding redundant copying of memory between layers. The DDS synchronization code is implemented as an ECS system which decouples it from the simulation code. This allows developers of simulation logic to be unaware of the distribution mechanism. Furthermore, the decoupling enables late-binding of simulation data with DDS, such that changing which components are distributed and which remain local is straightforward. This further enhances the reusability of applications and application code. Figure 2 depicts the approach showing how ECS encapsulated simulation components can communicate via the DDS bus.

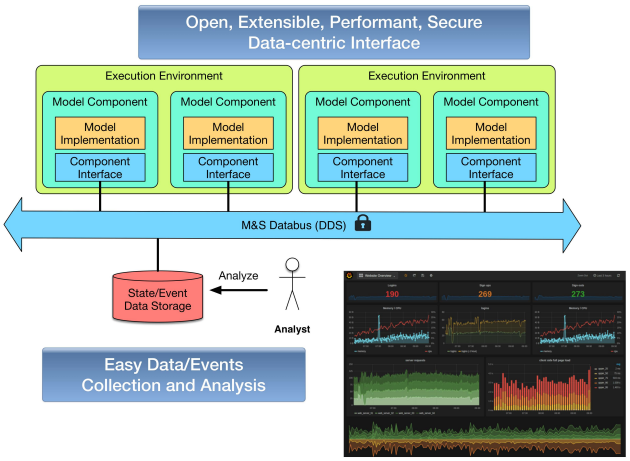


Fig. 2. Integrating ECS with DDS

Additionally, DDS uses typed interfaces. This approach fills the gap between the typed interfaces (e.g. Protocol Buffers) and the pub/sub messaging protocols (e.g. MQTT and AMQP). With typed interfaces, developers can directly use and manipulate native data types of programming languages. Type interfaces are safe because data types can be validated at compile time. They are efficient as they can be serialized in binary format. DDS also standardizes extensibility of data types called DDS Extensible Types (DDS XTypes). With DDS XTypes, the data types used by DDS applications can gracefully evolve by adding new elements without affecting other compiled applications.

DDS standardizes its interface definition language, a wire-protocol format, APIs, QoS policies, and interface extensibility. Therefore, our DDS-based solution can achieve interoperability of distributed simulations in a reliable and flexible way. For our implementation, we have used RTI Connext DDS (https://www.rti.com/products/connext-dds-professional) as the communication middleware to support ECS-based simulations that are distributed across networked nodes.

3) Meeting Requirement 3: Use Domain-specific Modeling: Requirement 3 calls for raising the level of abstraction at which the simulation developer can reason about the complete system and be able to address both the inherent and accidental complexities incurred in bridging the semantic gap between ECS and DDS middleware. To that end, we leveraged WebGME (www.webgme.org) to realize the model-driven middleware solution, which involves defining the syntax and semantics of a DSML through meta-modeling (see Figure 3).

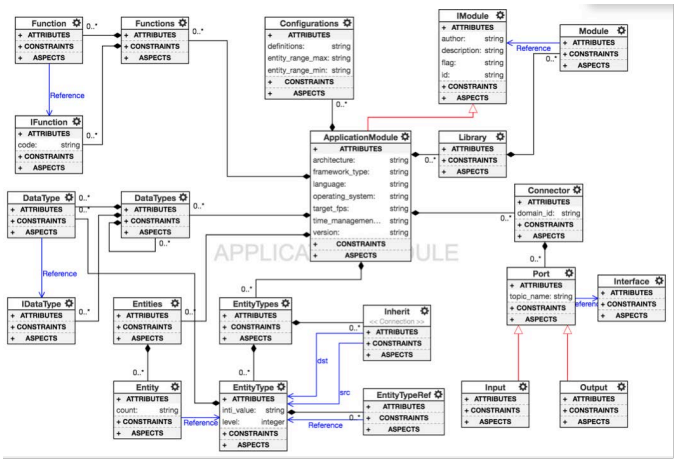


Fig. 3. Application Module Model defines modeling elements for a composable and distributed simulation application using ECS and DDS

Figure 3 depicts the meta-model for the Application Module. This is the highest level of our DSML where users can first define an Application and specify its attributes. The application can contain predefined libraries to configure it. An application may also encapsulate connectors containing zero or more input and output ports, which are needed to pass information across distributed deployments using a pub/sub approach. An application publishing data must have an output port or publisher that matches an input port or subscriber of the receiving application. The application will contain entity types, which are similar to a class within an object oriented paradigm. The entities contained within the application are specific instances of entity types that are defined by the user. An application within our DMSL will also contain data types which will be used to create ECS components. A component can be made up of one or many basic data types and a component can be composed with other components to build a simulation entity type. For example, a Position component has two integers. One corresponding to a value on an x-axis and another on a y-axis. A Color component has a string

corresponding to a value of color. When a user creates a simulation entity having both position and color attributes, the Position and Color components can be composed for a simulation entity type and corresponding simulation functions for these components are automatically applied to the simulation entities. Finally, an application can contain functions, which are ECS systems. They could be things like gravity that are applied to one or more entities. These functions are able to act on the components attached to an entity.

The functions supported by the DSML are one of four types. This typing dictates when the function is applied to the module in which it is a part of. The first type applies the function to the module that it is attached to upon the start of the application. This function type is called **onStart**. Similarly, a second type of function, **onStop**, is applied when the application concludes. Functions that are invoked periodically based on settings supplied by the user are referred to as **onPeriod**. Finally, functions can be invoked when external data is acquired. These are known as **onUpdate** functions. These different types give the user more control over defining their ECS systems.

Data types (i.e., ECS components) also have multiple types and can be used as parameters for functions (i.e., ECS systems). Currently, our DSML includes primitive data types such as integer or float but not all the possible data types represented in a programming language. However, this can be expanded upon in the future. When used as parameters to a function, data types can serve as one of three types of parameters. They can either be read only, write only, or read and write. In our model these are referred to as **InputParam**, **OutputParam**, and **InOutParam**, respectively.

Users can use MIDDECS to deploy distributed simulations using DDS as the communication bus by including a connector into an application module. Connectors contain ports which can either be input or output ports representing DDS writers and readers, as explained above. They serve as the means for separate applications to communicate with each other to share states and events of entities.

After the application model is built, a user uses special plugins, i.e., model interpreters developed within WebGME for our DSML. Using the plugins, MIDDECS will then traverse the application model and generate the necessary code to launch such an application. This means that the users need not concern themselves with the complexities of ECS or DDS beyond building their application module because the plugin will handle most of it for them. In addition, another plugin was developed to check the validity of the DDS interfaces. Currently, it does so by looking at two user applications that are bound by a connection and ensuring that the source application contains an output interface of the same topic as an input interface in the destination application. If this is the case then the connection is valid. Otherwise, the connection is invalid. In its current state the verifier merely reports on the validity of connections rather than attempting to correct them.

In summary, our DSML provides the syntax and semantics to bridge the ECS and DDS paradigms which enables a user to build a ECS-based simulation application that can use DDS for data distribution by dragging and dropping predefined models. Using this approach we can successfully abstract away the complexities of composition of simulation logic and integration of distributed simulations. This effectively creates a layer over the ECS and DDS middleware yielding our vision of a model-driven middleware (MDM).

C. MIDDECS Usage

The following describe the steps undertaken by a user of MIDDECS:

1) The Model Developer develops algorithmic functions with given data types as input or output parameters.
2) The Component Developer develops simulation application components that can use algorithmic models shared via an Algorithmic Model Repository.
3) The Component Developer shares interface specifications with other Component Developers so that they can develop a component to be composed via the interface.
4) The Simulation System Developer composes and validates application components shared via an Application Component Repository by using a modeling tool.
5) A modeling tool generates deployable artifacts. They will run via execution environments and communicate through a DDS-based databus.
6) External analysis tools can collect simulation states and events via the DDS-based databus.

IV. Evaluating the MIDDECS Solution

In this section we evaluate the properties of MIDDECS and its performance in the context of two simple but representative simulation examples. Our first example is a *shapes* example where different geometrical shapes are modeled and simulated. This example can be considered as a "Hello World" example for MIDDECS. Our second example is drawn from a real-world use case comprising radars that monitor airplanes and ships in civilian and military situations.

A. Modeling and Deployment using MIDDECS

To demonstrate modularity, composability and type extensibility of our solution, we developed our applications with the meta-model described in the previous section and then generated application code of an ECS framework. The Radar example application uses the data model, a snippet of which is shown in Figure 4. DDS data models represent the data structures of messages exchanged between DDS-based applications in an Interface Definition Language (IDL). Data structures in IDL (e.g. Position2D and Radar) correspond to ECS components. A composition of these data structures is an interface used by a DDS-based application (e.g. IShip).

Figure 5 illustrates how a user can define a model for the application using our DSML. Due to the complexity of the Radar example, we show the simpler Shape example in this figure. Specifically, it displays simulation models (e.g. Square) encapsulated with ECS components (e.g. Position) and the simulation models are connected by DDS databus. The DSML

```
module TrafficControl {
        /* -- Entity id -- */
        typedef long long Entity;
        /* -- Components -- */
        struct Position2D {
          float x;
          float y;
        };
        struct Radar {
           float t;   /* current angle */
           float v;   /* velocity */
           float r;   /* radius */
         };
         /* -- Interfaces -- */
         struct IShip {
           string app_id; // @key
           sequence<Entity, 65535> entities;
           sequence<Position2D, 65535> position;
           sequence<Rotation2D, 65535> rotation;
         }; //@Extensibility FINAL_EXTENSIBILITY
};
```

Fig. 4. Data Model for the Radar Application Components and its Interfaces in IDL

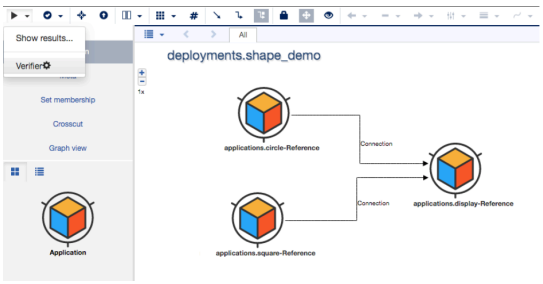


Fig. 5. Modeling using our DSML

is then used to synthesize glue code and other artifacts that integrates ECS and DDS to host the application.

To demonstrate modularity and composability of simulation logic, we created entities built with different types of components. For example, a Circle entity is composed of Position, Size, and Velocity components while a Square v1 entity is composed of Position, Size, Velocity, and Gravity components. Because the Square v1 entity has the Gravity component, the movement of the Square v1 entity is affected by a gravity system (e.g. pulling down the Y coordinate for 2D simulation). Since simulation logic (e.g. Gravity) is developed as a modular ECS system, the logic can be easily reused and composed.

Similarly, the Radar application comprises the Plane, Ship and Radar processes. The Plane and Ship processes are made up of the Plane and Ship entities, respectively, while the Radar process is made up of the Radar and Track entities. In turn, the Plane and Ship entities each are composed of Position, Rotation, Speed and Angular Speed components; the Radar entity is composed of Radar Range, Radar Sweep and Radar components; and the Track entity is composed of Size, Color and Track Data components. Figure 6 shows a deployment for the Radar application enabled by the MIDDECS solution. In this deployment, each application process simulates the movement of its corresponding entities, e.g., Ship application simulating the Ship entity while Radar simulates the Radar and Track entities. Each process is distributed and shares entity states with each other (e.g. Position and Speed of Ship or Plane Entities) via DDS-based interfaces over the DDS messaging bus, which could be shared memory or RTPS, which is a real-time pub/sub transport over UDP. Such a deployment is seamlessly made feasible by MIDDECS.

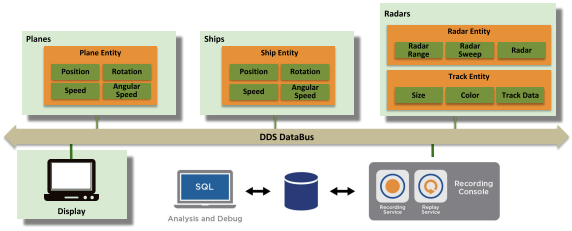


Fig. 6. Deployment of Radar Simulation

This example also demonstrates type extensibility of distributed simulation applications. As a distributed simulation evolves, attributes of simulation interfaces may be added or removed. By leveraging the DDS Type Extensibility, we were able to support extensibility of interfaces between distributed simulations that use different data types.

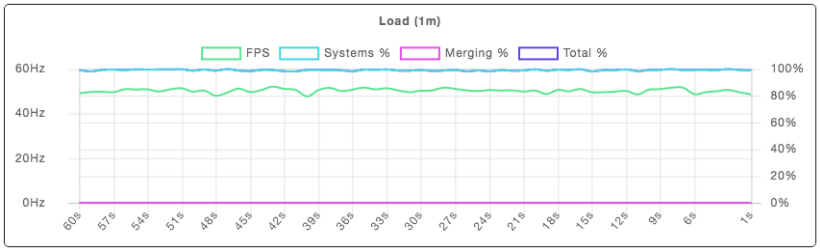
B. Performance Measurements

We measured the throughput of simulations with different types of deployments to understand the performance of the ECS-based framework as well as the overhead of data distributions by DDS. We developed fully integrated local simulation as well as distributed simulation for testing. The test simulations in our experiments create entities and randomly update the positions of the entities at 60 times per second. We conducted performance testing on a MacBook Pro (2.9 GHz Intel Core i5 Dual Core and 16 GB RAM).

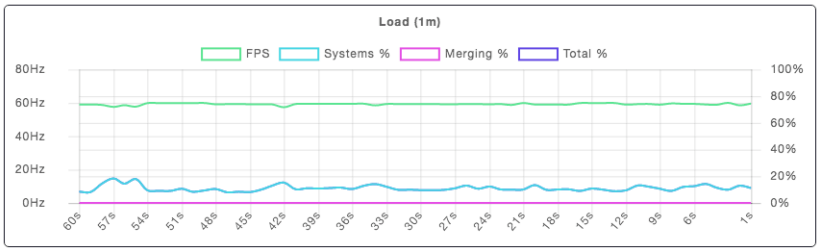
Figure 7 illustrates the performance results for the fully integrated local and distributed with DDS deployment scenarios. The green line indicates the update frequency and the blue line is the system load over time. If a testing simulation reaches its limit, it cannot maintain the configured frequency, 60 frames per second (FPS), and the CPU usage becomes 100%.

Our primary goal of the testing was to understand the scalability of the ECS framework and the integration overhead of data distributions with DDS. As shown in the performance results, the fully integrated simulation is highly performant and it can achieve simulating 250K entities at 50 FPS. Detailed comparative performance results with another ECS framework can be found in this link (https://github.com/SanderMertens/ecs_benchmark).

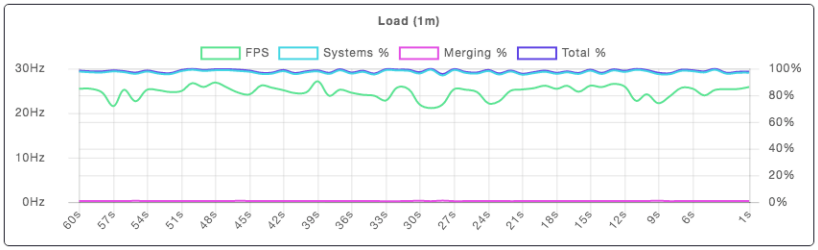
For DDS-based distributed simulations, we measure performance on both reader and writer sides. As shown in the results, the reader side can simulate 10K entities at 25 FPS while the writer side can maintain the configured 60 FPS and the system load was about 10%. This means that the writer side can potentially handle 10 times more entities. The writer side incurs much less overhead because a DDS writer directly used the memory allocated by the ECS framework. We could not apply this optimization on the reader side because while a DDS reader receives data, the order of entities is not necessarily the same as the order in ECS. This requires a lookup per



(a) FPS and System Load of Fully Integrated Local Simulation (250K Entities)



(b) FPS and System Load of DDS-based Distributed Simulation on Writer Side (10K Entities)



(c) FPS and System Load of DDS-based Distributed Simulation on Reader Side (10K Entities)

Fig. 7. Comparing Performance Metrics of Integrated versus Distributed Simulations

entity before insertion, which results in the overhead. In conclusion, the integration overhead mainly caused by copying and serialization between the framework is not trivial and requires further optimizations to reduce the overhead.

V. Conclusions

In this paper we presented a model-driven middleware approach to compose and integrate distributed simulation applications built using ECS and DDS middleware. To ensure ease of use and abstract away inherent and accidental complexities in blending these two middleware, we defined a DSML, which provides users with modular components so that they can build their own simulations by composing them and integrating the simulations via DDS. We validated our approach in the context of representative simulation use cases.

The MIDDECS approach offers several benefits as follows:

- **Modularity:** Components can be independently updated or replaced without affecting the rest of a system.
- **Reusability:** Software is reusable at the component level.
- **Interoperability:** Well-defined ports and standardization ensure interoperability between distributed applications.
- **Extensibility:** A component-based architecture is inherently loosely-coupled, supporting easier extensibility of component and system functionality.
- **Reduced complexity:** Encapsulation, modularity and separation of concerns help to reduce design-time and run-time system complexity.

In the future we would like to extend our DSML to be more inclusive of various data types. Currently we only support four of them. A second improvement would be to the verification process. Currently, our approach verifies based on the pub/sub topic only. The approach could be expanded to include data types, time management policies, among others.

In addition, to simply making the model more mature, there are multiple other avenues to explore. Dynamic changes to the model deployment could be explored. This would be useful in circumventing any problems due to node failure. We could also explore deployment of simulations on the edge using our approach and performing dynamic resource management decisions. Finally, performance can potentially be optimized for the reader side just like we did for the writer side.

Acknowledgments

This work was supported in part by a US DoD project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsor.

References

[1] T. Alatalo, "An Entity-Component Model for Extensible Virtual Worlds," *IEEE Internet Computing*, vol. 15, no. 5, pp. 30–37, 2011.

[2] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel *et al.*, "Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models," in *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, no. 076. Linköping University Electronic Press, 2012, pp. 173–184.

[3] J. S. Dahmann, R. M. Fujimoto, R. M. Weatherly *et al.*, "The Department of Defense High Level Architecture," in *Winter Simulation Conference*. Citeseer, 1997, pp. 142–149.

[4] A. Gokhale, D. C. Schmidt, B. Natarajan, J. Gray, and N. Wang, "Model Driven Middleware," in *Middleware for Communications*, Q. Mahmoud, Ed. New York: Wiley and Sons, 2004, pp. 163–187.

[5] S. Gorecki, Y. Bouanan, J. Ribault, G. Zacharewicz, and N. Perry, "Including Co-simulation in Modeling and Simulation Tool for Supporting Risk Management in Industrial Context," 2018.

[6] D. D. Hodson and J. Millar, "Application of ECS Game Patterns in Military Simulators," in *Proceedings of the International Conference on Scientific Computing (CSC)*. The Steering Committee of The World Congress in Computer Science, Computer, 2018, pp. 14–17.

[7] S. Kelly and J.-P. Tolvanen, *Domain-specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[8] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-specific Languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, 2005.

[9] H. Neema, J. Gohl, Z. Lattmann, J. Sztipanovits, G. Karsai, S. Neema, T. Bapty, J. Batteh, H. Tummescheit, and C. Sureshkumar, "Model-based Integration Platform for FMI Co-simulation and Heterogeneous Simulations of Cyber-Physical Systems," in *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, no. 096. Linköping University Electronic Press, 2014, pp. 235–245.

[10] *Data Distribution Service for Real-time Systems Specification*, 1.2 ed., Object Management Group, Jan. 2007.

[11] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 2003, pp. 200–206.

[12] T. Raffaillac and S. Huot, "Polyphony: Programming Interfaces and Interactions with the Entity-Component-System Model," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. EICS, p. 8, 2019.

[13] D. Wiebusch and M. E. Latoschik, "Decoupling the Entity-Component-System Pattern using Semantic Traits for Reusable Realtime Interactive Systems," in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE, 2015, pp. 25–32.