Performance Summary of Ensemble-MD Patterns

Nikhil Shenoy

March 17, 2016

Abstract

Modern software, no matter its application, always incorporates performance as a design factor. Without efficient algorithms and implementations, the software becomes less attractive to the user because it does not accomplish the user's tasks in a timely fashion. Such factors are seriously considered when designing software for scientific computing, as many simulations in areas such as molecular dynamics require efficient tools to process the ever-increasing amount of data. In this article, we discuss the EnsembleMD-Toolkit and benchmark the efficiency of two standard Patterns using a sample workload. We assume that the reader has a basic familiarity with the Toolkit, and we will define only parameters relevant to the performed experiments.

1 Introduction

Traditional distributed systems utilize batch queueing systems in order to schedule jobs to a high performance machine's resources. In such systems, scripts are written to execute several different tasks sequentially, and a scheduler assigns the tasks to the resources it requires. The problem with this approach is that since a task will most likely require only a fraction of the available resources, the remaining resources will stay idle. Leaving cores unutilized severely limits the efficiency and throughput of the system and must be avoided. Additionally, the total waiting time in the queue for a simulation can scale quickly in the number of component tasks, which can significantly increase the time to completion. These problems burden the user who, in many cases, is attempting to run computation-heavy scientific simulations.

To alleviate this problem, the concept of Pilot-Jobs was introduced. A pilot job is a special type of container designed to process and manage several different tasks during its lifetime. In a pilot scheme, information about each task of a simulation is associated with the pilot, and only the pilot is placed into the scheduling queue. Once the scheduler schedules the pilot, it then establishes communication with the user script and allows the user to directly schedule tasks to it for execution, provided that enough resources are available. The main advantage in using a pilot system is that one can circumvent the scheduler when running multiple tasks. By placing only the pilot into the scheduling queue, the user obviates the time necessary to request and assign resources for the total number of tasks. This can result in a decreased time-to-completion for each task, and high throughput of the number of tasks. More importantly, such a scheme leads to efficient usage of resources, as not only can a single pilot keep its assigned resources occupied for long periods of time, but a series of pilots can ensures that a large percentage of the entire grid's resources are kept busy. Since the submission of the pilot and its workload is decoupled from the assignment of resources, complex applications can be written to take advantage of all the resources in the system.

Based on these advantages, the RADICAL-Cybertools group implemented RADICAL-Pilot, which is a Python API that provides user-friendly abstractions for the pilot model. By abstracting the pilot paradigm, one can use RADICAL-Pilot as a foundation for more complex use cases that require high performance machines. The EnsembleMD-Toolkit, an API designed for large scale molecular dynamics simulations, is a prime example of this, as it utilizes RADICAL-Pilot to implement simulations that are popular within the molecular dynamics community.

1.1 Ensemble-MD Toolkit

In this article, we present the EnsembleMD Toolkit (EnMD) as one of the complex applications built on top of the RADICAL-Pilot architecture, and provide a brief overview of the software here. The execution flow of the Toolkit is centered around three main components; the Application Pattern, the Execution Context, and the Kernel Plugin.

The first component that the user interacts with, the Application Pattern, is a general template for executing tasks. These patterns present the user with high-level descriptions of control flow which are task-independent. The user need only choose a pattern and provide it with the details necessary for the simulation, and the API takes care of the rest. By doing this, the details of executing the simulation are abstracted from the user, providing an interface that makes translating biological experiments into computer simulations very simple. For example, the Pipeline Application Pattern can run a collection of tasks sequentially as a series of steps. In this scheme, the pattern will wait for each step to finish before moving on to the next step. The pattern also allows for several instances of a given pipeline, as shown in Figure 1, if redundancy is required. Other available patterns include Simulation-Analysis and the Replica Exchange.

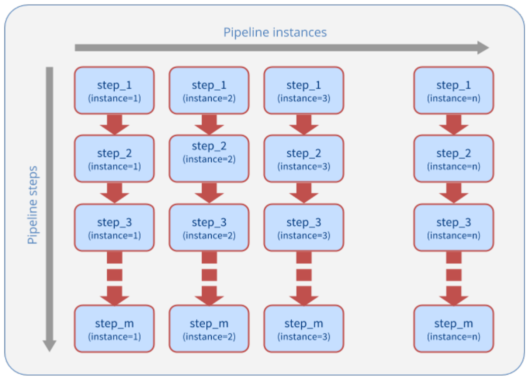

Figure 1: Block Diagram of the Pipeline Pattern

The Execution Context represents the distributed computing resource that the simulation is running on. It contains the infrastructure needed to interface with the machine itself, and is the agent that allows the Toolkit to adapt to and include a variety of resources. Its main functions are to allocate and deallocate the resource, as well as run an execution pattern on the resource. Aside from specifying which resource to use, the user does not interact with this component directly as it is meant to be behind the wall of abstraction.

Finally, the Kernel Plugin represents the scientific tool to be used. This can be anything from molecular dynamics executables such as Amber to simple tasks such as counting the number of characters in a file. The Toolkit provides a list of built-in kernels, but provides a mechanism for users to add their own. The Kernel Plugin abstracts aspects of the tools that are specific to a particular resource and presents a uniform interface with the user can define tasks.

2 Experiments

2.1 General Parameters

These performance experiments were designed to analyze the Pipeline and Simulation Analysis Patterns. Using these Patterns, we execute a two-stage workload; the first stage consists of creating a 10Mb file of random characters, and the second stage performs a character count on this file. In Table 1, we have included the commands used to execute each stage as well as the average execution time from the Bash Shell. We also use the environment parameters specified in Table 2.

Kernel	Bash Command	Avg. Execution (seconds)
misc.mkfile	base64 /dev/urandom | head -c 100 > test.txt	.008
misc.ccount	grep -o . test.txt | sort | uniq > out.txt	.004

Table 1: Kernels, their commands, and their expected execution times.

Parameter	Value
EnsembleMD-Toolkit version	0.3.14-27-g65bc062
EnsembleMD Branch	devel
RADICAL-Pilot version	0.40.1
Target Machine	XSEDE Stampede

Table 2: Environment Parameters.

2.2 Types of Scaling

For these experiments, we measured performance as a function of the number of cores allocated to a script as well as the number of instances of the Pattern being executed. We initially implemented weak scaling, in which the number of cores scales at the same rate as the number of instances, with the goal of observing a relatively constant execution time. The intuition behind this is that the core to instance ratio stays the same, which implies that the work per core/instance combination is the same. We also perform strong scaling, in which we hold the number of instances constant while we scale the number of cores. In this case, we expect that the Pattern will make use of the additional cores to finish the task more quickly. The scale that we use for these experiments is [1,16,32,64,128].

In each script, we measured the time taken to complete various stages of execution. These will be elaborated upon in the subsection for each pattern.

2.3 Pipeline

For the Pipeline pattern, we measured the parameters defined in Table 3.

Parameter	Definition
EnMD Core Overhead	(alloc_stop-alloc_start) + (dealloc_stop-dealloc_start)
EnMD Pattern Overhead	((step1_wait-step1_start) + (step1_stop-step1_res)) + (step2_wait-step2_start) + (step2_stop-step2_res))
RP Overhead	((step1_wait-step1_res) - step1_data_movement - step1_execution_time) + ((step2_wait-step2_res) - step2_data_movement - step2_execution_time)
Step 1 Execution Time	PendingAgentOutputStaging - Executing
Step 2 Execution Time	PendingAgentOutputStaging - Executing
Data Movement Time	((step1_Done - step1_PendingAgentOutputStaging) + (step1_Allocating - step1_StagingInput)) + ((step2_Done - step2_PendingAgentOutputStaging) + (step2_Allocating - step2_StagingInput))

Table 3: Pipeline Definitions

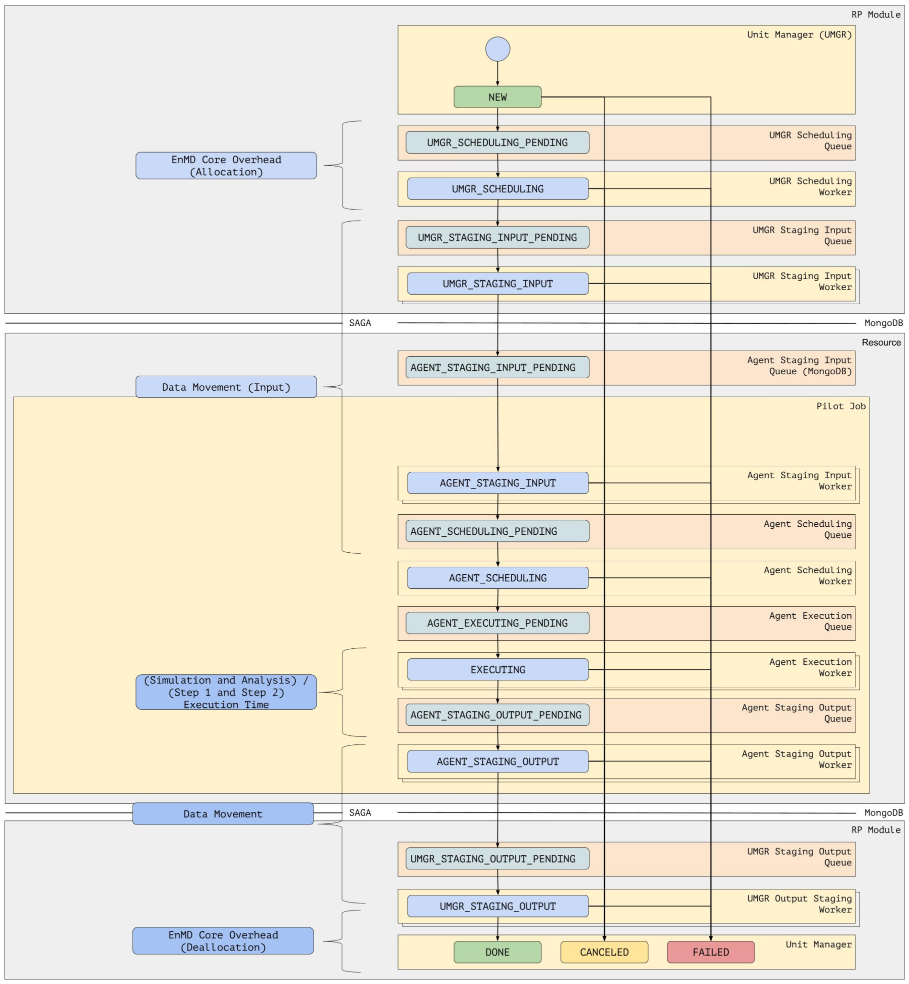

Figure 2: Mapping of parameters to pilot state model

We also map the recorded parameters to the global pilot state model in Figure 2. This mapping holds for both the Pipeline and Simulation Analysis patterns, and shows how the measured components fit into the pilot paradigm.

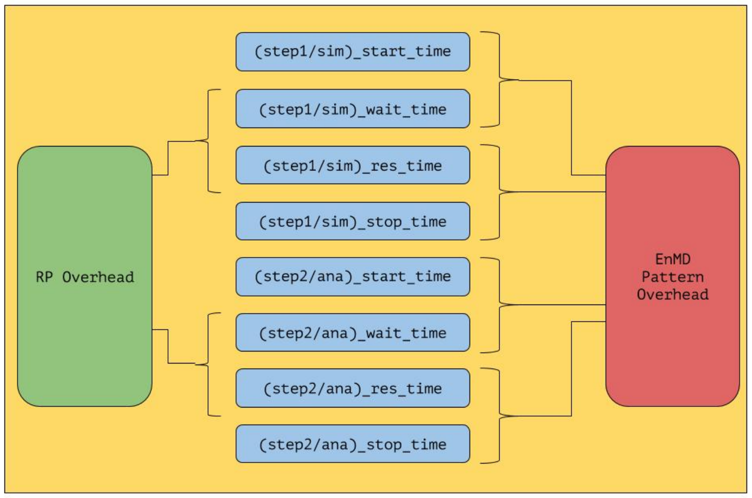

Figure 3: EnMD and RP Overheads

Figures 3 details the measurements taken within the Execution Time state shown in Figure 2. The start and stop times indicate when execution starts and ends for that particular stage. The wait time indicates when the stage submits all of its tasks and waits for them to finish. Finally, the res time is defined as the point when all tasks of that stage have finished execution and the EnMD Toolkit resumes execution to prepare for the next stage.

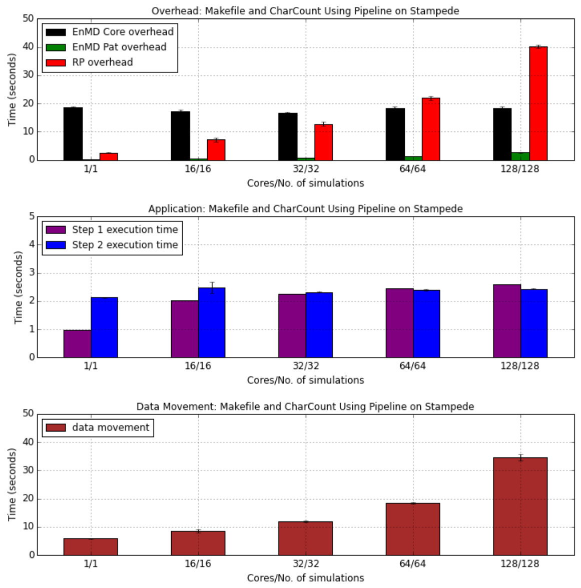

Figure 4: Weak Scaling with the Pipeline Pattern

Figure 4 shows the scaling behavior of each of the parameters. In the first graph, we observe that the EnsembleMD Core overhead stays relatively constant at about 18 seconds throughout the scaling. We also see that the EnsembleMD Pattern Overhead is at most 3 seconds, which is a fraction of the Core overhead. Finally, we see that the RADICAL-Pilot overhead increases rapidly as the scale increases. Ratios between RP overheads from a configuration to the previous configuration yield a value of about 1.8, which implies that the overhead is growing linearly. In the second graph of Figure 1, we see that the execution times for both Step 1 and Step 2, which were the Makefile and Character Count respectively, stay very much constant throughout the scaling, except for the anomaly of Step 1 in the first configuration. This is to be expected, as the settings in the configuration should have no effect on how long it takes to run the underlying Bash commands. However, the times shown are much larger than those observed when the commands are executed directly from a Bash prompt. It is possible that the extra time is due to overhead from a combination of EnsembleMD and RADICAL-Pilot, but we have not dissected this additional component. Finally, the data movement shows a steady increase in duration. The ratios from one configuration to the previous configuration fluctuate from 1.6 to 1.9, but generally imply a linear increase as a function of the number of cores and execution instances.

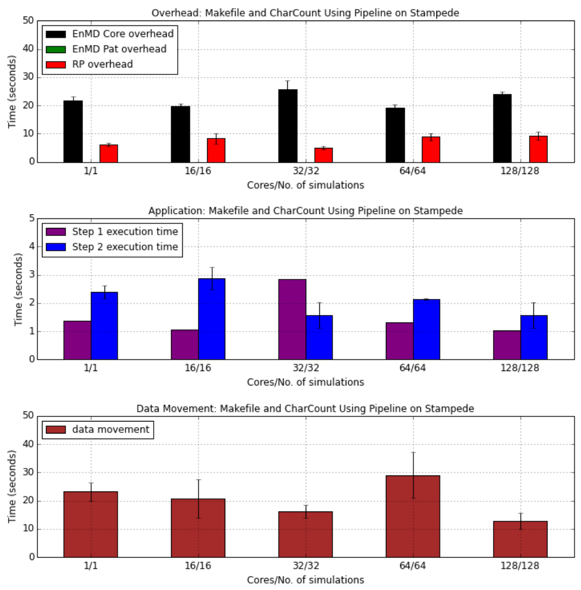

Figure 5: Strong Scaling with the Pipeline Pattern

Figure 5 displays the strong scaling results from the Pipeline. Looking at the EnsembleMD Core Overhead, we find that it averages to around 22 seconds. On the other hand, the Pattern Overhead was measured at 15ms on average, so the values at each configuration did not register on the scale of the Core Overhead. The RP overhead remains approximately constant at around 8 seconds. The execution times for Steps 1 and 2 show a much different trend than they did in the weak scaling experiments; in general, Step 1 showed execution times that were closer to the actual execution time, but were still three orders of magnitude greater. Step 2 fluctuated much more than it did during the weak scaling experiments. Data movement shows a general decline in duration, aside from the anomaly with 64 cores. This may be due to the increased number of cores available for the movement.

2.4 Simulation Analysis

For Simulation Analysis, we considered the definitions in Table 4.

Parameter	Definition
EnMD Core Overhead	(alloc_stop-alloc_start) + (dealloc_stop-dealloc_start)
EnMD Pattern Overhead	((sim_wait-sim_start) + (sim_stop-sim_res)) + ((ana_wait-ana_start) + (ana_stop-ana_res))
RP Overhead	((sim_wait-sim_res) - sim_data_movement - sim_execution_time) + ((ana_wait-ana_res) - ana_data_movement - ana_execution_time)
Simulation Execution Time	PendingAgentOutputStaging - Executing
Analysis Execution Time	PendingAgentOutputStaging - Executing
Data Movement Time	((sim_Done - sim_PendingAgentOutputStaging) + (sim_Allocating - sim_StagingInput)) + ((ana_Done - ana_PendingAgentOutputStaging) + (ana_Allocating - ana_StagingInput))

Table 4: Simulation Analysis Definitions

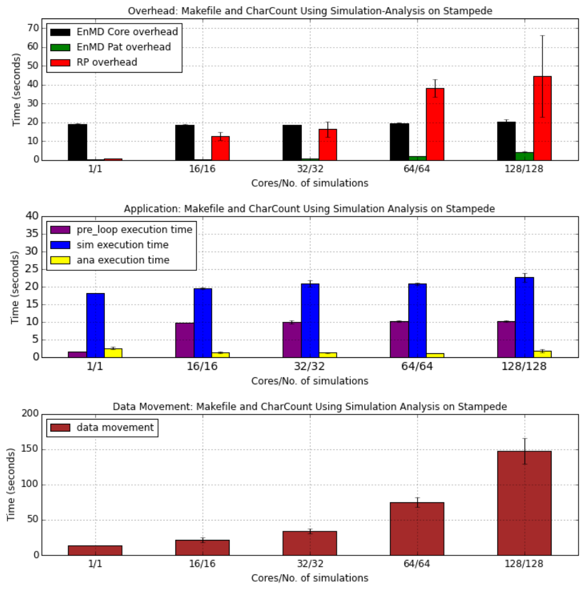

Figure 6: Weak Scaling with the Simulation Analysis Pattern

In Figure 6, we see scaling behavior similar to what we saw in the original pipeline weak scaling experiment. The EnsembleMD Core Overhead is essentially constant, the Pattern Overhead is small in comparison and is capped at around 5 seconds, and the RP overhead does show an increase across the combinations. However, the RADICAL-Pilot overhead does not show a distinct linear progression in the same fashion as the Pipeline Weak Scaling plot did. The second plot shows the phases of execution of the experiment. The pre-loop phase contained no logic, so our inference is that the times shown are the overhead introduced to make the call to the pre-loop. The Simulation execution time, which contained the Makefile Kernel, was constant throughout the scaling, but took longer on average than it did for the Pipeline. The Character Count, encapsulated by the Analysis stage, was truer to the values recorded in the Pipeline Weak Scaling plot. The Data movement plot shows a similar increase in the time needed to download the output data, but the trend does not seem to be linear.

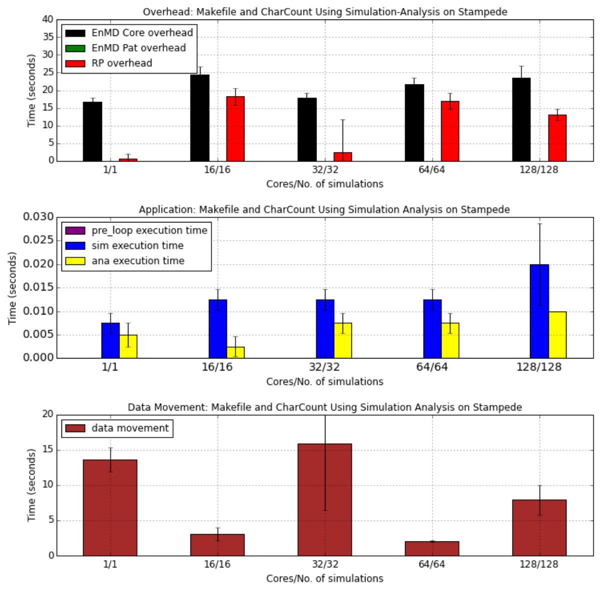

Figure 7: Strong Scaling with the Simulation Analysis Pattern

Finally, we examine the behaviors of each measurement during the Strong Scaling experiments with Simulation Analysis in Figure 7. EnsembleMD Core Overhead wavers around 20 seconds, whereas the RADICAL-Pilot Overhead has a large amount of variation in its time. Again, the EnsembleMD Pattern Overhead averages to approximately 20ms, and thus are not be visible on the same graph. In the Simulation phase, we find that the execution time has been reduced dramatically, and that a representative value for it could be 13ms. The Analysis phase also varies slightly, but can be approximated at 7ms. The Data Movement durations are troubling, seeing as there is not a clear pattern to how the different values were obtained. The scaling does not seem to have had a visible effect on the duration of this phase.

3 Conclusion

This set of experiments were conducted to further understand the EnsembleMD Toolkit's scaling behavior. Based on the data from our experiments, we were able to identify trends that hinted at the Toolkit's capabilities for different core-task configurations. We considered stages of operation within the Toolkit that best exemplified its functionality; the Core Overhead, the Pattern Overhead, the RADICAL-Pilot Overhead, the Task Execution Time, and the Data Movement. While not all of these parameters showed distinct trends, we were able to glean information from a few of the parameters.

The Task Execution time, for both the Makefile and Character Count steps, were found to be relatively constant across most configurations. This behavior is expected, as the time required to execute a bash shell command should only change if different input is given to it. In our case, the kernels in each step are doing the exact same tasks in every configuration, so a change in the execution time would be troubling. Even though the execution time was constant, we noticed that it was much higher than the time the same tasks took when executed from the Bash command prompt. This indicates that there are additional RADICAL-Pilot and EnsembleMD wrappings around the Bash commands that increase the duration. While dissecting that overhead was not the goal of this experiment, wit is a candidate for future work. Throughout all the experiments performed, we observed that the EnsembleMD Core Overhead stayed relatively constant over all configurations. This implies that users and developers can rely on the performance of the EnMD Core staying constant regardless of the tasks the user gives it.

In almost all measurements of the EnMD Pattern Overhead, we found that its duration was on the order of milliseconds. This is useful to users because they can be assured that any preparation done by EnMD to before executing the task is very efficient and is not expected to hinder the completion of the task. For developers, this indicates that the Toolkit makes quick transitions between states in the flow of execution. RP Overhead seems to scale in some cases, stay constant in other cases, and fluctuate in others. Because of this, we cannot extract a useful pattern and characterize RP's behavior at scale solely based on these experiments. As a majority of RP was implemented before EnMD, we would expect its performance at scale to have been worked on extensively and made reliable. We would need more experiments to properly pin down the behavior, as knowledge of how efficiently the pilot functions is crucial to the application's execution.

In these experiments, we measured data movement as the time needed to download the output of our tasks from the remote machine to the local machine, seeing as all the input data was being generated on the remote machine. Both of the weak scaling cases showed that the time required for the data movement was directly proportional to the core/instance configuration. The Pipeline case seemed to imply a linear relationship between the variables, whlie the Simulation-Analysis case implied a more quadratic relationship. We have not investigated the relationship in detail, but determining such relationships would allow the user to restructure his application and explore the tradeoffs between effiency and data usage. Unfortunately, we were not able to notice distinct patterns in the strong scaling cases. We do believe that relationships similar to the ones suggested by the weak scaling experiments do exist, but further testing would be required to find them.

4 Future Work

Performance testing is an ongoing body of work in any scientific experiment, and the results presented here can be extended into many different directions. First, one could examine the relationship between the time taken to process varying sizes of data and the number of cores given to process that data. Such information would be useful in gauging EnsembleMD's capability to handle large and small amounts of data. The same experiments could be run on all the high performance machines supported by the Toolkit, with the expectation that the user gets similar performance no matter which machine he uses. Next, one could consider the Toolkit's role in data movement; in these experiments, we considered only the Toolkit's involvement in transfers between the local and remote machine. An interesting avenue to explore would be the duration of all the operations and changes that the Toolkit makes to the data while it is on the remote node but not being processed. This, combined with this article's measurment of local/remote transfers, may better reflect the overhead incurred by data movement. Finally, discovery of best-fit functions for the important components of the Toolkit, such as the Pattern Overhead, RP Overhead, and Data Movement, would characterize the Toolkit in an easy-to-use way. The benefit of using a simple function to approximate the behavior of a Toolkit component would provide the user with instant information on how to structure his application, leading to increased usability.