Ensuring Deployment Predictability of Distributed Real-time and Embedded Systems

Gan Deng, Douglas C. Schmidt, Aniruddha Gokhale
Dept. of EECS, Vanderbilt University, Nashville, TN
{dengg,schmidt,gokhale}@dre.vanderbilt.edu

Abstract

The dynamic deployment and configuration (D&C) of components in response to environmental changes or system mission mode changes is essential to facilitate runtime resource allocation for component-based distributed real-time and embedded (DRE) systems. This paper provides several contributions to the study of predictable D&C for component-based DRE systems. First, we describe how the predictability of component-based D&C can be affected by application dependency relationships and priorities. Second, we describe how a multi-graph algorithm called partial priority inheritance via graph recomposition *(PARIGE) can improve D&C predictability. Third, we empirically evaluate the effectiveness of PARIGE on a representative DRE system based on NASA Earth Science Enterprise's Magnetospheric Multi-Scale (MMS) mission system. The results show that PARIGE can avoid unbounded deployment time priority inversion when component assemblies with different priorities have complex dependencies among each other, thereby significantly improving the responsiveness of mission-critical tasks with higher priorities.*

Keywords: Component middleware, Distributed Real-time and Embedded systems, Deployment and Configuration.

1. Introduction

Emerging trends and challenges. Developing distributed real-time and embedded (DRE) systems whose quality of service (QoS) can be assured even in the face of changes in available resources or QoS requirements is an important and challenging R&D problem. Systems with such characteristics are called *open* DRE systems [1] since they operate in open environment and must be prepared to accommodate changing operating conditions or requirements, such as power levels, CPU/network bandwidth or mission modes. Examples of open DRE systems include shipboard computing environment [2], and intelligence, surveillance and reconnaissance systems +[3].

Open DRE system are often large and complex, *e.g.*, a shipboard computing system may consist of thousands of software components that run a wide range of missions, such as ship navigation, ship structural health monitoring, vision-based object tracking and object characterization. To manage the overall complexity of such systems, the missions are often decomposed into many domain-related tasks that can be modeled as *operational strings* [4], which are assemblies of software components that capture the partial order and workflow of a set of executing software capabilities for particular domain tasks.

Operational strings have the following properties that make them useful building blocks for open DRE systems:

- **Distributable**, *i.e.*, operational strings can be deployed onto multiple nodes of the target running environment, and different components in operational strings can communicate remotely with each other.
- **Concurrent**, *i.e.*, operational strings can run concurrently in the same target environment and share many system resources, such as CPU, memory, and network bandwidth.
- **On-demand**, *i.e.*, operational strings can be dynamically populated at system runtime and then deployed into the target running environment on-demand to accommodate changing mission goals.
- **Cooperative**, *i.e.*, to achieve certain mission goals different operational strings can cooperate with each other through their *ports*, which delegate to the ports of monolithic components that consist of the operational strings.
- **Prioritized**, *i.e.*, different operational strings can be assigned with different priorities by system architects or online planners to reflect their importance to certain mission tasks or to the overall system.

The dynamic nature of open DRE systems requires the deployment and configuration (D&C) of scores of operational strings at runtime to ensure that executing systems keep in sync with changing mission goals and resource availability. The runtime management of operational strings in DRE systems is hard since the D&C framework must be scalable and predicable. It is therefore essential that D&C frameworks be able to dynamically deploy and configure operational strings in a timely and predictable manner.

In complex DRE systems, many operational strings may be deployed dynamically, *e.g.*, in response to mission mode or environmental changes. If dependencies exist among these operational strings, *deployment priority inversions* can occur at runtime. A deployment priority inversion occurs when a higher priority operational string cannot be deployed before lower priority operational string(s) because of the dependencies between them. Existing D&C frameworks [5, 6, 7] only consider the dependency between operational strings and ignore their priorities, which can cause unbounded deployment priority inversions for DRE systems.

Solution Approach → Partial Priority Inheritance via Graph Recomposition.

To address the challenges of open DRE systems described above, we developed a technique based on an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm analyzes the dependencies between operational strings and removes all the dependencies from higher priority operational strings to lower priority ones by promoting[1] components from lower priority operational strings to higher priority ones. By applying our technique, a D&C framework can avoid potential priority inversions when multiple operational strings are deployed at runtime.

Figure 1 shows the three three main steps of our approach:

- Step 1 converts a deployment descriptor (which contains metadata describing a set of operational strings) into an in-memory *directed graph* representation. Each vertex in the graph represents a component in the operational string and each edge represents a connection between two components.
- Since a deployment plan may have multiple operational strings with different priorities having dependencies among each other, step 2 analyzes the dependency relationship between all the operational strings by perform a graph-based algorithm called *partial priority inheritance via graph recomposition*. This algorithm removes all the priority inverted dependencies between operational strings by promoting component(s) from the lower priority operational string to the higher priority string.
- After graphs are recomposed, step 3 converts them back to deployment descriptor format and fed to the D&C framework for deployment. For the operational strings with dependencies with each other, the D&C framework can then deploy the operational strings from the highest priority to the lowest priority.

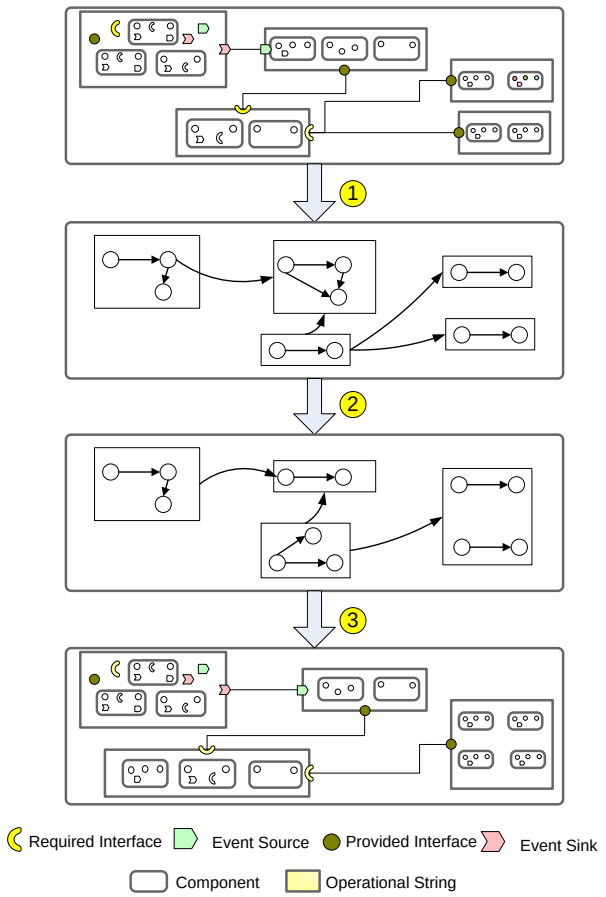

Figure 1: Overview of Our Approach

When a DRE system has many operational strings with complex dependencies it is hard to determine manually which components in which operational strings should be migrated and which operational string to migrate. This paper therefore makes the following three contributions to the research on D&C for component-based DRE systems:

- Analyze dependency relationships among opera-

1 In the context of this paper, *promoting* a component means that before this component is deployed it is temporarily moved from a lower priority operational string to a higher priority operational string for deployment purpose only.

tional strings to determine how each relationship can affect deployment predictability.
- Present a multi-graph algorithm called "partial priority-inheritance via graph recomposition" to avoid deployment priority inversion.
- Empirically evaluate the multi-graph algorithm to determine how effective it is on a representative DRE system.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes a representative DRE system case study that elicits key challenges to ensure the predictability of operational string D&C; Section 3 presents the PARIGE algorithm based on a multi-graph recomposition technique that resolves these challenges; Section 4 presents results of experiments that evaluate our techniques empirically; Section 5 compares our work with related research; and Section 6 presents concluding remarks and lessons learned.

2. Motivating Case Study

This section describes different configurations of operational strings in DRE systems that can cause deployment priority inversion to occur due to the dependencies among the strings. To make our discussion concrete, we use NASA's Magnetospheric Multi-Scale (MMS) mission system [8] as a case study. We first present the case study and then identify key challenges that must be addressed to ensure D&C predictability for the case study.

2.1. Overview of NASA MMS Mission System

The NASA Earth Science Enterprise's MMS mission system system uses five satellites with multiple sensors on each satellite to perform solar-terrestrial probe task. The satellites orbit the earth in formation and collect electromagnetic and particle data in the earth's magnetosphere. The MMS mission operates in three data modes: *slow*, *fast*, and *burst*. These data modes may also include different goals, orbits, and data priorities. Each satellite must be capable of determining the necessary task sequences to achieve prescribed goals based on the current environmental and system conditions, as well as revising task sequences in response to changing conditions.

To achieve autonomy, an automated planner is deployed within the MMS system to handle autonomous mode changes driven by the satellite position and the results of analyzing collected data. The task sequences are implemented by components for coordinating the trajectory and orientation of satellites, sensor selection and data collection for individual satellites, and data integration and compression to create telemetry streams that are beamed down to earth stations.

Figure 2 shows three operational strings that a planner generates for a mission task of one of the satellites. Each operational string has different deployment

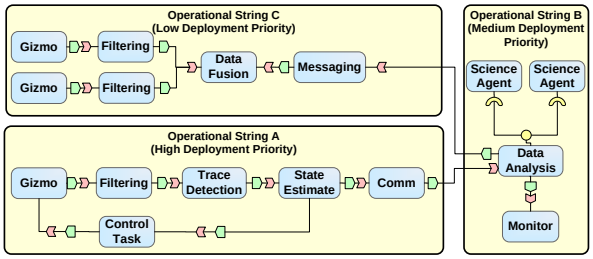

Figure 2: Operational Strings Generated by Planner

priority (*i.e.*, high, medium, and low) that are determined by how each operational string is accessed by the overall MMS system. The three operational strings are briefly described as follows:

- **Operational string A** defines a mission-critical task that collects field data when a satellite moves to particular locations. To ensure this task is performed properly, the operational string must be deployed as fast as possible to avoid loss of data. Operational string A can store the collected data in its own data store, but can also send the data to other operational strings through its event sources.
- **Operational string B** is designed for a domain-centric data analysis. Different scientific analysis tasks can be configured through the facets of components of this operational string. For example, Science Agent components can be configured to achieve scientific objectives, such as analyzing models of complex phenomena like extended weather forecasting.
- **Operational string C** is for less essential data analysis task and can collect auxiliary field data, such as Sun zenith, satellite view zenith [9], which can be served as additional input for analysis. This operational string only operates occasionally, *e.g.*, when the data analysis component in operational string B explicitly issues a request to request such data as additional input for scientific analysis models. The components in operational string C are driven by events exchanged through their event sources and sinks.

Operational strings are organized from domain perspective, *e.g.*, each operational string is designed to accomplish certain domain tasks, such as collecting certain field data, or perform certain analysis on different

data models. In our MMS scenario, operational string A services (*e.g.*, collecting essential field data for scientific analysis) are most important for the MMS system, so it has the highest deployment priority among the three operational strings. Conversely, operational string C has the lowest deployment priority among the three since it is designed a less essential, *i.e.*, collecting auxiliary data only when necessary. Finally, operational string B is designed to have medium priority because its scientific analysis role is less important than operational string A, but more important than operational string C. Operational string B, however, needs to send events to operational C to notify it to collect auxiliary field data and perform analysis when necessary.

As shown in Figure 2, there are two dependencies between operational strings: from *A to B* and from *B to C*. These dependencies cross the boundary of an individual operational string. We therefore call them *external dependencies*, in contrast to those dependencies within an operational string.

2.2. Challenges of Ensuring D&C Predictability in the MMS Case Study

Below we describe four challenges that arise when operational strings are deployed dynamically in open DRE systems, such as the NASA MMS mission case study described above.

Challenge 1: Avoid deployment priority inversion between two operational strings. In conventional D&C technologies, such as the OMG D&C specification [10, 6], when a component of an operational string has a connection (either facet/receptacle or event sink/source) to another component in a separate operational string, an *external reference* must be specified to indicate the remote component and the provided port in the other operational string upon which it depends. To deploy this operational string successfully, the external reference endpoint of the other operational string must be activated before the deployment of source operational string can occur. When such a dependency is from a higher priority operational string to a lower priority operational string, however, the low priority operational string must be deployed *before* the high priority operational string can be deployed to avoid deployment failure caused by the dependency, which results in a priority inversion at deployment-time.

For example, in our MMS system case study described in Section 2.1 the dependency from operational string B (medium priority) to operational C (low priority) can cause a deployment priority inversion between operational strings B and C. This dependency requires the deployment of operational string C before operational string B to resolve the dependency. Not all components in operational string C need be deployed to resolve the external dependency between B and C.

Subsection 3.2.1 describe how we address this challenge by promoting components from the lower priority string to the higher priority string.

Challenge 2: Avoid deployment priority inversion propagation effect. A more general priority inversion situation involves multiple operational strings. In this case, to resolve a dependency from a higher priority string to a lower priority string, not only must the lower priority string be deployed before the high priority operational string, but also the operational strings the lower priority string depends on. When these operational strings have lower priority than the high priority string, however, deployment priority inversion will occur between operational strings.

For example, in our MMS system case study operational string A has a high priority and an external dependency on operational string B. More specifically, it is the `Data Analysis` component of operational string B that A depends on. The `Data Analysis` component further depends on the `Messaging` component in operational string C, however, which can cause another deployment priority inversion between A and C.

Subsection 3.2.2 describes how we address this challenge by recursively tracing dependencies from the high priority string to all lower priority strings.

Challenge 3: Avoid deployment failure when circular dependency exists among multiple operational strings. Conventional D&C techniques [10, 6] will fail if a circular dependency exists among multiple operational strings. The central coordinated phased deployment technique applied in conventional D&C technologies can effectively address the circular dependency problem within a single operational string. Conventional D&C technologies cannot handle cases, however, where a D&C request involves multiple operational strings and if a circular dependency exists among these operational strings.

Limitations with circular dependencies arise because the deployment of an operational string is treated as an indivisible process, *i.e.*, conventional D&C technologies treat operational strings as primitive units. If a dependency exists between two operational strings, therefore, one must be deployed before another to resolve such dependency requirements. Such treatment, however, makes it hard to handle circular dependencies among different operational strings.

For example, in our MMS system case study a circular dependency will exist between *A, B, and C* if the less essential operational string C depends on operational

string A because it needs to access a type of service provided by A. Since conventional D&C approaches treat each operational string deployment as an indivisible process, such deployments cannot be handled properly.

Subsection 3.2.3 describes how we address this challenge by promoting components in the circular dependency trace among operational strings.

Challenge 4: Improve the overall utility of the operational strings being deployed. The dynamic nature of open DRE systems require on-demand deployment of many operational strings that cooperate with each other to ensure the system is kept in sync with changing mission goals or environmental changes. Since multiple operational strings with different importance to the entire DRE system may be deployed at the same time, the goal of a D&C framework is to deploy these operational strings in an effective way to improve the overall QoS of DRE systems in the following two dimensions:

- Since operational strings with the highest priority are the most important to the entire DRE system, these operational strings should be deployed as early as possible to ensure the DRE system responsiveness due to the changing environment or mission modes. For example, in our MMS system case study, operational string A should always be deployed immediately since it has the highest priority.
- Since each operational string has a utility value associated with it, a D&C framework should try to finish deployment of each individual operational string as early as possible by taking its utility value into account. For example, in our MMS system case study there is a dependency from operational string A to operational string B. Although the deployment of operational string A should finish first due to its higher priority, the time to finish deploying operational string B is also an contributing factor to the overall system utility and should thus be considered.

Subsection 3.2.4 describes how we address this challenge by selectively applying the PARIGE algorithm to the input of operational strings.

3. An Algorithm for Partial Priority Inheritance via Graph Recomposition

This section describes how we resolved the challenges described in Section 2.2 using an algorithm called *partial priority inheritance via graph recomposition* (PARIGE). This algorithm converts each operational string into a graph, where each vertex and edge of the graph represent a component and a connection/dependency between two components, respectively. If there is an external dependency between operational strings, then the graph converted from one operational string will have a special type of vertex that represents the external dependency. This special vertex type contains information about the actual refereed operational string and the component in the operational string that it depends on.

The PARIGE algorithm migrates components from one graph to another based on operational string characteristics, including their priorities and their dependency relationships with other operational strings in the same deployment request. After graphs for all the operational string are recomposed to account for the component promotion, a new set of operational strings will be populated from these recomposed graphs. These new strings avoid deployment priority inversion between operational strings and break the circular dependency among all the operational strings. These new set of operational strings can then be deployed by conventional D&C technologies, such as J2EE or OMG D&C models.

To demonstrate the effectiveness of PARIGE, we integrated the algorithm into the central coordinator `ExecutionManager` of OMG D&C model to perform experimental analysis. The `ExecutionManager` is runs as a daemon and is used to manage the deployment process for a domain. In accordance with the D&C specification, a *domain* is a target environment composed of *nodes*, *interconnects*, *bridges*, and *resources*. An `ExecutionManager` plays the role of central coordinator that manages the nodes in the DRE system environment. On each node, a **NodeManager** runs as a daemon process and manages the deployment of all components that reside on that node, irrespective of which application they are associated with.

3.1. Overview of the PARIGE Algorithm

Although the PARIGE algorithm recompose operational strings by promoting components from one operational to another, it has also the following properties that makes it well-suited for D&C of DRE systems:

1. The PARIGE algorithm does not affect the functional behavior of component-based DRE systems. Component-based DRE systems are deployed in the form of operational strings that consist of multiple monolithic components connected with each other via their ports. Two operational strings can have dependencies with each other. A dependency between two operational strings is es-

sentially a connection from a monolithic component in one operational string to a port of a monolithic component in the other operational string.

The PARIGE algorithm evaluates the component dependency relationships and their priorities and recomposes these operational strings to avoid deployment priority inversion. From the perspective of *all* operational strings to deploy, however, the individual monolithic components and their connections among each other are not modified by the algorithm. In particular, the effect of the PARIGE algorithm on operational string recomposition is only visible for the D&C framework, which does not affect the running system's functional behavior. This algorithm thus does not affect the functionally of operational strings because the topology of all the operational strings (including all the monolithic components and connections) that fulfills functional behavior of the system remains unchanged.

2. The PARIGE algorithm only migrates components that can improve QoS behavior of operational strings. When components are migrated from a lower priority operational string to a higher priority operational strings, the priority of the components is also bumped up to match the priority of the higher priority string, which is essential for a task to avoid priority inversion at deployment-time [11]. Since the PARIGE algorithm only migrates components that one or more higher priority operational strings have dependencies on—and does not migrate components without such dependencies—the QoS behavior of the operational strings will only be improved and never worsened.

Figure 3 presents an overview of the PARIGE algorithm by showing an example with three operational strings having priorities high, medium, and low. The dotted and solid arrows represent dependencies between operational strings. In particular, the dotted arrows in the figure represent priority inverted dependencies, *i.e.*, dependencies from higher priority operational strings to lower priority operational strings. Likewise, the solid arrows represents external dependencies without causing priority inversion.

The numbered vertices in Figure 3 denote the vertices migrated from one graph into another. For example, in the first iteration of the algorithm, one vertex is migrated from the medium priority operational string to the high priority operational string and another vertex is migrated from the low priority operational string to the high priority string. In the second iteration, another vertex is migrated from the low priority operational string to the medium priority string.

The PARIGE algorithm recomposes the graphs by

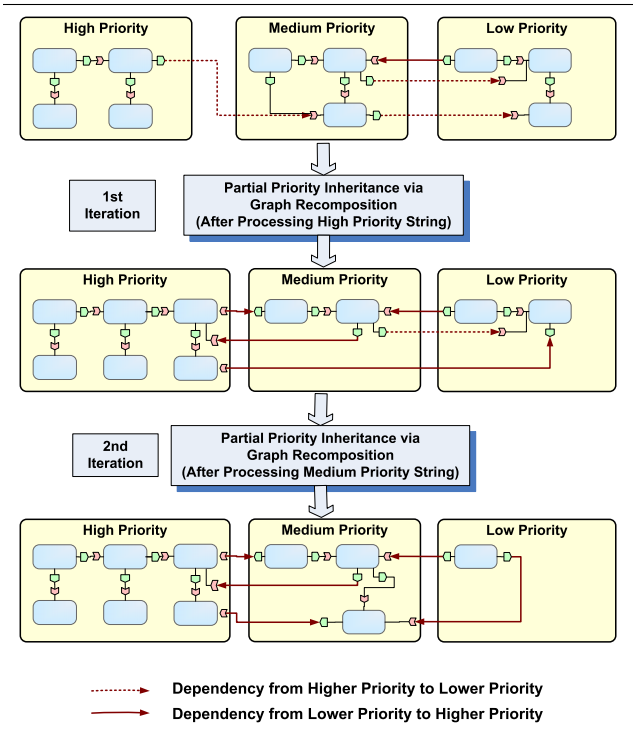

Figure 3: An Overview of the PARIGE Algorithm

parsing the input set of graphs and removing dotted arrows by promoting some component(s) from a lower priority operational string to a higher priority string. This process may introduce some new dependencies between operational strings due to component promotion. The algorithm, however, only introduces solid arrows, *i.e.*, only dependencies from lower priority operational strings to higher priority strings exist after the recomposition.

When the algorithm finishes, all dotted arrows in the graphs will be removed and there will be no dependencies from higher priority operational strings to lower priority operational strings. As a result, both priority inversion at run-time and deployment-time can be avoided. Moreover, all circular dependencies among operational strings, if any, will also be removed in the recomposition process.

3.2. Analysis of Operational String Dependencies with Deployment Priority Inversions

The goal of the PARIGE algorithm is to remove *all* dependencies from higher priority operational strings to lower priority operational strings. To accomplish this, the algorithm starts with the operational string having the highest priority and processes all the external dependencies of this operational string. After all

external dependencies from the highest priority operational string are removed, the algorithm then processes the operational string with the next highest priority. When multiple operational strings have the same priority, we apply the following tie-breaking policies sequentially: (1) evaluating the second metric of each operational string, if given, (2) evaluating the number of external dependencies to the same priority operational strings and treat the operational string with the least number of external dependencies as the higher priority than others, and (3) break the tie randomly if such a tie still exists.

When processing an external dependency from a higher priority operational string to a lower priority operational string, the algorithm must trace the dependency into other operational strings and migrate components from them if the lower priority operational string has dependencies to them. For example, if a high priority operational string depends on a component X in a medium priority operational string, and if component X also has dependency on a component in a low priority operational string, then the component in the low priority string also must be migrated into the high priority string.

We define a *dependency trace* as a totally ordered sequence S. Each element in the sequence is a component of an operational string that has a priority value associated with it. The starting element of the sequence is the source component of the external dependency of interest. The PARIGE algorithm analyzes all the dependency traces in the operational strings and recomposes the operational strings based on dependency trace characteristics. To analyze the dependency trace we classify the operational dependency relationships into the three categories described below, which will be used to address the first three challenges in Section 2.2.

3.2.1. Handling Challenge 1 → Promotion of Components Between Two Operational Strings (Non-circular) In this case, a dependency occurs between two operational strings, where a high priority operational string has a dependency to a lower priority operational string, as shown in Figure 4. We assume no circular dependency exists in this case (circular dependencies will be discussed in Section 3.2.3).

As shown in Figure 4, the unique characteristic of this category is that the dependency trace does not cross the boundary of the lower priority operational string. Since no other operational strings are involved besides the two operational strings of interest, removing such a priority inverted external dependency only requires promoting all components in the dependency

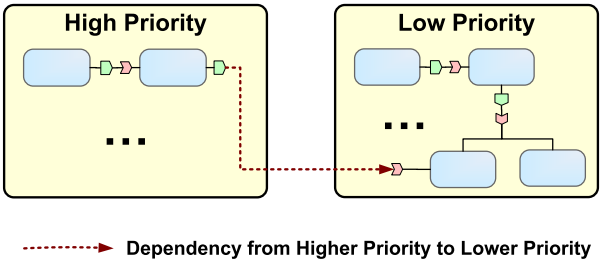

Figure 4: A Dependency Trace Spanning Two Operational Strings Only

trace from the lower priority operational string to the high priority one.

In the context of MMS case study, a priority inverted external dependency exists between operational string B and operational string C, as illustrated in the upper half of Figure 5. Our solution removes this priority in-

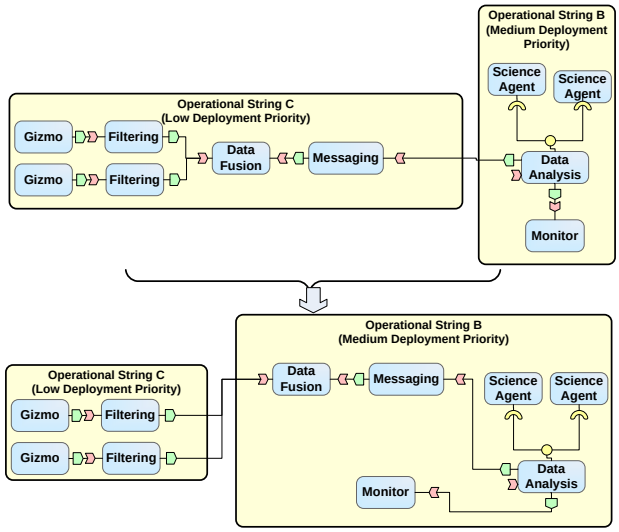

Figure 5: A Dependency Trace across Two Operational Strings

verted external dependency by promoting components `Messaging` and `Data Fusion` from operational C to operational B, as illustrated in the bottom half of Figure 5. After the promotion, operational string B does not have any dependencies to operational string C. The two newly created external dependencies from the two `Filtering` components in operational string C to operational B are essential to make sure the functional behavior of the MMS system is not changed. Since these two newly created external dependencies are not priority inverted, deployment priority inversion between the

two operational strings can be avoided.

3.2.2. Handling Challenge 2 → Promotion of Components Across Multiple Operational Strings (Non-circular) This more general case involves multiple operational strings, with a dependency trace that spans across the operational strings. As before, we assume no circular dependency exists since that will be discussed as a separate case in Section 3.2.3.

A dependency trace that spans across multiple operational strings can be further categorized into the following two situations.

1. Ordered dependency trace. Figure 6 shows an ordered dependency trace. In an ordered dependency trace the priorities of each element in the sequence have a non-increasing order, *i.e.*, all external dependencies in the sequence are priority-inverted. As a result, all the priority-inverted external dependencies must be removed through the component promotion mechanism described in Section 3.1. The category described in Section 3.2.1 where only two operational strings are involved is a special case of an ordered dependency trace. To remove all priority-inverted external dependencies, the PARIGE algorithm simply migrates all components in the dependency trace into the operational string where the first component of the dependency trace is located.

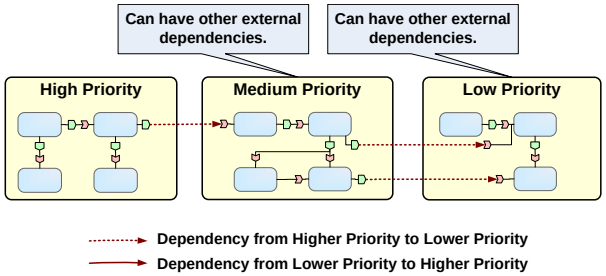

Figure 6: An Ordered Dependency Trace

2. Unordered dependency trace. Figure 7 shows an unordered dependency trace, where the priorities of the elements in the dependency trace do not have a particular order, *i.e.*, some external dependencies are priority-inverted (shown as dotted lines), whereas others are not (shown as solid lines). The PARIGE algorithm always starts with the highest priority operational string and removes all external dependencies on it before moving to the next operational string. The algorithm therefore ensures that in an unordered dependency trace, the elements whose priorities are higher than that of the starting element will have no external dependencies, which ensures the convergence of the algorithm.

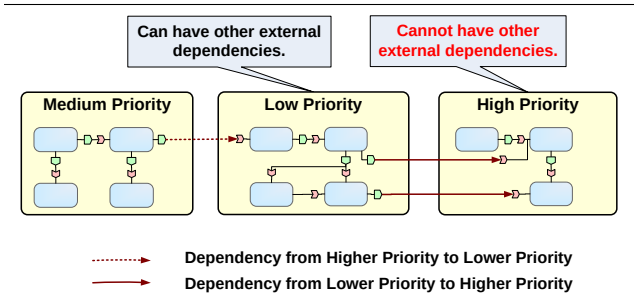

Figure 7: An Unordered Dependency Trace

For example, given the 3 operational strings from Section 2, if the high priority operational string has an external dependency to a component in low priority operational string and this component must be migrated into the medium priority operational string. When this promotion happens, the high priority string will depend on the medium priority string, which introduces additional priority-inverted external dependencies.

To remove all priority-inverted external dependencies of an unordered dependency trace, we break the entire dependency trace into two concatenated segments. As shown in Figure 8, the first segment is a priority unordered subsequence, where all the priorities of operational strings are lower than the priority of the source of the dependency trace. The second segment is a priority ordered subsequence, where all priorities are higher than the priority of the source of the dependency trace. For the first segment, we can migrate all the components in the subsequence into the operational string where the first component of the dependency trace is located, which will ultimately result in an ordered dependency trace.

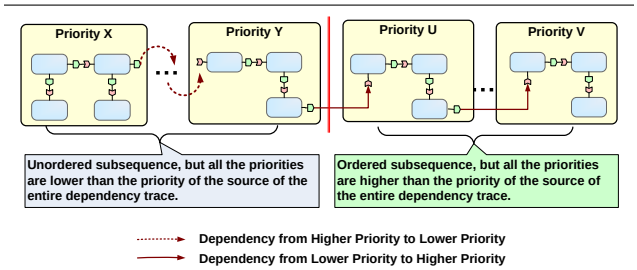

Figure 8: Two Partitions of An Unordered Dependency Trace

In the context of MMS case study, a priority inverted external dependency exists between operational string

B and operational string C, and another priority inverted external dependency exists between operational string A and operational string B, as illustrated in the upper half of Figure 9. Our solution traverses the de-

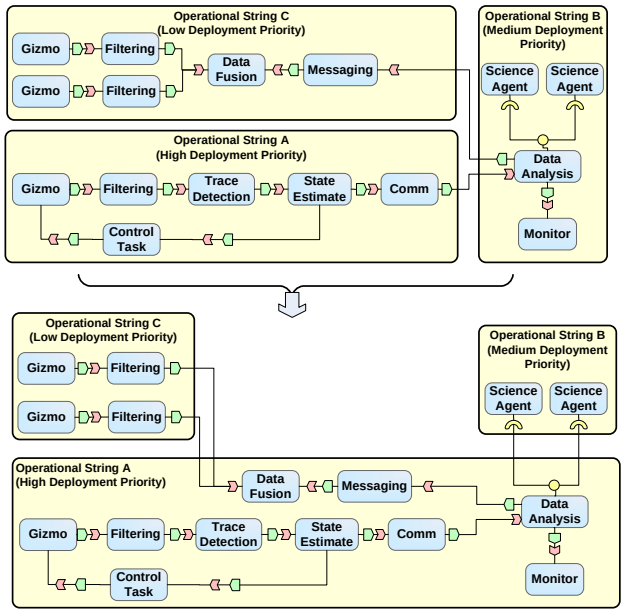

Figure 9: A Dependency Trace across Three Operational Strings

pendency trace across all the three operational strings, and removes both priority inverted external dependencies. By doing this, components `Messaging` and `Data Fusion` are migrated from operational C to operational A, and components `Data Analysis` and `Monitor` are migrated from operational B to operational A, as illustrated in the bottom half of Figure 9. In our MMS case study, only one priority inverted external dependency exists from operational string A, so after traversing it, operational string A does not have any more priority inverted external dependency remaining. The D&C service can then deploy operational string A before other operational strings due to its highest priority, therefore avoiding priority inversion between operational string A and other operational strings.

3.2.3. Handling Challenge 3 → Promotion of Components in the Circular Dependency Trace Among Operational Strings Circular dependencies may exist among two or more operational strings, as shown in Figure 10. When we discussed how the PARIGE algorithm processed an unordered dependency trace in Section 3.2.2, we showed that a dependency trace can be divided into two subsequences. The priority of each element in the first subsequence is less

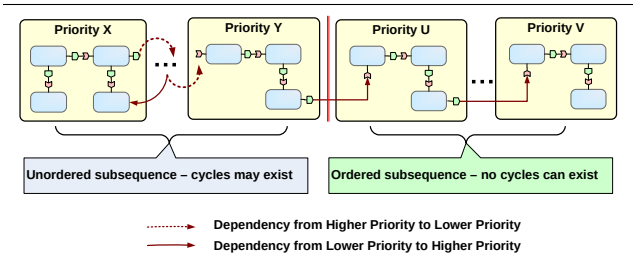

Figure 10: Circular Dependencies in a Dependency Trace

than or equal to the priority of the source element of the dependency trace. Likewise, the priority of each element in the second subsequence is greater than that of the source elements of the dependency trace.

Since no components in the second subsequence can have priority inverted external dependencies, cycles among operational strings can only occur in the first subsequence, as shown in Figure 10. To remove cycles across multiple operational strings, only components in the first subsequence must be migrated. The PARIGE algorithm therefore only needs to migrate components existing the first subsequence to bring cycles into the same operational string.

In the context of MMS case study, a circular dependency exists between operational string A and operational string B, as illustrated in the upper half of Figure 11. Our solution traverses the dependency trace

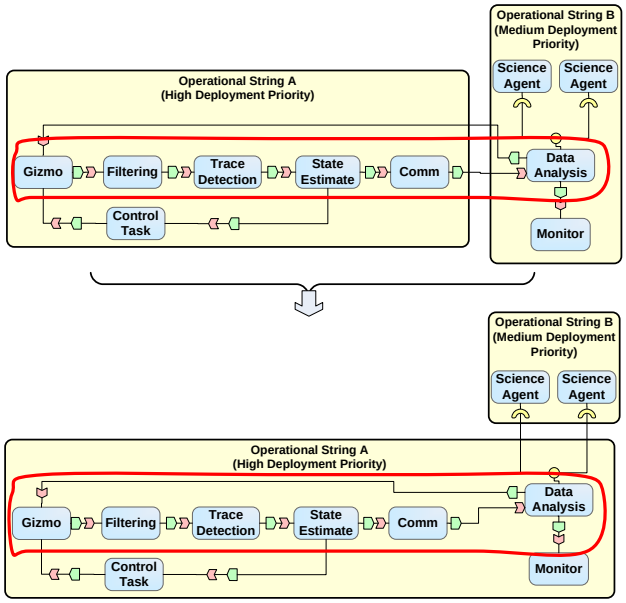

Figure 11: A Circular Dependency Trace Case

across from the operational string A, and promoting components `Data Analysis` and `Monitor` from operational B to operational A, as illustrated in the bottom half of Figure 11. After such promotion, the circular decadency trace still exists but it is contained *within* the operational string A rater than across the boundary operational strings, thereby avoiding deployment failure.

3.2.4. Handling Challenge 4 → Selectively Applying the Algorithm to Operational Strings The deployment latency of an operational string is affected by the size of the operational string. The dependency trace technique described above will migrate components from lower priority operational strings to higher priority operational strings. Although this technique can ensure operational strings with the highest priority be deployed as early as possible, it may reduce the overall utility by delaying the deployment of lower priority operational strings in the worst case, as described as Challenge 4 in Section 2.2. This delay occurs when *all* components in the lower priority operational strings must be migrated into the higher priority operational string, which makes both operational strings merge together and hence finish their deployment at the same time. Merging operational strings can adversely affect the responsiveness of lower priority operational strings since these strings could have been deployed first to satisfy the dependency requirements without affecting the responsiveness of higher priority operational strings.

To overcome this difficulty, we apply an optimization technique that is based on the results produced by all the dependency traces of an operational string, as described in Subsections 3.2.1, 3.2.2, and 3.2.3. If the dependency traces of an operational string migrate *all* the components of an lower priority operational string into the higher priority operational string, the PARIGE algorithm simply chooses the cached graph representation of these two operational strings before the dependency traces techniques are applied. This optimization improves the overall utility of the operational strings by finishing the deployment of lower priority operational string first. The deployment of lower priority string can thus be finished earlier rather than at the moment when all components in both operational strings are deployed.

3.3. Design of the PARIGE Algorithm

The PARIGE algorithm uses multi-graph breadth first search (BFS) to trace dependencies and graph reconstruction to migrate components and connections between components. Each graph corresponds to an operational string and can have two types of vertices:

- A **regular vertex**, which refers to a component within the operational string.
- A **reference vertex**, which refers to a component in another operational string on which it has a dependency.

For example, two operational strings have dependencies shown in Figure 12. Based on the dependency relationship between these two operational strings, the PARIGE algorithm converts them into two graphs with one reference type vertex in each graph, with reference type shown as shaded vertices in the figure. Given a reference type vertex, in O(1) time we can find the referred operational string and referred component within the operational string.

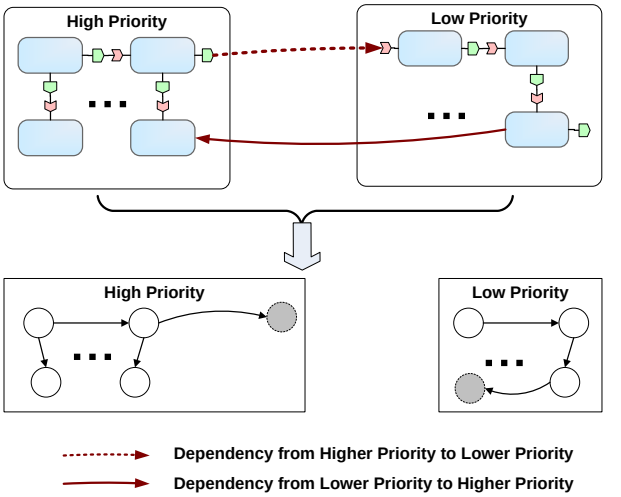

Figure 12: Convert External Dependencies into Reference Type Vertices

In our algorithm, we define the following two types of operators against vertices in the graph:

- $Promote(V)$: Promote a reference type vertex to a regular type vertex.
- $Demote(V_1, G_1, V_2, G_2)$: Demote a regular type vertex V_1 in graph G_1 in to a reference type vertex, and the referred graph and vertex will be G_2 and V_2, respectively.

Algorithm 1 presents the PARIGE algorithm, which uses a recursive procedure defined in Algorithm 2 to process each dependency trace.

Algorithm 1 PARIGE Algorithm

Input: Set of Operational Strings (Represented as Graphs)

Output: Set of Operational Strings (Represented as Graphs)

```
Sort strings with decreasing priority;
foreach String G_i in the sorted array do
    foreach External Dependency j of String G_i do
        Get the source vertex V_j1 of j;
        Get the destination vertex V_j2 of j;
        if priority(V_j1) > priority(V_j2) then
            Get the referenced string G_e and vertex V_e;
            process_dependency_edge (G_i, V_j2, G_e, V_e);
        end
    end
end
```

Algorithm 2 A Recursive Procedure to Process a Dependency Trace

Input: G_1: source graph with higher priority
G_2: destination graph with lower priority
V_1: source vertex (reference type)
V_2: destination vertex (regular type)

Output: Recomposed graphs G_1 and G_2

```
Promote (V_1);
Demote (V_2, G_2, V_1, G_1);
Do a BFS (G_2, V_2) and foreach visited edge E_i do
    Get the source vertex V_i1;
    Get the destination vertex V_i2;
    if V_i2 is regular type then
        Remove_Vertex (G_2, V_2);
        Add_Vertex (G_1, V_2);
        Remove_Edge (G_2, E);
        Add_Edge (G_2, E);
    end
    else
        Get the referred graph G_ie of V_i2;
        Get the referred vertex V_ie of V_i2;
        process_dependency_edge (G_1, V_i1, G_ie, V_ie);
    end
end
```

3.4. Analysis of the Algorithm

To show that it is possible to apply the PARIGE algorithm at run-time to deploy operational strings dynamically, we now analyze the time complexity of the PARIGE algorithm and evaluate different effects of applying this algorithm to different configurations of operational string.

3.4.1. Time Complexity Analysis The input of the PARIGE algorithm is defined as follows:

N: Total number of operational strings

C: Total number of dependencies between operational strings

$|V|$: Average number of components within an operational string

$|E|$: Average number of connections within an operational string

As shown in Algorithm 1, the sort of all external dependencies has a complexity of $O(C * log(M))$. The recursive procedure shown in Algorithm 2 shows that processing each dependency trace has a time complexity of $O(N * (|V| + |E|))$. The overall time complexity of the PARIGE algorithm exhibits a linear complexity of $O(C * N * (|V| + |E|)$. In practice, C tends to be much smaller compared with $|V|$ and $|E|$.

In summary, the linear property of our algorithm makes it possible to apply it dynamically at run-time. Section 4 evaluate the performance overhead empirically in the context of the NASA MMS mission system case study.

3.4.2. PARIGE Algorithm Effects on Operational String Deployment Two effects that the PARIGE algorithm could have on the predictability of operational string deployment are described below.

Operational string growth effect. This effect measures the cost of promoting a number of components from lower priority operational strings to higher priority operational strings. Since the deployment of each component takes time and consumes resources, the fewer components that are migrated, the more benefits the algorithm can provide since priority-inverted dependencies can be satisfied without deploying many components in lower priority operational strings.

In the worst case, *all* components from lower priority operational strings could be migrated to higher priority operational strings, which essentially merges different operational strings together. In production DRE systems, such worst cases happen rarely, *i.e.*, all the components in all operational strings have just only one dependence trace. Even in such situations, the PARIGE algorithm still performs the same as a conventional approach that does not take priority into account and only accounts for dependencies among operational strings.

Component host distribution effect. This effect means that due to the promotion of compo-

nents, components that can be deployed by contacting the `NodeManager` once now contacts the same `NodeManager` multiple times during deployment. For example, if an operational string A has 1 component to deploy on $Node_A$ and operational string B has 2 components to deploy on $Node_B$ these two operational strings can be deployed by contacting the `NodeManager` on each node only once. If the algorithm moves one component from operational string B to operational string A, then the `NodeManager` on $Node_B$ must be contacted twice, once when deploying the migrated component in operational string A and once when deploying the remaining component in operational string B.

Such an effect can increase the overall deployment time due to the increasing number of round trip delays. One way to alleviate this problem is to increase the parallelism among different nodes by using asynchronous techniques between the `ExecutionManager` and `NodeManagers`, such as the Asynchronous Method Invocation (AMI) messaging policy provided by CORBA [12]. For example, AMI can coordinate all the `NodeManagers` in the domain parallelism deployment can be achieved among all the nodes, therefore alleviating the component host distribution effects.

4. Empirical Results

To evaluate the benefits of our PARIGE algorithm, we applied it to a representative DRE system prototype of the NASA MMS mission system described in Section 2. This section first describes the characteristics of the hardware and software testbed and explains our experiment design. We then empirically evaluate the effectiveness of our PARIGE algorithm in three dimensions: (1) Sections 4.2 and 4.3 empirically measure the effectiveness of PARIGE to address challenges 1, 2, and 3 in Section 2, Section 4.4 empirically measures the effectiveness of PARIGE to address challenge 4 in Section 2, which uses higher level metrics to measure the DRE system D&C QoS, and (3) Section 4.5 empirically measures the performance overhead of the PARIGE algorithm.

4.1. Hardware and Software Testbed

We used the ISISlab open testbed (`www.dre.vanderbilt.edu/ISISlab`) for all our experiments. Our experiments used up to 6 nodes running Linux FC4 with Ingo Molnar's real-time kernel patches. When operational strings are deployed we use one node to run the central coordinator `ExecutionManager` and the rest of the nodes as the deployment targets.

The NASA MMS mission system prototyped used for our experiments was developed using the CIAO [13] and DAnCE [7] component middleware. This application consists of 45 components grouped together into 3 operational strings Figure 13 shows an example operational string consisting of a science agent component that decomposes mission goals into navigation, control, data gathering, and data processing applications. Multiple gizmo components are connected to the science agent and are also connected to different payload sensors. Each gizmo component collects data from the sensors, which have varying data rate, data size, and compression requirements.

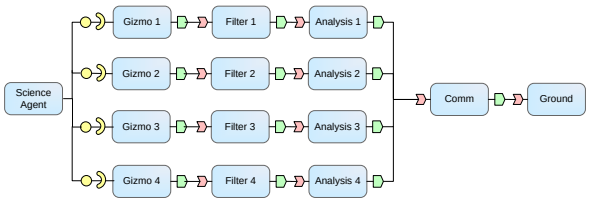

Figure 13: A Sample Operational String of the Experiments

The collected data is passed through filter components, which remove noise from the data. The filter components pass the data onto analysis components, which compute a quality value indicating the likelihood of a transient plasma event. Finally, the analyzed data from each analysis component is passed to a comm (communication) component, which transmits the data to the ground component at an appropriate time.

An operational string can span across multiple physical nodes. The assignment of components to nodes is determined by a planner using high-level resource planning algorithms, and was available as input to our PARIGE algorithm. We intentionally choose different assignments for our experiments to compare how they could affect the predictability and performance different operational strings deployments.

4.2. Effects of Operational String Recomposition

Hypothesis. The hypothesis of this experiment is that the PARIGE algorithm should not change the functional correctness of the input operational strings but should produce correct dependencies between operational strings. In particular, the PARIGE algorithm must ensure (1) all the dependencies between components of original operational strings should remain the

same after the recomposition and (2) the dependencies between operational strings must be changed such that no dependencies exist from higher priority operational strings to lower priority operational strings.

Experimental design. The experiments consist of 3 operational strings with each operational string having 15 components. The high priority operational string has one dependency to the medium priority operational string and the medium operational string has one dependency to the low priority operational string. We measure the total number of components, number of nodes, and number of dependencies (both internal and external) of each operational string before and after applying the PARIGE algorithm.

Empirical results and analysis. Table 1 summarizes the number of components, number of nodes, and number of dependencies for each operational string before and after we run the PARIGE algorithm.

	High	Medium	Low	Total
Components Before	15	15	15	45
Components After	18	14	13	
Dependencies Before (H-¿L / L-¿H)	1/0	1/0	1/0	3
Dependencies After (H-¿L / L-¿H)	0/0	0/3	0/3	3
Dependencies Before (Internal + External)				63 (60+3)
Dependencies After (Internal + External)				63 (57+6)

Table 1: PARIGE Effect on Operational Strings

The results in the figure indicate that the number of components of high priority operational string increases from 15 to 19, while that of both medium priority and low priority operational strings decreases from 15 to 13, so the total number of components do not change. In addition, before the experiment, the 3 operational strings have 60 internal dependencies and 3 external dependencies, with 63 dependencies in total. After the experiment, the number of internal dependencies decreases by 1 to 59 and the total number of dependencies increase by 1 to 4, with the total number still remains the same, which is in accord with the first part of our hypothesis. Finally, after applying the PARIGE algorithm, all dependencies from higher priority to lower priority operational strings are removed, which validates the second part of our hypothesis.

4.3. Deployment Latency vs. Deployment Priority

Hypothesis. The hypothesis of this experiment is that the PARIGE algorithm can avoid priority inversion when deploying multiple operational strings where higher priority operational strings have dependencies on lower priority operational strings.

Experimental design. We conducted two experiments on different configurations of operational string dependencies. Our first experiment consisted of 3 operational strings, each of which having 15 components evenly distributed into 5 nodes. Therefore, each node has 9 components. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. The dependency between two operational strings is shown in Figure 14.

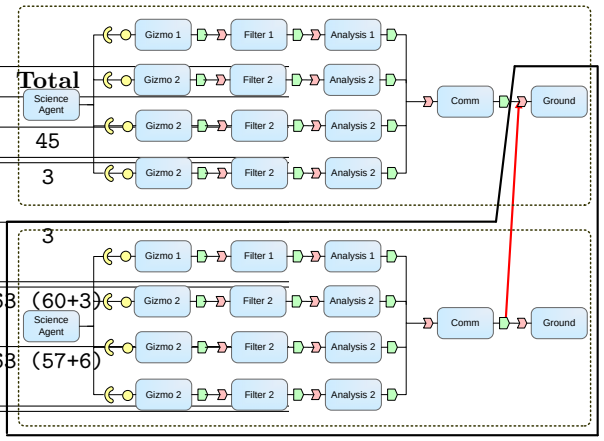

Figure 14: Operational String Configuration with Low Growth Rate

Next we conducted another experiment with the external dependency between two different components in the operational strings, as shown in Figure 15. We measured how the end-to-end deployment latency of each operational string can be affected in this configuration. In this experiment, there are only two operational strings, each having 15 components. The high priority operational string has one dependency on the low priority operational string, as shown in Figure 15.

Both experiments first measured the end-to-end latency for deploying each operational string without applying the PARIGE algorithm. We then measured the end-to-end latency for deploying each operational strings with the PARIGE algorithm to see how latency relates to their priorities.

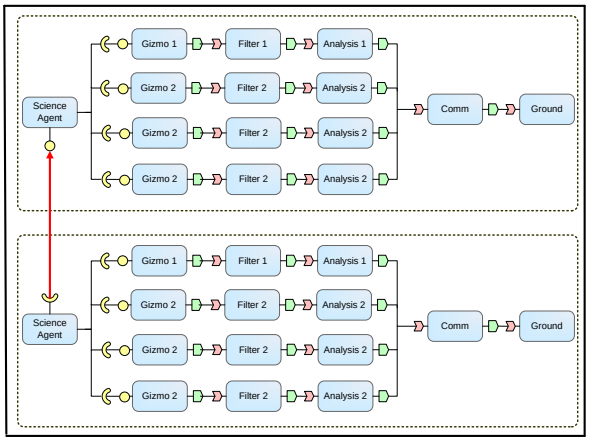

Figure 15: Operational String Configuration with High Growth Rate

Empirical results and analysis.

Figures 16 and 17 shows the end-to-end latency of D&C request for each operational string in the two experiments described above. As shown in the Figure 16,

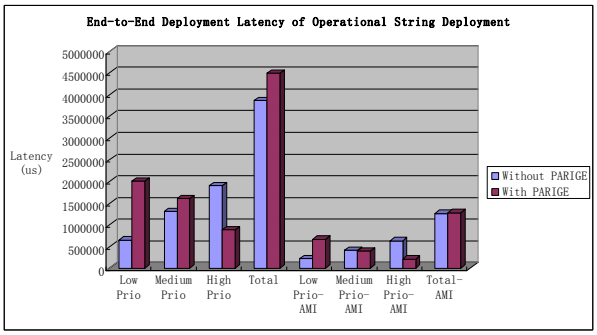

Figure 16: D&C Latency Changes by the PARIGE Algorithm

without applying the PARIGE algorithm, the high priority operational string yields the highest latency while the low priority operational string yields the lowest latency, while the latency of medium priority operational string lies in between.

In our experiments, there is one dependency from high priority operational string to medium priority operational string and another dependency from medium priority operational string to low priority operational string. Without applying the PARIGE algorithm, therefore, the low priority operational string must be deployed first among the three, followed by medium priority and high priority operational strings, respectively. The PARIGE algorithm removes the priority inverted dependencies which avoids deployment priority inversion, as illustrated in the figure.

Figures 16 also shows how the *component host distribution effect* introduced by the PARIGE algorithm is masked by applying AMI messaging policy, as described in Section 3.4.2. In our experiment, applying AMI improves the performance of the deployment in two aspects. First, the deployment latency of *each* operational string is reduced because of the `ExecutionManager` can coordinate the `NodeManagers` to do deployment in parallel. Second, it masks the component host distribution effect, which results in a reduced total latency of all operational strings, as shown in the figure.

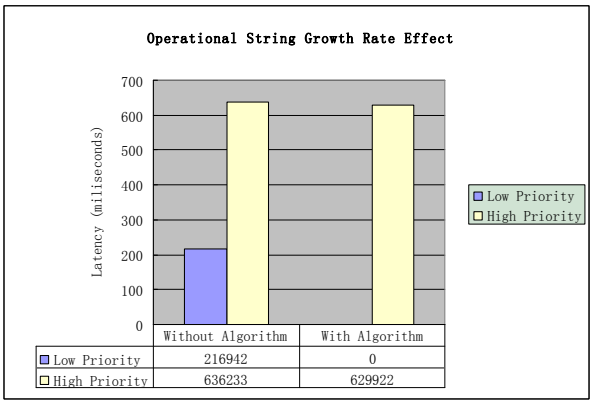

Figure 17: D&C Latency Changes by the Algorithm

High Operational String Growth Figure 17 shows that after applying the PARIGE algorithm the high priority operational string has the lowest latency since it has no external dependency on any other operational strings. The size change of each operational string is also minimal since the number of migrated components is small due to the dependency trace characteristics.

On the other hand, the dependency between the two operational strings in our second experiment caused all components from the low priority operational string to migrate to the high priority string, essentially merging the two operational strings together. As a result, the latency of deploying the high priority operational string is nearly the same as deploying it without applying the PARIGE algorithm. However, in a DRE system with multiple operational strings to deploy, it is rare that all components have only one dependence trace, as described in Section 3.4.2.

4.4. Effectiveness of PARIGE Algorithm Based on High Level Metrics

Hypothesis. The hypothesis of this experiment is to reduce the quiescence time of operational strings during the period when these operational strings are deployed. This experiment measures the effectiveness of the PARIGE algorithm based on human-perceivable metrics. As described in Section 1, the dynamic nature of open DRE systems require on-demand injection of certain operational strings to ensure systems are kept in sync with changing mission goals. When operational strings with different importance to the entire DRE system must be deployed at the same time, the D&C framework must deploy these operational strings in effectively to ensure the overall system QoS. In this experiment, therefore, each operational string is assigned with a mission effectiveness value (MEV) to quantitatively represent its utility value for the entire system, using two utility metrics described below.

Metric *M1*: average MEV loss. The following utility function *M1* measures the average MEV loss caused by the operational string deployment cost.

$$M1 = \sum_{i=1}^{m} \frac{DeploymentTime(S_i)}{TotalDeploymentTime} \times MEV(S_i), \quad i = 1..m \tag{1}$$

Since each operational string has an associated deployment cost (measured by its deployment latency), this cost will necessarily cause a period of quiescence time, which results in a MEV loss.

M1 is computed by first dividing the deployment time of each operational string by the total end-to-end deployment time for all operational strings and then multiply by the MEV of that operational string to produce the weighted MEV loss for that operational string. Next, we sum up all weighted MEV loss of each operational string. In the context of our MMS case study as descried in Section 2, *M1* measures the weighted quiescence time of the the three operational strings and their peer components during the period when the three operational strings are deployed.

Metric *M2*: Highest priority operational string MEV loss. The following utility function *M2* is similar to *M1*.

$$M2 = \sum \frac{DeploymentTime(S_{max})}{TotalDeploymentTime} \tag{2}$$

Rather than measuring the weighted MEV loss of all operational strings, however, *M2* measures the MEV loss the highest priority operational string(s) caused by the deployment.

M2 is computed by dividing the total deployment time of the highest priority operational string by total end-to-end deployment time for all operational strings. In the context of our MMS case study as descried in Section 2, *M1* measures the quiescence time of the highest priority operational string *A* and its peer components during the period when the three operational strings are deployed.

Experimental design. These experiments consisted of 3 operational strings having priorities high, medium, and low, respectively, with each string having 15 components. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. The mission effectiveness values of the three operational string are summarized in Table 2. In general, higher priority operational strings provide higher mission effectiveness values to DRE system than lower priority operational strings.

String Priority	High	Medium	Low
MEV	3	2	1

Table 2: Mission Effectiveness Values of Operational Strings

The first experiment has a high promotion growth effect, and the second experiment has a low promotion growth effect, as described in Section 3.4. The exact configuration of external dependencies in the experiment are the same as those in Section 4.3.

Empirical results and analysis. Table 3 shows that when the operational growth effect is low, *M1* was reduced by 40% and *M2* was reduced by 69%, which indicates that the PARIGE algorithm can significantly reduce the mission effectiveness loss for all operational strings. In the high growth effect situation,

	M1	*M2*
Baseline (without PARIGE algorithm)	4.71	1
Low growth effect	2.83	0.31
High growth effect	5.79	1
High growth effect w/ optimization	4.71	1

Table 3: Impact of Operational String Growth Effect

however, the worst case scenario happens when operational strings are merged together. The result in such worst case scenario shows that *M2* is the same as our baseline case, *i.e.*, without applying the PARIGE algorithm. This operational string merge effect indicates that the utility to the entire DRE system by the highest priority operational string(s) remains the same, but *M1*

(which measures the weighted MEV of all operational strings) is worse than before. To overcome this shortcoming, we applied an optimization technique that selectively chooses the deployment descriptor before applying the PARIGE algorithm, as described in Section 3.2.4, thereby avoiding the degradation of *M1*, as shown in Table 3.

4.5. Performance Overhead of the PARIGE Algorithm

Hypothesis. The hypothesis of this experiment is that the performance overhead of the PARIGE algorithm is small enough so it can be applied to deploy operational strings at run-time. In contrast to off-line analysis techniques, the PARIGE algorithm must be deployed by `ExecutionManager` to handle requests at runtime, therefore, the PARIGE algorithm should not incur excessive performance overhead to the end-to-end latency of deployment of operational strings.

Experimental design. The experiments consist of 3 operational strings each having 15 components and 2 external dependencies in total. The high priority operational string has one dependency on the medium priority operational string, which in turn has one dependency on the low priority operational string. We first measured the end-to-end latency for deploying all the operational strings without applying the PARIGE algorithm. We then measured the end-to-end latency for deploying increasing number of operational strings with the PARIGE algorithm to measure how much latency overhead was contributed by running the algorithm.

Empirical results and analysis. We first measure the PARIGE algorithm performance itself to determine how its performance is affected by the size of the problem, *i.e.*, number of components (determined by number of operational strings) and number of priority-inverted external dependencies. We then measure its performance overhead against an actual example with 3 operational strings and 2 external dependencies, as described above.

Figure 18 shows the performance result of PARIGE algorithm itself with increasing number of components and number of external dependencies. The results show that the performance of PARIGE algorithm is roughly linear to both the number of components and number of external dependencies, which is consistent with the analysis performed in Section 3.4.1. The linear runtime performance characteristics of PARIGE algorithm makes it suitable for dynamically deploying operational strings online at runtime because the deployment latency of all operational strings exhibits a *linear* time

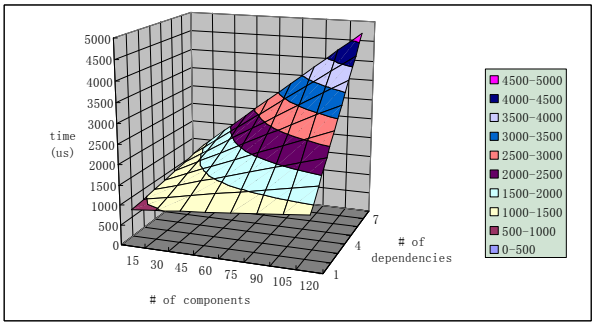

Figure 18: Running Time of the PARIGE Algorithm

complexity to the number of components in the operational strings.

As long as the performance overhead of the PARIGE algorithm is acceptable to deploy one component, therefore, it should be acceptable to deploy any number of components. To validate this claim, we conducted an experiment that deployed up to 64 operational strings with `960` total components. The results in Figure 19 shows that the deployment latency of all operational strings with and without the PARIGE algorithm. The experiment measures different number of operational strings and different number of components, ranging from `1` operational string with `15` components to `64` operational strings with `960` components. These results show that the actual per-

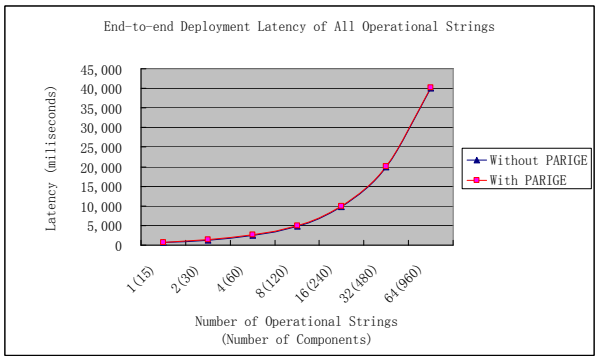

Figure 19: Performance Overhead of the PARIGE Algorithm

formance overhead of PARIGE algorithm for our experiment is consistently less than ∼1%, which further validates our earlier analysis.

5. Related Work

This section compares our work on the PARIGE algorithm with related work. We classify the related work

into two categories. The first category includes formal models and using various formal method techniques to ensure consistency and dependency correctness issues of software deployment. The second category includes related work that addresses various QoS issues with software deployment.

5.1. Dependency Management Models

Much prior research has been conducted on dependency management of software components. Most literature in this area analyzes deployment installability via formal models that explicitly express dependencies of software components or software packages. For example, [14] present a model to formalize deployment dependencies of software components. These dependencies are expressed in a logical language associated with a D&C framework that allows proving properties (such as whether the dependency is mandatory, optional, or negative) of the deployment plan. [15] defines an *application buildbox* as a software deployed environment and defines a formal Labeled Transition System (LTS) on the buildbox with transitions for deployment operations that include build, install, ship, and update. Formal properties of the LTS are introduced, including version dependency and software component dependencies. This Prior work, however, does not address deployment predictability, which makes it unsuited for DRE systems.

Some software frameworks have been developed to support software component deployment or redeployment. For examples, [16] and [17] uses framework-guided reconfiguration mechanism for component-based distributed systems. These frameworks offer mechanisms to analyze dependencies between peer components to deal with reconfiguration consistency, which is similar to our work. Unlike our approach, however, this work does not consider how to improve the predictability of the deployment process, but instead focuses only on consistency.

5.2. QoS Assurance of Software Deployment

Prior related work has focused on improving the QoS of software D&C frameworks. Existing literature in this area focuses on performance related issue, but ignores predictability related issues, which are important to ensure DRE systems QoS. For example, The work reported in [18] relies on pre-allocating resource (*e.g.*, pre-load components in .NET dynamic reconfiguration framework) to make the D&C process of components more predictable. Unlike our approach, however, [18] is only concerned with how to improve the response time within a single deployment unit, rather than of managing multiple deployment units at the same time.

The approach described in [19] uses a virtual machine (VM) technique that provides automated D&C of flexible VMs that can be configured to meet application needs and then subsequently cloned and dynamically instantiated to improve the predictability of deployments. This approach supports a graph-based model for the definition of customized VM configuration actions. Our work on PARIGE is complementary to this work and can further improve the predictability of component deployment by differentiating services of multiple operational strings.

5.3. Comparison between PARIGE algorithm and Priority Inheritance Protocol

The PARIGE algorithm has many similarities to the priority inheritance protocol [20] used for synchronization in real-time systems. The priority inheritance protocol ensures that when a thread blocks one or more high priority threads, it executes its critical section at the highest priority level of all the threads it blocks, *i.e.*, it inherits the highest threads priority. After executing its critical section, the thread returns to its original priority level.

In the PARIGE algorithm lower priority operational strings execute at (*i.e.*, "inherit") the priority of higher priority operational strings to avoid deployment priority inversions. The "critical section" in the priority inheritance protocol is thus similar to the "deployment and configuration" activities in the PARIGE algorithm. Our work differs from the priority inheritance protocol in several ways, however. First, the priority inheritance protocol does not avoid deadlocks, but PARIGE algorithm does not have such limitation because the algorithm removes all priority inverted dependencies between operational strings and then deploy operational strings from the highest priority to the lowest priority sequentially. Second, only part of the operational string is affected, *i.e.*, the PARIGE algorithm just increases the deployment priorities of components with dependencies from higher priority operational strings. Third, the PARIGE algorithm is much more complex than the priority inheritance protocol because it needs to traverses multiple graphs to identify which components require promotion.

6. Concluding Remarks

The predictability and scalability of D&C frameworks is essential to support the QoS requirements of open DRE systems. This paper describes a multi-graph algorithm that helps ensure the predictability of

deploying multiple operational strings. We first analyze how deployment priority inversion can occur when operational strings have various dependency relationships. We then empirically show how the *partial priority inheritance via graph recomposition* (PARIGE) algorithm can effectively avoid deployment priority inversions and thus improve the predictability of component deployment in DRE systems.

The following summarizes our lessons learned thus far from developing and applying the PARIGE algorithm to ensure the predictability of deployment of operational strings in DRE systems:

1. The overlap of deployment-time with runtime makes D&C frameworks essential to ensure system QoS. The benefits provided by component middleware significantly change the lifecycle of DRE system development. Due to the complexities of open DRE systems, D&C frameworks assume more responsibilities to ensure system QoS because deployment of system services/components occurs throughout the lifecycle of the systems. By using information available at deployment time, D&C frameworks can effectively identify the complex dependency relationships among operational strings and perform various on-line optimizations, such as the operational string recomposition presented in Section 3.

2. Automated Recomposition of operational strings can help ensure deployment predictability of DRE systems. Although operational strings can simplify the design of DRE systems, it is hard to manually ensure deployment predictability of all operational strings due to the complex dependencies among many operational strings. The PARIGE algorithm presented in this paper enhances the deployment predictability of different operational strings by recomposing operational strings automatically based on the input to the D&C framework and transparently to system deployers.

3. Recomposition of operational strings by the PARIGE algorithm is orthogonal to later management of operational strings. As described in Section 1, component-based DRE systems are often designed, deployed and managed in the form of operational strings that are first class entities whose lifecycles are managed by D&C frameworks. Although our PARIGE algorithm recomposes operational strings and modifies their topologies to avoid deployment priority inversion, it does not change the behavior of the operational strings once the algorithm-modified operational strings are deployed. In particular, the D&C framework's `ExecutionManager` only recomposes operational strings for its initial deployment, but once deployed the metadata descriptors for the operational strings are still the original ones without any modification. Subsequent management of operational strings will thus not be affected by the PARIGE algorithm.

4. The effectiveness of the PARIGE algorithm depends on operational string characteristics and their dependencies. The PARIGE algorithm can improve deployment predictability of operational strings with negligible performance overhead, as demonstrated in Section 4.5. The effectiveness of the PARIGE algorithm varies for different configurations of operational strings and the external dependencies among them. As shown by the empirical results in Section 4.3, the PARIGE algorithm is most effectively when operational string growth is minimal and least effectively when growth is large. In the worst case scenario, however, when there exits only a single dependency trace across all the operational strings, all the operational strings are merged into a single operational string. In this case, no predictability improvement can be made by the PARIGE algorithm. Since the performance overhead of the PARIGE algorithm is negligible, however, the PARIGE algorithm based D&C approach will at least performs approximately the same as without the algorithm being applied. Our future work therefore will investigate how to quantify the gain of the PARIGE algorithm by measuring the operational string growth effect and the deployment cost of different components based on the input of the operational strings.

5. Advanced OS and middleware features are important complements to the PARIGE algorithm. The PARIGE algorithm can incur certain effects when recomposing operational strings, such as the *component host distribution effect* discussed in Section 3.4. Our experience shows that modifying the multi-graph based PARIGE algorithm itself alone is insufficient to address this undesired effect because the algorithm introduces constraints on host collocation/-distribution, which affects its performance. One way to alleviate this problem is to apply asynchronous method invocations, as presented in Section 4.3.

Our future work will also determine whether/how the results from the PARIGE algorithm runs can provide feedback to system designers. For example, the D&C framework can analyze the input and output to the PARIGE algorithm for each deployment request. Using this historical input/output information, a D&C framework can potentially identify those operational strings responsible for most deployment priority inversion. We conjecture that this approach will help improve DRE system D&C by reorganizing operational strings more effectively.

The PARIGE algorithm is an integral part

of DAnCE, and both DAnCE and CIAO are open-source and available for download at `www.dre.vanderbilt.edu/ciao`.

References

[1] Gill, C., Gossett, J., Loyall, J., Schmidt, D., Corman, D., Schantz, R., Atighetchi, M.: Integrated Adaptive QoS Management in Middleware: A Case Study. In: Proceedings of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS), Toronto, Canada, IEEE (May 2004)

[2] Schmidt, D.C., Schantz, R., Masters, M., Cross, J., Sharp, D., DiPalma, L.: Towards Adaptive and Reflective Middleware for Network-Centric Combat Systems. CrossTalk - The Journal of Defense Software Engineering (November 2001)

[3] Sharma, P., Loyall, J., Heineman, G., Schantz, R., Shapiro, R., Duzan, G.: Component-Based Dynamic QoS Adaptations in Distributed Real-time and Embedded Systems. In: Proc. of the Intl. Symp. on Dist. Objects and Applications (DOA'04), Agia Napa, Cyprus (October 2004)

[4] Lardieri, P., Balasubramanian, J., Schmidt, D.C., Thaker, G., Gokhale, A., Damiano, T.: A Multi-layered Resource Management Framework for Dynamic Resource Management in Enterprise DRE Systems. Journal of Systems and Software: Special Issue on Dynamic Resource Management in Distributed Real-time Systems **80**(7) (July 2007) 984–996

[5] Desertot, M., Cervantes, H., Donsez, D.: Frogi: Fractal components deployment over osgi. In: Software Composition. (2006) 275–290

[6] Quéma, V., Balter, R., Bellissard, L., Féliot, D., Freyssinet, A., Lacourte, S.: Asynchronous, hierarchical, and scalable deployment of component-based applications. In: Component Deployment, Second International Working Conference, CD 2004, Edinburgh, UK, May 20-21, 2004, Proceedings. Volume 3083 of Lecture Notes in Computer Science., Springer (2004) 50–64

[7] Deng, G., Balasubramanian, J., Otte, W., Schmidt, D.C., Gokhale, A.: DAnCE: A QoS-enabled Component Deployment and Configuration Engine. In: Proc. of the 3rd Working Conf. on Component Deployment (CD 2005), Grenoble, France, Springer-Verlag (November 2005) 67–82

[8] Suri, D., Howell, A., Schmidt, D.C., Biswas, G., Kinnebrew, J., Otte, W., Shankaran, N.: A Multi-agent Architecture for Smart Sensing in the NASA Sensor Web. In: Proceedings of the 2007 IEEE Aerospace Conference, Big Sky, Montana (March 2007)

[9] Rigole, P., Clerckx, T., Berbers, Y., Coninx, K.: Possibilities to improve ground-based cloud cover observations using satellite application facility (safnwc) products. GEOGRAFIJA. **43**(1) (2007) 21–29

[10] Object Management Group: Deployment and Configuration Adopted Submission. OMG Document mars/03-05-08 edn. (July 2003)

[11] Bettati, R., Liu, J.W.S.: End-to-end scheduling to meet deadlines in distributed systems. In: International Conference on Distributed Computing Systems. (1992) 452–459

[12] Schmidt, D.C., Vinoski, S.: Programming Asynchronous Method Invocations with CORBA Messaging. C++ Report **11**(2) (February 1999)

[13] Wang, N., Gill, C., Schmidt, D.C., Subramonian, V.: Configuring Real-time Aspects in Component Middleware. In: Proc. of the International Symposium on Distributed Objects and Applications (DOA'04). Volume 3291., Agia Napa, Cyprus, Springer-Verlag (October 2004) 1520–1537

[14] Belguidoum, M., Dagnat, F.: Dependency management in software component deployment. Electron. Notes Theor. Comput. Sci. **182** (2007) 17–32

[15] Liu, Y.D., Smith, S.F.: A formal framework for component deployment. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM (2006) 325–344

[16] Chen, X., Simons, M.: A Component Framework for Dynamic Reconfiguration of Distributed Systems. In: CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment, London, UK, Springer-Verlag (2002) 82–96

[17] Kon, K., Campbell, R.: Dependence Management in Component-based Distributed Systems. IEEE Concurrency **8**(1) (2000) 26–36

[18] Rasche, A., Polze, A.: Configuration and dynamic reconfiguration of component-based applications with microsoft .net. In: ISORC '03: Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03), Washington, DC, USA, IEEE Computer Society (2003) 164

[19] Krsul, I., Ganguly, A., Zhang, J., Fortes, J.A.B., Figueiredo, R.J.: Vmplants: Providing and managing virtual machine execution environments for grid computing. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2004) 7

[20] Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority Inheritance Protocols: An Approach to Real-time Synchronization. IEEE Transactions on Computers **39**(9) (September 1990)