Abstract—This paper presents a simulation tool integration framework which aims to integrate tools from various research fields in order to provide a seamless design flow for distributed systems with Emergent Self-Organization (ESO). ESO is a decentralized dynamic process with many favorable properties which makes it attractive for large scale distributed systems. However, ESO has a “non-linear” and “random” nature requiring expertise and tools from various research fields. Instead of extending every simulator and modeling framework to contain some of the required tools we propose a tool integration framework consisting of a repository of ESO design tools together with an integration infrastructure. The tools can be immediately used, thus significantly reducing development time and helping to fully explore the potential of ESO. Moreover, these tools can be created by experts on various aspects of ESO and seamlessly used in any relevant application domain. We demonstrate the usefulness of the proposed framework by showing how it can be used to improve an existing self-organizing data collection algorithm for Wireless Sensor Networks.

I. INTRODUCTION

Emergent self-organization (ESO) is a dynamic process in which a system-wide (called macro-level) solution emerges solely based on lower-level interactions (called micro-level) and exchange of only local information [1], [2]. We consider only ESO systems where elements do not have learning capabilities and they follow simple rules created during the design. Such systems exhibit many interesting features which make them suitable for a wide range of distributed systems. Some of the desirable properties are: autonomous adaptation to dynamic changes in the environment without any external help or centralized control, great degree of robustness, and yet they use only simple rules which consume only a small amount of system resources (CPU, battery, memory, network bandwidth etc.). The last property makes ESO even feasible for systems with a lack of resources such as Wireless Sensor Networks (WSNs). There is a great number of applications with ESO proving its feasibility [3], particularly for WSNs [4].

Every ESO system has two essential ingredients [5]: i) The “nonlinearity”, i.e. the whole is greater than the sum of its parts, and ii) “randomness” in the behavior of nodes. These ingredients impose many engineering challenges and require expertise from several fields. The authors in [6] argue that proper engineering of such systems rests on three research areas: (i) Agent-based modeling (domain specific models for structure and behavior), (ii) Dynamical system theory (analysis of the behavior), and (iii) Automation control theory (“tuning” the “non-linear” behavior). Given the broad spectrum of skills required to master all of these fields, application designers naturally are not specialists in all of them. For this reason, as the authors in [7] point out, many designs are done in an ad-hoc manner not fully exploiting the potential of the ESO paradigm in their fields of application.

Some designers even create own tools tailored for particular applications to cope with the design challenge of ESO. Because of tight coupling with other design tools and design environments such as used simulators (WSN or agent based), these tools remains proprietary and very hard to reuse in other applications or different simulators. For this reason, almost every new design with ESO is a reinvention of various tools and adaptation of a simulator with these tools.

In order to solve this inter-disciplinary design problem, we propose to create an integration framework for simulation tools [8] which integrates existing tools from different fields (modeling, simulation, analysis, tuning etc.) into a single seamless design flow. This enables designers of applications to explore the full potential of the ESO paradigm in a straightforward way, using tools from other fields, as well as their domain specific modeling and simulation tools/frameworks without having to modify them. Moreover, designers will save a significiation amount of time by reusing tools instead of writing their own ones. Our approach is contrary to the existing frameworks which lock a designer into a particular agent-modeling simulation framework with a limited set of tools.

The main design goal of the framework was to maximize the usability and the efficiency of the development
process, so that users are not required to be experts in any underlying technology and field [6]. Moreover, the framework was also designed for seamless addition of new tools including simulators. To further speed up the process of algorithm/protocol development, the framework also contains tools for deploying simulators into a grid. Our initial evaluation of the framework is based on several self-organizing applications for WSNs and shows its feasibility. In this paper, we present a case study about improving the design of the self-organizing data collection protocol for wireless sensor networks, called Sodac [9]. During this process, the framework helped to address several design considerations and improved the overall performance of the protocol by a considerable amount without the need to redesign it.

II. RELATED WORK

There are several frameworks which address engineering of ESO behavior such as [10], [11], [12]. All these works provide an own modeling language with a limited set of tools tailored to the used modeling language and simulator. To use such a framework, a designer has to re-model the application with the given framework’s language (input model for the simulator) forcing him to abandon its domain specific design tools and languages which are very efficient in the application domain. For example, [10] can generically model a multi-agent system but cannot realistically describe the communication stack, packet collisions, radio wave propagation, which in turn have significant impact on the designed ESO applications in WSNs. Another category consists of frameworks which are narrowed down to a particular domain (e.g. optimal traffic light coordination) and are quite efficient for that purpose. However, they cannot be used outside of the domain. For example [13] can simulate and design optimal self-organizing traffic lights but cannot be used to design self-organizing communication protocols.

To our knowledge, there is no such integration framework as the one presented in this paper [8] that allows use of various tools from different fields and frameworks under the same design flow, provides a repository of tools, allowing the designer to use its domain models, and at the same time does not require deep expertise in order to use the framework.

III. SODAC: CASE STUDY IN WSN

Sodac [9] is a highly efficient self-organizing multi-path protocol for routing and data collection in resource-constrained wireless sensor networks. Its operation is divided into two phases: discovery and data routing phases. During the first, the discovery phase, a routing topology on top of the WSN is established, while, during the second phase, the topology is used for routing and continuous update.

Figure 1 illustrates the route discovery phase. In the figure, the wireless links between nodes are represented by solid (black) lines. Subfigure (a) shows the forward route discovery. Forward discovery packets/requests are depicted using dotted (green) arrows. (b) and (c) depict the backward discovery/confirmation, with solid (brown) arrows representing backward discovery packets. During this stage, the link failure between nodes A and C fails. However, since Sodac builds a dense network of routes during the forward discovery, other path alternatives (see Figure 1c) are used in order to establish/confirm routes between all nodes in the network and node A.

There is a myriad of applications for wireless sensor networks requiring a protocol with properties as outlined above [14]. We considered the following two during the development of Sodac [9]: First, in a space scenario, sensor nodes are dropped into the atmosphere of a planet. During their flight towards the surface of the planet, the sensors sense data, which is routed using Sodac towards one data collection node that possesses the capability to transmit it to
its final destination. Second, in the aviation scenario, sensors are integrated into the surface of an aircraft body. During the flight of the aircraft, the sensors sense data needed for preventive maintenance, like temperature and vibration. After the landing of the aircraft, the Sodac protocol creates a routing topology on top of the sensor network and collects the data towards a data collection node. Both scenarios consist of networks of hundreds of nodes. However, this does not imply a limitation on a certain number of nodes: Sodac is capable of scaling beyond hundreds of nodes.

The main design objective for Sodac (and most of WSNs applications) is to minimize sensors’ energy consumption by minimizing the amount of communication. In Sodac, this is achieved by a trade-off between creating and choosing short routes as well as balancing the load incurred by communication evenly over the network. This trade-off is crucial to the considered scenario since transmitting a single bit consumes approximately the same amount of energy as executing hundreds or thousands of instructions [14]. At the same time, load balancing is aimed at decreasing the probability of overburdening specific nodes, leading to their disfunctionality and all other consequences associated with this, such as an increase in communication costs.

The amount of communication in Sodac is modeled in Equations (1) and (2) as Total Number of Transferred Bits (TNTB).

\[
TNTB = \sum_{i} ntb(i) \quad (1)
\]

\[
ntb(i) = Tx_i + Rx_i + TxC_i + RxC_i \quad (2)
\]

TNTB is an aggregated value of nodes’ Number of Transferred Bits (NTB) in the simulated network of size N nodes. NTB itself is defined as sum of number of successfully transmitted \( Tx \) and received \( Rx \) bits plus number of bits which were unsuccessfully transmitted \( TxC \) or received \( RxC \) due to collisions. When a collision occurs, a message has to be retransmitted. Thus collisions drain additional energy from the battery, shortening lifetime of a node. Generally speaking, to minimize TNTB (and in turn maximize the lifetime of the network) there have to occur less collisions and overall number of bits in successfully transmitted messages should be small.

During the design of Sodac we have identified 11 micro-level parameters that impact the main design objective, i.e. minimizing the amount of transmitted bits in the whole network. Their relation to the overall performance is not clear, supporting the observation by [5], which identify the previously mentioned two essential ingredients of the ESO paradigm: non-linearity and randomness. Moreover, in the space scenario, the topology deliberately changes (based on a realistic movement pattern), which translates to changing link properties. Every simulation run with the same setup of parameters but a different random number generator seed yields different results. Since the optimal values for the 11 parameters are unknown, there is an immense number of combinations (\( \subseteq R^{11} \)), each which has to be executed a number of times producing a sufficient amount of statistical significance. Obviously, such an approach is hardly feasible.

In order to obtain highly realistic simulation results, while minimizing the amount of development time, the ShoX Wireless Network Simulator [15] was used. Since ShoX (like any other WSN simulator) neither contains advanced tools for analysis of the behavior by applying dynamical system theory nor “smart” parameter search, during the tuning phase of Sodac, the values of the 11 parameters were initially selected empirically but based on a realistic reasoning of experienced protocol developers. Simulations with given empiric values yielded on average better result than a reference data-collection protocol [9]. However, the full potential of Sodac could not be exploited because the concrete contribution of each parameter to the overall (macro-level) performance was unclear. Due to space limitations, we present only three of the parameters and questions related to their influence on the system-wide (macro-level) behavior in the following.

Q1: What should be the length of the history field, which is included in each packet? The trade-off is that while the history field costs a considerable amount of transmission energy, it could help nodes to make better routing decisions by avoiding duplicity, avoiding loops and thus saving energy as well.

Q2 and Q3: How long should be the randomization window for sending i) route discovery and ii) data packets? Setting the window too short will cause more packet collisions, since the nodes in the network are not synchronized and requests for data are received at approximately the same time. A larger window, on the other hand, translates to fewer transmission collisions but it prolongs the data collection time, if it is too long. To obtain answers for these questions and finding out optimal values for parameters, we used the devised framework.

IV. Framework Architecture

The underlying integration principle behind the proposed framework [8] is the following: Improving the behavior of the ESO system can be rigorously represented as a stochastic optimization problem where we systematically select values of parameters which will eventually lead to the maximization or minimization of the given cost function called fitness. The fitness function can be compound from several so called Macroscopic Variables (MVs) which represent system-wide (macro-level) properties of the designed ESO system, e.g. number of nodes in a given state, average node energy consumption, average lifetime of the WSN, average link quality. To obtain values of MVs, we have to perform number of simulations with a particular domain specific simulator. The tool for “sensing” values of MVs is called
the Probe. Due to the embedded randomness in systems with ESO behavior, the fitness function is actually a random variable which can be statistically described. In order to do that, we have to perform a number of simulations so we will get a statistically significant number of realizations of the fitness function. Based on the observed random realizations of the fitness function, we can derive an estimate of its value, like average value, confidence interval, etc.

The overall architecture of the framework is shown in Figure 2. In order to meet the main requirements on the framework (e.g. ease of use for the application designer, seamless extensibility for new tools, platform independence), we combined two enterprise software architectural styles: Service-Oriented Architecture (SOA) and Message Bus (MB). The basic idea is that each tool exposes (provides) and can also consume a certain functionality which is made available to other tools via MB (shown in italics). We have identified five different types of tools. The Optimizer selects new values for the MVs and subsequently receives fitness estimations of them. The Simulator tool is conveying simulations of the ESO application which are “observed” by the Probe tool. The Probe is then forwarding raw simulation data to the Analyzer which in the end derives the fitness estimation. The last tool, the FitnessProvider, provides split and merge of the control flow and is also a placeholder for various tools such as visualizations, graph generation, as well as several common interfaces indicating the direction of the control flow. The most significant benefits of combining these two architectural styles are: extensibility (several tools can be attached to the same bus), the functional behavior of the bus does not need to have knowledge about other tools, their implementation, and location), low complexity (the tool has to provide only one connector to the bus instead of having individual connectors to almost every tool it can potentially need). This combined architectural style also satisfies the main design goal of the integration framework, which is to offer ease of use and straightforward extendability by new tools. With such an architecture, it is simple to let various tools from different domains to cooperate. The application designer just needs to select the right tools and attach them to the proper Message Bus. Plugging an existing tool into the framework requires the writing of a plugin or an interface adapter. Due to the used design patterns, judging based on our experience, this is not a difficult task.

In order to simplify the use of the integration framework even further, we applied several design patterns. The most important pattern is the Dependency Injection (DI) which is used to initialize and to wire up design tools to the proper message buses at runtime. DI also contributes to the high internal cohesion of tools because they only contain code related to their functionality and do not need to care about the resolution of other services (tools). To provide full platform independence, the framework was implemented in Java.

In practice, the framework is implemented as a collection of libraries (Figure 3). The central library is called Core and contains essential infrastructure of the framework (definition of tool interfaces, message buses, abstract data classes, grid computation extensions, etc.), set of graphical modules for visualization and graph generation, as well as several common tools for simple analysis and stochastic optimization. All other libraries (called extensions) represent collections of tools provided by various tool designers. For example, in Figure 3 are shown libraries which extend the framework with plugins for simulators such as ShoX, ns-2, and MASON [12]. On the left, there are extensions for tuning the design with genetic algorithms, Particle Swarm Optimization, etc.

V. FRAMEWORK SETUP FOR SODAC

In the course of improving Sodac, we used several tools, as shown in Figure 4. From the set of stochastic optimization tools, we initially selected the Particle Swarm Optimization (PSO). The reason why PSO was chosen among other available tools (e.g. genetic algorithms, ant colony optimization) is beyond the scope of this paper. The PSO tool minimizes the objective function which was earlier defined (Sec. III) as the Total Number of Transferred Bits (TNTB). The PSO was configured to work inside the eleven-dimensional search-space where each dimension represents a parameter of Sodac. Each particle then represents a set of values for the parameters. According to the questions in Section III, we let

![Figure 2. Simplified top-level architecture of the framework. Message Buses are in italics.](image)

![Figure 3. Organization of the framework into the Core and Extension libraries. Arrows denote an extension.](image)
the particles move only within the three dimensions while the remaining eight had empirically chosen values. To get a statistically sound fitness estimation, we ran 50 simulations and used a statistical tool to aggregate the obtained fitness function into a confidence interval of \( \alpha = 0.05 \). The fitness function was defined to consist only of one macroscopic variable—Total Number of Transferred Bits (TNTB).

To be able to simulate Sodac, we used a tool called ShoX Runner which wraps ShoX simulator and provides connectors to the relevant Message Bus. The Fitness Graph and Macro Value Graph are practical graph tools for visualizing fitness and macroscopic values as functions of selected parameters. Another tool, Cache, avoids performing double fitness estimation by remembering the history of fitness estimations. When a new request contains parameters which were already evaluated in the past, the Cache immediately returned the corresponding estimation of the fitness function. The Cache can be also persistent and stores simulated data in a file or a database. To speed-up the process of tuning, we could select between two currently available middleware plugins for grid computations which can be used interchangeably: JPPF and SlimGrid. We selected SlimGrid over JPPF because we needed to coordinate several nodes behind the firewall. The SlimGrid was built on top of the open-source project for distributed coordination, Apache ZooKeeper. The middleware plugin extends the Simulation and Analyzer Message Buses across the Internet so the tools running remotely will look like they are attached to the same local bus (i.e., providing location and access transparency).

To practically make use of the framework, we had to setup a Java project called configuration project which in our case is SodacTuning in Figure 5. In the project, the above mentioned tools are selected and configured including their interconnection into a tool flow shown in Figure 4. The configuration and interconnection of tools is specified in the XML-file. The application (i.e., the code to perform tuning of Sodac) activates the simulations by calling the corresponding function in the Core library passing the configuration file as parameter. Based on the file, the framework automatically launches all tools, configures them, sets interconnections and triggers the execution. Full implementation of the data-collection algorithm is in the Sodac package which has the dependency to the ShoX simulator used to simulate it. In order to use ShoX, a plugin from the ShoX Extension library is used. Particle Swarm Optimization tools were taken from the PSO Extension library.

VI. RESULTS

To obtain answers to the questions in the Sec. III we used a grid to run simulations. The grid was formed from nearly 40 computers located in three countries (Norway, Germany, USA). To use the power of the grid, beside the PSO, we also created a landscape of all three parameters’ combinations shown in Figure 6. The grid had to perform around 50.000 simulations and each point represents aggregated information from 50 simulations. Layers in the figure correspond to three different sizes of the packet history field. The fitness function TNTB is on the Z axis where we show only the center value from the confidence interval. The cross in the graph shows the reference performance of the original Sodac [9] (approx. 59MBit).

Answers to all three questions from Sec. III can be easily read from the landscape figure. A1: The history field does...
not bring any expected benefit at all. Zero history field has in all cases about 5-8% better performance than 1 hop memory. A2 and A3: The minimum is visible in the upper-right corner and is about 11-15% more optimal than original parameters’ setting. We can also see a region forming a global maximum which yields the worst TNTB.

If we tried to obtain answers for the given questions without employing the proposed framework, we would have to design and implement all used tools from scratch. To design, implement, and test tools for distributed simulations take itself considerable large amount of time. By using off-the-shelf tools in the framework, we saved that tool development time and obtained answers for questions which were not trivial to solve. The time needed to configure the framework for the desired flow of tools is much shorter than developing a single tool.

VII. CONCLUSIONS

In this paper we have presented a framework for integrating tools needed in the design of distributed systems with Emergent Self-Organizing (ESO) behavior [8]. The presented framework provides an infrastructure allowing the needed tools to be seamlessly used in a single tool chain, significantly reducing the development time of an application by: i) Reuse of integrated tools which otherwise would have to be created from scratch including the grid extension, and ii) reused tools that can provide a “smart” search for stochastic optimization instead of the commonly used empiric approach. The framework also allows the design of an application in various domain specific languages which is an important feature in several domains (e.g. WSNs). We demonstrate the usefulness of the framework employing the example of improving the performance of the self-organizing data collection protocol Sodac. This requires determining the values of a set of configuration parameters yielding a near-optimal behavior of the protocol. The future work on the framework includes integrating more tools, e.g. stochastic optimization, volunteer grid computing like BOINC, as well as simulators from different application domains (e.g. multi-agent systems). The goal is to make ESO systems easier to design, thus making them available for a much wider community of application designers.

REFERENCES


