A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems [⋆]

Krishnakumar Balasubramanian [a,∗],
Jaiganesh Balasubramanian [a], Jeff Parsons [a],
Aniruddha Gokhale [a], Douglas C. Schmidt [a]

[a] *Dept. of EECS, Vanderbilt University, Nashville*

Abstract

This paper provides two contributions to the study of developing and applying domain-specific modeling languages (DSMLS) to distributed real-time and embedded (DRE) systems – particularly those systems using standards-based QoS-enabled component middleware. First, it describes the Platform-Independent Component Modeling Language (PICML), which is a DSML that enables developers to define component interfaces, QoS parameters and software building rules, and also generates descriptor files that facilitate system deployment. Second, it applies PICML to an unmanned air vehicle (UAV) application portion of an emergency response system to show how PICML resolves key component-based DRE system development challenges. Our results show that the capabilities provided by PICML – combined with its design- and deployment-time validation capabilities – eliminates many common errors associated with conventional techniques, thereby increasing the effectiveness of applying QoS-enabled component middleware technologies to the DRE system domain.

Key words: CoSMIC, Model-driven Development, Real-time CORBA Component Model

[⋆] This work was sponsored in part by AFRL Contract# F33615-03-C-4112 for DARPA PCES Program, Raytheon, and a grant from Siemens CT.
[∗] Corresponding author E-mail: kitty@dre.vanderbilt.edu

1 Introduction

Emerging trends and challenges Reusable components and standards-based component models are increasingly replacing the use of monolithic and proprietary technologies as the platform for developing large-scale, mission-critical distributed real-time and embedded (DRE) systems [1]. This paradigm shift is motivated by the need to (1) reduce life-cycle costs by leveraging standards-based and commercial-off-the-shelf (COTS) technologies and (2) enhance software quality by amortizing validation and optimization efforts over many users and testing cycles. Component technologies, such as the OMG's Lightweight CORBA Component Model (CCM) and Boeing's Boldstroke PRiSm, are establishing themselves as effective middleware platforms for developing component-based DRE software systems in domains ranging from software-defined radio to avionics mission computing and total ship computing environments.

The trend towards developing and reasoning about DRE systems via components provides many advantages compared with earlier forms of infrastructure software. For example, components provide higher-level abstractions than operating systems, third-generation programming languages, and earlier generations of middleware, such as distributed object computing (DOC) middleware. In particular, component middleware, such as CCM, J2EE, and .NET, supports multiple views per component, transparent navigation, greater extensibility, and a higher-level execution environment based on containers, which alleviate many limitations of prior middleware technologies. The additional capabilities of component-based platforms, however, also introduce new complexities associated with composing and deploying DRE systems using components, including (1) the need to design consistent component interface definitions, (2) the need to specify valid interactions and connections between components, (3) the need to generate valid component deployment descriptors, (4) the need to ensure that requirements of components are met by target nodes where components are deployed, and (5) the need to guarantee that changes to a system do not leave it in an inconsistent state. The lack of simplification and automation in resolving the challenges outlined above can significantly hinder the effective transition to – and adoption of – component middleware technology to develop DRE systems.

Solution approach → Model-driven development of component-based DRE systems To address the needs of DRE system developers outlined above, we have developed the *Platform-Independent Component Modeling Language* (PICML). PICML is an open-source domain-specific modeling language (DSML) available for download at `www.dre.vanderbilt.edu/cosmic` that enables developers of component-based DRE systems to define application interfaces, QoS parameters, and system software building rules, as well as generate valid XML descriptor files that enable automated system deployment. PICML also provides capabilities to handle complex component engineering tasks, such as multi-aspect visualization of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling of component assemblies.

PICML is designed to help bridge the gap between design-time verification and model-checking tools (such as Cadena, VEST, and AIRES) and the actual deployed component implementations. PICML also provides higher-level abstractions for describing DRE systems, using component models that provides a base for (1) integrating analysis tools that reason about DRE systems and (2) platform-independent generation capabilities, *i.e.*, generation that can be targeted at multiple component middleware technologies, such as CCM, J2EE, and ICE.

2 Overview of PICML

Model-Driven Development (MDD) [2] is a paradigm that focuses on using models in most system development activities, *i.e.*, models provide input and output at all stages of system development until the final system itself is generated. A key capability supported by the MDD paradigm is the definition and implementation of *domain-specific modeling languages* (DSMLs), which can be viewed as a five-tuple [3] consisting of: (1) concrete syntax (C), which defines the notation used to express domain entities, (2) abstract syntax (A), which defines the concepts, relationships and integrity constraints available in the language, (3) semantic domain (S), which defines the formalism used to map the semantics of the models to a particular domain, (4) syntactic mapping (M_C: A→C), which assigns syntactic constructs (*e.g.*, graphical and/-or textual) to elements of the abstract syntax, and (5) semantic mapping (M_S: A→S), which relates the syntactic concepts to those of the semantic domain.

Crucial to the success of DSMLs is *metamodeling* and *auto-generation.* A *metamodel* defines the elements of a DSML, which is tailored to a particular domain, such as the domain of avionics mission computing or emergency response systems. Auto-generation involves automatically synthesizing artifacts from models, thereby relieving DSML users from the specifics of the artifacts themselves, including their format, syntax, or semantics. Examples of such artifacts includes (but are not limited to), code in some programming language and/or descriptors, in formats such as XML, that can serve as input to other tools.

To support development of DRE systems using MDD, we have defined the

Platform-Independent Component Modeling Language (PICML) DSML using the *Generic Modeling Environment* (GME) [4]. GME is a meta-programmable modeling environment with a general-purpose editing engine, separate view-controller GUI, and a configurable persistence engine. Since GME is meta-programmable, the same environment used to define PICML is also used to build models, which are instances of the PICML metamodel.

At the core of PICML is a DSML (defined as a *metamodel* using GME) for describing components, types of allowed interconnections between components, and types of component metadata for deployment. The PICML metamodel defines ∼115 different types of basic elements, with 57 different types of associations between these elements, grouped under 14 different folders. The PICML metamodel also uses the OMG's Object Constraint Language (OCL) to define ∼222 constraints that are enforced by GME's constraint manager during the design process.

Using GME tools, the PICML metamodel can be compiled into a *modeling paradigm*, which defines a domain-specific modeling environment. From this metamodel, ∼20,000 lines of C++ code (which represents the modeling language elements as equivalent C++ types) is generated. This generated code allows manipulation of modeling elements, *i.e.*, instances of the language types using C++, and forms the basis for writing *model interpreters*, which traverse the model hierarchy to perform various kinds of generative actions, such as generating XML-based deployment plan descriptors. PICML currently has ∼8 interpreters using ∼222 generated C++ classes and ∼8,000 lines of hand-written C++ code that traverse models to generate the XML deployment descriptors (described in Sidebar 1) needed to support the OMG D&C specification [5]. Each interpreter is written as a DLL that is loaded at run-time into GME and executed to generate the XML descriptors based on models developed by the component developers using PICML.

To motivate and explain the features in PICML, we use a running example of a representative DRE system designed for emergency response situations (such as disaster recovery efforts stemming from floods, earthquakes, hurricanes) and consists of a number of interacting subsystems with a variety of DRE QoS requirements. Our focus in this paper is on the unmanned aerial vehicle (UAV) portion of this system, which is used to monitor terrain for flood damage, spot survivors that need to be rescued, and assess the extent of damage. The UAV transmits this imagery to various other emergency response units, including the national guard, law enforcement agencies, health care systems, firefighting units, and utility companies.

3 Building DRE Systems with PICML

Developing and deploying emergency response systems is hard. For example, there are multiple modes of operations for the UAVs, including aerial imaging, survivor tracking, and damage assessment. Each of these modes is associated with a different set of QoS requirements. For example, a key QoS criteria involves the latency requirements in sending images from the flying UAVs to ground stations under varying bandwidth availability. Similar QoS requirements manifest themselves in the traffic management, rescue missions, and fire fighting operations.

In conjunction with colleagues at BBN Technologies and Washington University, we have developed a prototype of the UAV portion of the emergency response system [6] described above using the CCM and Real-time CORBA capabilities provided by CIAO [7]. CIAO extends our previous work on *The ACE ORB* (TAO) [8] by providing more powerful component-based abstractions using the specification, validation, packaging, configuration, and deployment techniques defined by the OMG CCM [9] and D&C [5] specifications. Moreover, CIAO integrates the CCM capabilities outlined below with TAO's Real-time CORBA [8] features, such as thread-pools, lanes, and client-propagated and server-declared policies. In this section, we first briefly explain the key capabilities of CCM, and the motivation for using CCM to develop the UAV portion of the emergency response system. We then describe the challenges in developing this system using CCM, and then show how we resolved these challenges by applying a MDD approach using PICML.

3.1 Key Capabilities of the CCM

The CORBA Component Model (CCM) is an OMG specification that standardizes the development of component-based applications in CORBA. Since CCM uses CORBA's object model as its underlying object model, developers are not tied to any particular language or platform for their component implementations. The CIAO project is based on CCM rather than other popular component models, such as EJB or .NET, since CORBA is the only COTS middleware that has made a substantial progress in satisfying the QoS requirements of DRE applications. For instance, the OMG has adopted the following DRE-related specifications in recent several years:

- **Minimum CORBA**, which removes non-essential features from the full OMG CORBA specification to reduce footprint so that CORBA can be used in memory-constrained embedded system applications.

- **Real-time CORBA**, which includes features that allow applications to reserve and manage network, CPU, and memory resources predictably end-to-end.
- **CORBA Messaging**, which exports additional QoS policies, such as asynchronous invocations, timeouts, request priorities, and queuing disciplines, to DRE applications.
- **Fault-tolerant CORBA**, which uses entity redundancy of objects to support replication, fault detection, and failure recovery.

These QoS specification and enforcement capabilities are essential to support DRE applications. Key elements of the CCM include:

- **Component**, which is the basic building block used to encapsulate application functionality
- **Component Home**, which is a factory that creates and manages components
- **Container**, which provides components with an abstraction of the underlying middleware and regulate their shared access to the middleware infrastructure,
- **Component Implementation Framework**, which defines the programming model for constructing component implementations, using the Component Implementation Definition Language (CIDL) descriptions for automating generation of programming skeletons,
- **Component server**, which groups components and containers together to form an executable program.
- **ORB Services**, which provide common middleware services, such as transaction, events, security and persistence.

The components in this UAV application are shown in Figure 1 and the steps involved in this effort are described below:

1. Identify the components in the system, and define their interfaces, which involves defining component ports and attributes, using the CORBA 3.x IDL features provided by CCM. In the UAV example, each UAV is associated with a stream of images. Each image stream is composed of `Sender`, `Qosket`, and `Receiver` components. `Sender` components are responsible for collecting the images from each image sensor on the UAV. The `Sender` passes the images to a series of `Qosket` [7] components that perform operations on the images to ensure that the QoS requirements are satisfied. Some Qosket components include `CompressQosket, ScaleQosket, CropQosket, PaceQosket`, and a `DiffServQosket`. The final `Qosket` then passes the images to a `Receiver` component, which collects the images from the UAV and passes them on to a display in the control room of the emergency response team.

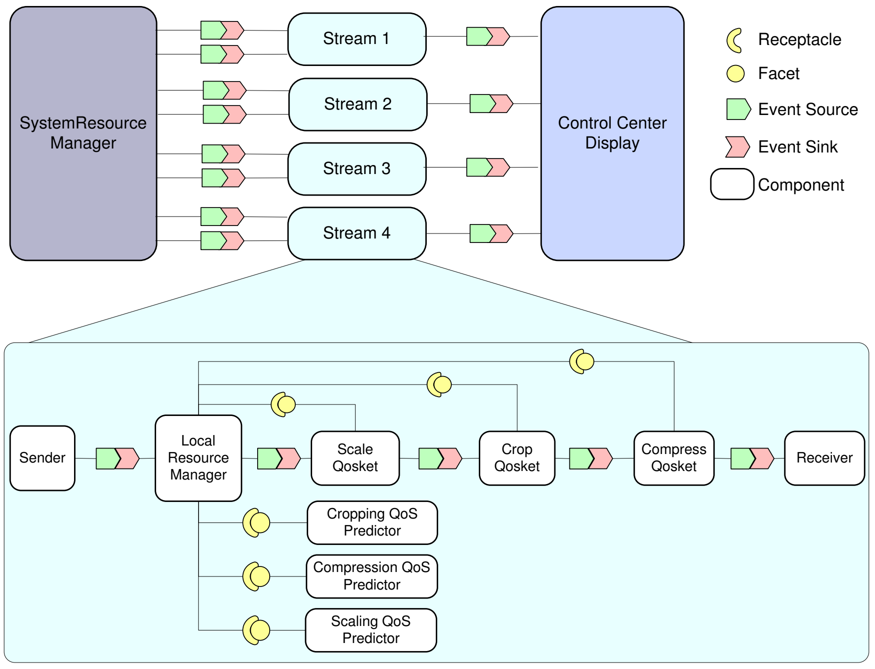

Fig. 1. Emergency Response System components

Each `Sender`, `Receiver`, and the various `Qosket` components pass images via CCM event source and sink ports. There are also manager components that define policies, such as the relative importance of the different mission modes of each UAV. These policies in turn modify existing resource allocations by the `Qosket` components. For example, the global `SystemResourceManager` component monitors resource allocation across all the UAVs that are operational at any moment, and is responsible for communicating policy decisions from the control center to each UAV by triggering mode changes. The per-stream `LocalResourceManager` component is responsible for instructing the `Qosket` components to adapt their internal QoS requirements according to the mode in which the UAV is currently operating.

2. Define interactions between components, which involves keeping track of the types of each component's ports and ensuring that components which must be interconnected have matching ports defined. In the UAV example, this involves connecting the different components that comprise a single stream in the correct order since some components (such as `DeCompressQosket`) do the reverse of an operation performed by another component (such as `CompressQosket`). The manager components need to be connected to receive monitoring information about the existing QoS in each stream of image data.

3. Compose the UAV application by defining CCM deployment descriptors, which involves selecting a set of component implementations from a library of available implementations, describing how to instantiate component instances using these component implementations, and specifying connections between component instances. In the UAV example, this first involves combining the different components that comprise a single stream of images into a single assembly, represented by an XML descriptor. The complete UAV application is then created by making copies of this file to represent each UAV in flight.

4. Deploy the UAV application onto its runtime platform, which involves ensuring that the implementation artifacts and the associated deployment descriptors are available on the actual target platform, and initiating the deployment process using the standard OMG D&C [5] framework and tools. In the UAV example, this involves taking the hand-written XML descriptors and deploying the application using these descriptors as input.

5. Refine the component-based UAV application, which involves making changes to existing component interface definitions or adding new component types, as part of enhancing the initial UAV application prototype. In the UAV example, this involves adding or removing a `Qosket` component in the pipeline for a single stream depending on results from empirical evaluation of the system.

One of the challenges of using just component middleware is that errors often go undetected until late in the development cycle. When these errors are eventually detected, moreover, repairing them often involves backtracking to multiple prior life-cycle steps, which impedes productivity and increases the level of effort. As a result, the advantages of transitioning from DOC middleware to component middleware can be significantly obstructed, without support from higher-level tools and techniques. These observations underscore the importance of enhancing design-time support for DRE systems built using component middleware, as well as the importance of automating the deployment of such systems.

3.2 Resolving the UAV Application Challenges with PICML

As discussed in [10], the use of QoS-enabled component middleware to develop the UAV application significantly improved upon an earlier DOC middleware prototype of this application [11]. In the absence of model-driven development (MDD) tool support, however, a number of significant challenges remain unresolved when using component middleware. For concreteness, the remainder of this section describes five key challenges that arose when the UAV application was developed using CCM and CIAO, and examines how key features of PICML can be applied to address the limitations associated with developing QoS-enabled component middleware-based DRE systems, such as the UAV application.

We use CCM and CIAO as the basis for our research because it is layered on top of Real-time CORBA, which provides significant capabilities for satisfying end-to-end QoS requirements of DRE systems [8]. There is nothing inherent in PICML, however, that limits it to CCM or CIAO. Likewise, the challenges described below are generic to component middleware, and not deficiencies of CCM or CIAO. For example, both J2EE and Microsoft .NET use XML to describe component assemblies, so the challenges we describe apply to them, as well.

3.2.1 Accidental Complexities in Component Interface Definition.

IDL for CCM (*i.e.*, CORBA 3.x IDL) defines extensions to the syntax and semantics of CORBA 2.x IDL. Every developer of CCM-based applications must therefore master the differences between CORBA 2.x IDL and CORBA 3.x IDL. For example, while CORBA 2.x interfaces can have multiple inheritance, CCM components can have only a single parent, so equivalent units of composition (*i.e.*, interfaces in CORBA 2.x and components in CCM) can have subtle semantic differences. Moreover, any component interface that needs to be accessed by component-unaware CORBA clients should be defined as a *supported* interface as opposed to a *provided* interface.

In any system that transitions from an object-based architecture to a component-based architecture, there is likelihood of simultaneous existence of simple CORBA objects and more sophisticated CCM components. Design of component interfaces must therefore be done with extra care. In the UAV application, for example, though the `Qosket` components receive both allocation events from the resource managers and images from the `Sender` and other `Qosket` components, they cannot inherit from base components implementing each functionality. Similarly, the `Receiver` component interface needs to be defined as a *supported* interface, rather than a *provided* interface.

3.2.2 Solution → Visual Component Interface Definition.

A set of component, interface, and other datatype definitions may be created in PICML using either of the following approaches:

- **Adding to existing definitions imported from IDL**. In this approach, existing CORBA software systems can be easily migrated to PICML using its *IDL Importer*, which takes any number of CORBA IDL files as input, maps their contents to the appropriate PICML model elements, and generates a single XML file that can be imported into GME as a PICML model. This model can then be used as a starting point for modeling assemblies and generating deployment descriptors.

- **Creating IDL definitions from scratch**. In this approach, PICML's graphical modeling environment provides support for designing the interfaces using an intuitive "drag and drop" technique, making this process largely self-explanatory and independent of platform-specific technical knowledge. Most of the grammatical details are implicit in the visual language, *e.g.*, when the model editor screen is showing the "scope" of a definition, only icons representing legal members of that scope will be available for dragging and dropping.

CORBA IDL can be generated from PICML, enabling generation of software artifacts in languages having a CORBA IDL mapping. For each logically separate definition in PICML, the generated IDL is also split into logical file-type units. PICML's interpreter will translate these units into actual IDL files with #include statements based on the inter-dependencies of the units detected by the interpreter. PICML's interpreter will also detect requirements for the inclusion of canonical CORBA IDL files and generate them as necessary.

Application to the UAV example scenario. By modeling the UAV components using PICML, the problems associated with multiple inheritance, semantics of IDL, etc. are flagged at design time. By providing a visual environment for defining the interfaces, PICML therefore resolves many problems described in Section 3.2.1 associated with definition of component interfaces. In particular, by modeling the interface definitions, PICML alleviates the need to model a subset of interfaces for analysis purposes, which has the added advantage of preventing skew between the models of interfaces used by analysis tools and the interface used in implementations. It also removes the effort needed to ensure that the IDL semantics are satisfied, resulting in a ∼50% reduction in effort associated with interface definition.

3.2.3 Defining Consistent Component Interactions.

Even if a DRE system developer is well-versed in CORBA 3.x IDL, it is hard to keep track of components and their types using plain IDL files, which are text-based and hence provide no visual feedback, *i.e.*, to allow visual comparison to identify differences between components. Type checking with text-based files involves manual inspection, which is error-prone and non-scalable. CCM defines the following valid interactions between the *ports* — Facets, Receptacles, Event Sources and Event Sinks — of a component: Facet-Receptacle interactions, and Event Source-Event Sink interactions. However, an IDL compiler will not be able to catch mismatches in the port types of two components that need to be connected together, since component connection information is not defined in IDL. This problem only becomes worse as the number of component types in a DRE system increases. In our UAV application for example, enhancing the UAV with new capabilities can increase the number of component types and inter-component interactions. If a problem arises, developers of DRE systems may need to revise the interface definitions until the types match, which is a tedious and error-prone process.

3.2.4 Solution → Semantically Compatible Component Interaction Definition.

PICML defines the *static semantics* of a system using a constraint language and enforces these semantics early in the development cycle, *i.e.*, at design-time. This type checking can help identify system configuration errors similar to how a compiler catches syntactic errors early in the programming cycle. Static semantics refer to the "well-formedness" rules of the language. The well-formedness rules of a traditional compiler are nearly always based on a language grammar defining valid syntax. By elevating the level of abstraction via MDD techniques, however, the corresponding well-formedness rules of DSMLs like PICML actually capture semantic information, such as constraints on composition of models, and constraints on allowed interactions.

```
let facets = self.connectedFCOs(invoke) in
  facets->forAll ( i : ProvidedRequestPort |
    let supertypes = i.refersTo().oclAsType(gme::Model).allParents(Set{}) in
     (supertypes->one (k: gme::FCO | k.name() = self.refersTo().name())
        or self.refersTo().name() = i.refersTo().name() ) )
```

Constraint Listing 1: A Receptacle should be connected to a matching Facet

There is a significant difference in the early detection of errors in the MDD paradigm compared with traditional object-oriented or procedural development using a conventional programming language compiler. In PICML, OCL constraints are used to define the static semantics of the modeling language, thereby disallowing invalid systems to be built using PICML. In other words, PICML enforces the paradigm of "correct-by-construction." For example, the constraint shown in Constraint Listing 1 is a PICML constraint, that checks that the type of a receptacle matches either the corresponding facet's type, or that the receptacle is a super type of the facet type. This is a good example of a constraint that ensures the type compatibility of components that are composed to form an assembly.

Constraints in PICML aren't necessarily restricted to type conformance. Constraint Listing 2 is an existential constraint, *i.e.*, it checks for the presence of an implementation corresponding to each *instance* of a component used in an assembly. Each component type may have different alternate implementations offering different QoS behavior, and the correct implementation is chosen at deployment time. Section 3.2.8 describes this process in greater detail. However, in order to select the correct component implementation at deployment time, it is critical that each instance of a component be associated with an implementation, and this constraint checks this invariant for every component instance in the model.

```
let instances = self.modelParts(Component) in
  let monolithicImpls = project.allInstancesOf (MonolithicImplementation) in
    instances->forAll (x : Component |
      let myType = x.ComponentParentType() in
        monolithicImpls->exists ( impl : MonolithicImplementation |
          let interfaces = impl.connectedFCOs(Implements) in
            interfaces->size() = 1 and
              interfaces->exists (interface : Reference |
                interface.refersTo().name() = myType.name() ) ) )
```

Constraint Listing 2: Every Component should have a corresponding implementation

Another example of a constraint in PICML is the ability to restrict the flow of information in a particular direction, *i.e.*, top-down or bottom-up. Attributes of components in CCM can have default values assigned to them in the implementation. Depending on the context, attributes of components can also have values propagated to them from outside. This propagation is done using an "attribute mapping", which is a mechanism to propagate the value of an attribute of a higher-order element like an assembly, to one or more attributes of one of more components inside the assembly. Constraint Listing 3 restricts this propagation of initial assignment to flow in a strictly top-down fashion, *i.e.*, to give a behavior that matches the behavior of turning of a top-level knob affecting the low-level knobs of a system.

```
let mappings = self.referenceParts (AttributeMapping) in
  let children = self.modelParts(ComponentAssembly) in
    mappings->forAll ( x : AttributeMapping |
    let delegates = x.connectedFCOs("dstAttributeMappingDelegate",
                                    AttributeMappingDelegate) in
      delegates->forAll ( y : FCO |
        let  delParent : Model = y.parent() in
          children->exists ( z : ComponentAssembly |
             delParent.name() = z.name() ) ) )
```

Constraint Listing 3: AttributeMappings can only be delegated from a high-level assembly to sub-assemblies, and not vice-versa

By using GME's constraint manager, PICML constraints can be (1) evaluated automatically (triggered by a specified modeling event such as attempting a connection between ports of two components) or on demand, (2) prioritized to control order of evaluation and severity of violation, and/or (3) applied globally or to one or more individual model elements

Application to the UAV example scenario. In the context of our UAV application, the components of a single stream can be modeled as a CCM assembly. PICML enables the visual inspection of types of ports of components and the connection between compatible ports, including flagging error when attempting connection between incompatible ports. For example, PICML will flag attempts to connect incompatible `LocalResourceManager` receptacles with the facets of the `Qoskets` in a stream in the UAV scenario. By constraining the direction of flow of information, PICML also ensures that the global policies that are set by the `SystemResourceManager` are honored by the `LocalResourceManager`s of the individual streams. PICML also differentiates different types of connections using visual cues, such as dotted lines and color, to quickly compare the structure of an assembly. By providing a visual environment coupled with rules defining valid constructs, PICML therefore resolves many problems described in Section 3.2.3 with ensuring consistent component interactions. By enforcing the constraints during creation of component models and interconnections — and by disallowing connections to be made between incompatible ports — PICML completely eliminates the manual effort required to perform these kinds of checks.

3.2.5 Generating Valid Deployment Descriptors.

Component developers must not only ensure type compatibility between interconnected component types as part of interface definition, but also ensure the same compatibility between instances of these component types in the XML descriptor files needed for deployment. This problem is of a larger scale than the one above, since the number of *component instances* typically dwarfs the number of *component types* in a large-scale DRE system. Moreover, a CCM assembly file written using XML is not well-suited to manual editing.

In addition to learning IDL, DRE system developers must also learn XML to compose component-based DRE systems. In our example UAV application, simply increasing the number of UAVs increases the number of component instances and hence the component interconnections. The increase in component interconnections is typically not linear with respect to increase in number of component instances. Any errors in this step are likely to go undetected until the deployment of the system at run-time.

3.2.6 Solution → Automatic Deployment Descriptor Generation.

In addition to ensuring design-time integrity of systems built using OCL constraints, PICML also generates the complete set of deployment descriptors that are needed as input to the component deployment mechanisms. The descriptors generated by PICML conform to the descriptors defined by the standard OMG D&C specification [5]. Sidebar 1 shows an example of the types of descriptors that are generated by PICML, with a brief explanation of the purpose of each type of descriptor.

Since the rules determining valid assemblies are encoded into PICML via its metamodel, and enforced using constraints, PICML ensures that the generated XML describes a valid system. Generation of XML is done in a programmatic fashion by writing a `Visitor` class that uses the Visitor pattern to traverse the elements of the model and generate XML. The generated XML descriptors also

Sidebar 1: Generating Deployment Metadata

PICML generates the following types of deployment descriptors based on the OMG D&C specification:

- **Component Interface Descriptor (.ccd)** — Describes the interfaces — ports, attributes of a single component.
- **Implementation Artifact Descriptor (.iad)** — Describes the implementation artifacts (e.g., DLLs, executables etc.) of a single component.
- **Component Implementation Descriptor (.cid)** — Describes a specific implementation of a component interface; also contains component inter-connection information.
- **Component Package Descriptor (.cpd)** — Describes multiple alternative implementations (e.g., for different OSes) of a single component.
- **Package Configuration Descriptor (.pcd)** — Describes a component package configured for a particular requirement.
- **Component Deployment Plan (.cdp)** — Plan which guides the run-time deployment.
- **Component Domain Descriptor (.cdd)** — Describes the deployment target *i.e.*, nodes, networks on which the components are to be deployed.

ensure that the names associated with instances are unique, so that individual component instances can be identified unambiguously at run-time.

Application to the UAV example scenario. In the context of the UAV application, the automated generation of deployment descriptors using PICML not only removes the burden of knowing XML from DRE system developers, it also ensures that the generated files are valid. Adding (or removing) components is as easy as dragging and dropping (or deleting) an element, making the necessary connections, and regenerating the descriptors, instead of hand-modifying the existing XML files as would be done without such tool support. This automation resolves many problems mentioned in Section 3.2.5, where the XML files were hand-written and modified manually in case of errors with the initial attempts.

For example, it is trivial to make the ∼100 connections in a graphical fashion using PICML, as opposed to hand-writing the XML. All the connections between components for the UAV application were made in a few hours, and the XML was then generated instantaneously, *i.e.* at the click of a button. In contrast, it required several days to write the same XML descriptors manually. PICML also has the added advantage of ensuring that the generated XML files are syntactically valid, which is a task that is very tedious and error-prone to perform manually.

3.2.7 Associating Components with the Deployment Target.

In component-based systems there is often a disconnect between software implementation related activities and the actual target system since (1) the software artifacts and the physical system are developed independently and (2) there is no way to associate these two entities using standard component middleware features. This disconnect typically results in failures at run-time due to the target environment lacking the capabilities to support the deployed component's requirements. These mismatches can also often be a source of missed optimization opportunities since knowledge of the target platform can help (1) optimizing component implementations, (2) selecting appropriate component implementations to be deployed and (3) customizing the middleware for the appropriate target environment. In our UAV application, components that reside on a single UAV can use collocation facilities provided by ORBs to eliminate unnecessary (de)marshaling. Without the ability to associate components with targets, errors due to incompatible component connections and incorrect XML descriptors are likely to show up only during actual deployment of the system.

3.2.8 Solution → Deployment Planning.

In order to satisfy multiple QoS requirements, DRE systems are often deployed in heterogeneous execution environments. To support such environments, component middleware strives to be largely independent of the specific target environment in which application components will be deployed. The goal is to satisfy the functional and systemic requirements of DRE systems by making appropriate deployment decisions that account for key properties of the target environment, and retain flexibility by not committing prematurely to physical resources.

To support these needs, PICML can be used to specify the target environment where the DRE system will be deployed, which includes defining: (1) **Nodes**, where the individual components and component packages are loaded and used to instantiate those components, (2) **Interconnects** among nodes, to which inter-component software connections are mapped, to allow the instantiated components to communicate, and (3) **Bridges** among interconnects, where interconnects provide a direct connection between nodes and bridges to provide routing capability between interconnects. Nodes, interconnects, and bridges collectively represent the target environment.

Once the target environment is specified via PICML, allocation of component instances onto nodes of the target environment can be performed. This activity is referred to as *component placement*, where systemic requirements of the components are matched with capabilities of the target environment and suitable allocation decisions are made. Allocation can either be: (1) **Static**, where the domain experts know the functional and QoS requirement of each of the components, as well as knowledge about the nodes of the target environment. In such a case, the job of the allocation is to create a deployment plan comprising the components→node mapping specified by the domain expert, or (2) **Dynamic**, where the domain expert specifies the constraints on allocation of resources at each node of the target environment, and the job of the allocation is to choose a suitable component→node mapping that meets both the functional and QoS requirement of each of the components, as well as the constraints on the allocation of resources.

PICML currently provides facilities for specifying static allocation of components. In order to compute the static deployment plan for the components and the component assemblies, PICML makes use of the following inputs specified in the model:

- **Component Implementation Capabilities.** Component middleware provide multiple implementations with different QoS characteristics for the same component interface. PICML provides mechanisms to annotate component implementations with capabilities at the modeling level.
- **Component Middleware Target Capabilities.** As described in Section 1, DRE systems are deployed in heterogeneous target environments, each of them exposing various capabilities like processing power, memory for the applications to be operated. PICML allows such capabilities to be specified while modeling the target environment in which the component middleware is going to be deployed.
- **Component Implementation Selection Requirements.** Different uses of the same component might need to perform under differing QoS requirements. PICML allows the system integrators to specify component implementation selection requirements with each use of a component in an assembly.

As shown in Figure 2, domain experts can visually map the components with the respective target nodes, as well as provide additional hints, such as whether the components need to be process-collocated or host-collocated, provided two components are deployed in the same target node. PICML generates a deployment plan which uses the component implementation capabilities, the target capabilities and the component implementation selection requirements, and generates the mapping of components to nodes. This deployment plan is then used by the CIAO run-time deployment engine to perform the actual deployment of components to nodes.

Application to the UAV example scenario. In the context of the UAV

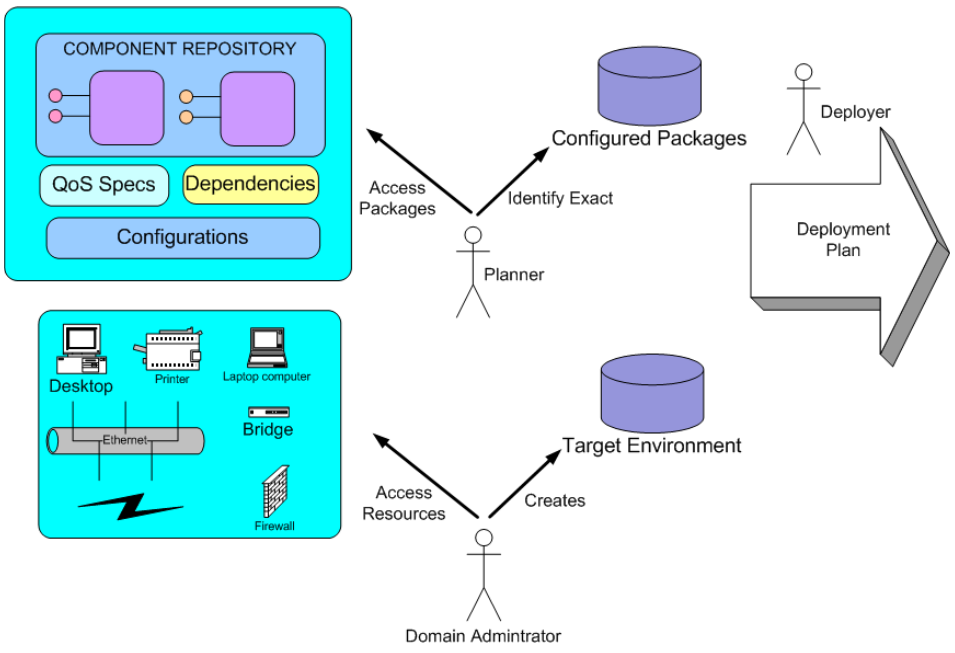

Fig. 2. Component Deployment Planning

example, PICML can be used to specify the mapping between the different `Qosket` components and the target environment, *i.e.*, the UAVs, in the path from each UAV to the `Receiver` component at the control center. By modeling the target environment in the UAV example using PICML, therefore, the problem with a disconnect between components and the deployment target described in Section 3.2.7 can be resolved. In case there are multiple possible component→node mappings, PICML can be used to experiment with different combinations since it generates descriptors automatically. PICML thus completely eliminates the manual effort involved in creating the deployment plan when there is a need to test different deployment scenarios.

3.2.9 Automating Propagation of Changes Throughout a DRE System.

Making changes to an existing component interface definition can be painful since it may involve retracing all the steps of the initial development. It also does not allow any automatic propagation of changes made in a base component type to other portions of the existing infrastructure, such as the component instances defined in the descriptors. Moreover, it is hard to test parts of the system incrementally, since it requires hand-editing of XML descriptors to remove or add components, thereby potentially introducing more problems. The validity of such changes can be ascertained only during deployment, which increases the time and effort required for the testing process. In our component-based UAV application, for example, changes to the basic composition of a single image stream are followed by laborious changes to each individual stream, impeding the benefits of reuse commonly associated with component-based development.

3.2.10 Solution → Hierarchical Composition.

In a complex DRE system with thousands of components, visualization becomes an issue because of the practical limitations of displays, and the limitations of human cognition. Without some form of support for hierarchical composition, observing and understanding system representations in a visual medium does not scale. To increase scalability, PICML defines a *hierarchy* construct, which enables the abstraction of certain details of a system into a hierarchical organization, such that developers can view their system at multiple levels of detail depending upon their needs.

The support for hierarchical composition in PICML not only allows DRE system developers to visualize their systems, but also allows them to compose systems from a set of smaller subsystems. This feature supports unlimited levels of hierarchy (constrained only by the physical memory of the system used to build models) and promotes the reuse of component assemblies. PICML therefore enables the development of repositories of predefined components and subsystems.

The hierarchical composition capabilities provided by PICML are only a *logical* abstraction, *i.e.*, deployment plans generated from PICML (described in 3.2.8) flatten out the hierarchy to connect the two destination ports directly (which if not done will introduce additional overhead in the communication paths between the two connected ports), thereby ensuring that at run-time there is no extra overhead that can be attributed to this abstraction. This feature extends the basic hierarchy feature in GME, which allows a user to double-click to view the contents of container objects called "models."

Application to the UAV example scenario. In the UAV example, the hierarchy abstraction in PICML allows the composition of components into a single stream assembly as shown in Figure 3, as well as the composition of multiple such assemblies into a top-level scenario assembly as shown in Figure 4.

As a result, large portions of the UAV application system can be built using reusable component assemblies. In turn, this increased reuse allows for automatic propagation of changes made to an subsystem to all portions of the system where this subsystem is used, resolving many problems mentioned in Section 3.2.9. PICML therefore helps prevent mismatches and removes duplication of subsystems.

Hierarchical assemblies in PICML also help reduce the effort involved in modeling of component assemblies by a factor of **N:1**, since N usages of a basic

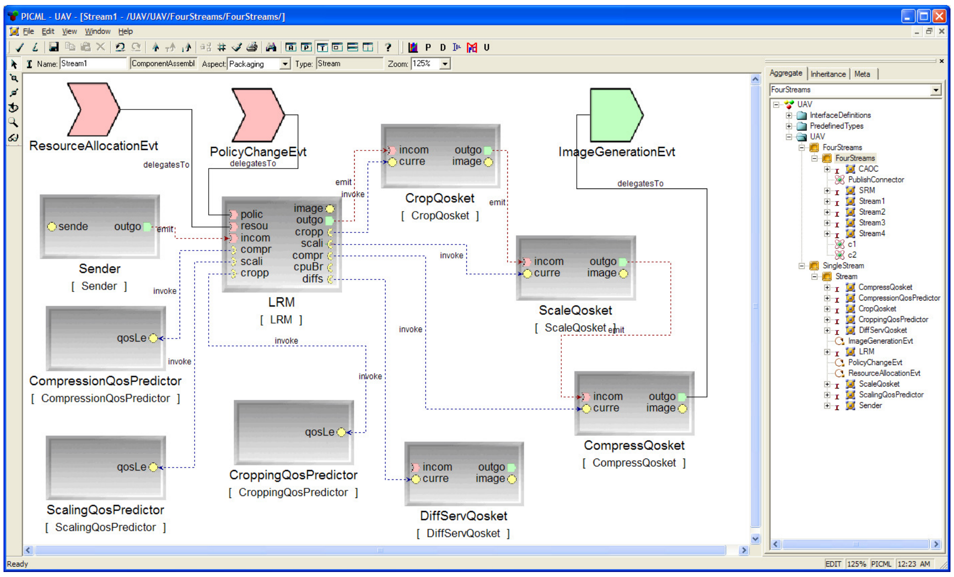

Fig. 3. Single Image Stream Assembly

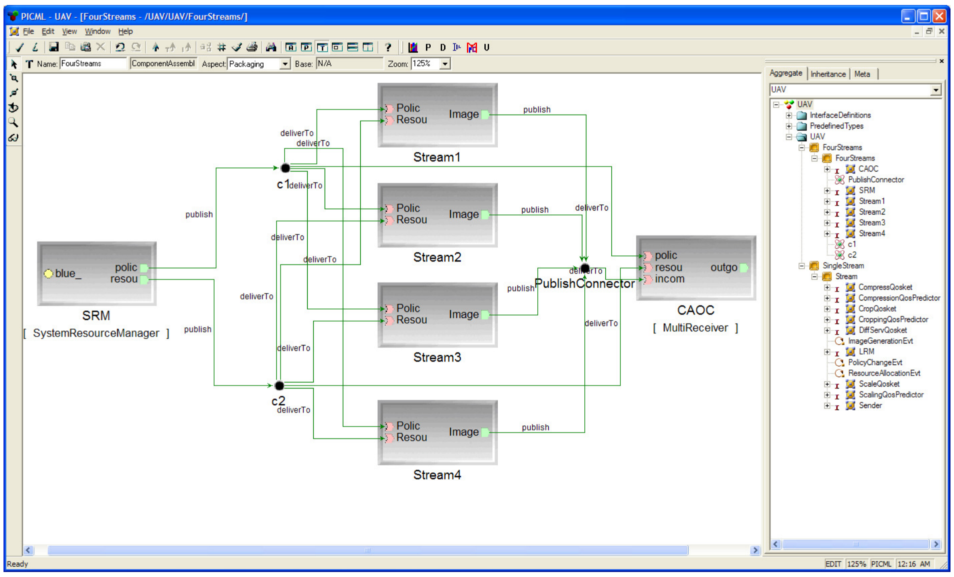

Fig. 4. UAV Application Assembly Scenario

assembly can be replaced with N instances of the same assembly, as well as providing for automatic generation of descriptors corresponding to the N instances. This technique was used to model a single stream of image from a UAV, and this single assembly was used to instantiate all the four streams of data, as shown in Figure 4.

4 Related Work

This section summarizes related efforts associated with developing DRE systems using an MDD approach and compares these efforts with our work on PICML.

Cadena Cadena [12] is an integrated environment developed at Kansas State University (KSU) for building and modeling component-based DRE systems, with the goal of applying static analysis, model-checking, and lightweight formal methods to enhance these systems. Cadena also provides a component assembly framework for visualizing and developing components and their connections. Unlike PICML, however, Cadena does not support activities such as component packaging and generating deployment descriptors, component deployment planning, and hierarchical modeling of component assemblies. To develop a complete MDD environment that seamlessly integrates component development and model checking capabilities, we are working with KSU to integrate PICML with Cadena's model checking tools, so we can accelerate the development and verification of DRE systems.

VEST and AIRES The *Virginia Embedded Systems Toolkit* (VEST) [13] and the *Automatic Integration of Reusable Embedded Systems* (AIRES) [14] are MDD analysis tools that evaluate whether certain timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. Components are selected from pre-defined libraries, annotations for desired real-time properties are added, the resulting code is mapped to a hardware platform, and real-time and schedulability analysis is done. In contrast, PICML allows component modelers to model the complete functionality of components and intra-component interactions, and doesn't rely on predefined libraries. PICML also allows DRE system developers the flexibility in defining the target platform, and is not restricted to just processors.

ESML The *Embedded Systems Modeling Language* (ESML) [15] was developed at the Institute for Software Integrated Systems (ISIS) to provide a visual metamodeling language based on GME that captures multiple views of embedded systems, allowing a diagrammatic specification of complex models. The modeling building blocks include software components, component interactions, hardware configurations, and scheduling policies. The user-created models can be fed to analysis tools (such as Cadena and AIRES) to perform schedulability and event analysis. Using these analyses, design decisions (such as component allocations to the target execution platform) can be performed. Unlike PICML, ESML is platform-specific since it is heavily tailored to the

Boeing Boldstroke PRiSm QoS-enabled component model [1,16]. ESML also does not support nested assemblies and the allocation of components are tied to processor boards, which is a proprietary feature of the Boldstroke component model. We are working with the ESML team at ISIS to integrate the ESML and PICML metamodels to produce a unified DSML suitable for modeling a broad range of QoS-enabled component models.

Ptolemy II Ptolemy II [17] is a tool-suite from the University of California Berkeley (UCB) that supports heterogeneous modeling, simulation, and design of concurrent systems using an actor-oriented design. Actors are similar to components, but their interactions are controlled by the semantics of models of computation, such as discrete systems. The set of available actors is limited to the domains that are natively defined in Ptolemy. Using an actor specialization framework, code is generated for embedded systems. In contrast, PICML does not define a particular model of computation. Also, since PICML is based on the metamodeling framework of GME, it can be customized to support a broader range of domains than those supported by Ptolemy II. Finally, PICML targets component middleware for DRE systems and can be used with any middleware technology, as well as any programming language, whereas Ptolemy II is based on Java, with preliminary support for C.

5 Concluding Remarks

Although component middleware represents an advance over previous generations of middleware technologies, its additional complexities threaten to negate many of its benefits without proper tool support. To address this problem, we describe the capabilities of the Platform-Independent Component Modeling Language (PICML) in this paper. PICML is a domain-specific modeling language (DSML) that simplifies and automates many activities associated with developing, and deploying component-based DRE systems. In particular, PICML provides a graphical DSML-based approach to define component interface definitions, specify component interactions, generate deployment descriptors, define elements of the target environment, associate components with these elements, and compose complex DRE systems from such basic systems in a hierarchical fashion.

To showcase how PICML helps resolve the complexities of QoS-enabled component middleware, we applied it to model key aspects of an unmanned air vehicle (UAV) application that is representative of emergency response systems. Using this application as a case study, we showed how PICML can support **design-time** activities, such as specifying component functionality, interactions with other components, and the assembly and packaging of components, and **deployment-time** activities, such as specification of target environment, and automatic deployment plan generation.

Acknowledgments

Portions of the work in this paper was done during Krishnakumar's internship at BBN Technologies, in Cambridge, MA. We would like to thank Prakash Manghwani, Matt Gillen, Praveen Sharma, Jianming Ye, Joe Loyall, Richard Schantz, and George Heineman at BBN for providing us with the component-based UAV application used as the motivating example in this paper. We would also like to thank Nanbor Wang, Venkita Subramonian, and Christopher Gill from Washington University for their efforts in implementing the original CIAO, and subsequent extensions that support Real-time CORBA features with CCM.

References

[1] D. C. Sharp, W. C. Roll, Model-Based Integration of Reusable Component-Based Avionics System, in: Proc. of the Workshop on Model-Driven Embedded Systems in RTAS 2003, 2003.

[2] J. Greenfield, K. Short, S. Cook, S. Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, John Wiley & Sons, New York, 2004.

[3] G. Karsai, J. Sztipanovits, A. Ledeczi, T. Bapty, Model-integrated development of embedded software, Proceedings of the IEEE 91 (1) (2003) 145–164.

[4] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, G. Karsai, Composing Domain-Specific Design Environments, IEEE Computer (2001) 44–51.

[5] Object Management Group, Deployment and Configuration Adopted Submission, OMG Document ptc/03-07-08 Edition (Jul. 2003).

[6] P. Sharma, J. Loyall, G. Heineman, R. Schantz, R. Shapiro, G. Duzan, 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, 2004.

[7] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, C. D. Gill, QoS-enabled Middleware, in: Q. Mahmoud (Ed.), Middleware for Communications, Wiley and Sons, New York, 2003, pp. 131–162.

[8] D. C. Schmidt, D. L. Levine, S. Mungee, The Design and Performance of Real-Time Object Request Brokers, Computer Communications 21 (4) (1998) 294–324.

[9] Object Management Group, CORBA Components, OMG Document formal/2002-06-65 Edition (Jun. 2002).

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

[11] R. Schantz and J. Loyall and D. Schmidt and C. Rodrigues and Y. Krishnamurthy and I. Pyarali, Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware, in: Proceedings of Middleware 2003, 4th International Conference on Distributed Systems Platforms, IFIP/ACM/USENIX, Rio de Janeiro, Brazil, 2003.

[12] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, V. Prasad, Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems, in: Proceedings of the 25th International Conference on Software Engineering, Portland, OR, 2003.

[13] J. A. Stankovic, R. Zhu, R. Poornalingam, C. Lu, Z. Yu, M. Humphrey, B. Ellis, VEST: An Aspect-based Composition Tool for Real-time Systems, in: Proceedings of the IEEE Real-time Applications Symposium, IEEE, Washington, DC, 2003, pp. 58–69.

[14] S. Kodase, S. Wang, Z. Gu, K. G. Shin, Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-time Systems using Shared Buffers, in: Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS), IEEE, Washington, DC, 2003.

[15] G. Karsai, S. Neema, B. Abbott, D. Sharp, A Modeling Language and Its Supporting Tools for Avionics Systems, in: Proceedings of 21st Digital Avionics Systems Conf., 2002.

[16] W. Roll, Towards Model-Based and CCM-Based Applications for Real-Time Systems, in: Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC), IEEE/IFIP, Hakodate, Hokkaido, Japan, 2003.

[17] J. T. Buck, S. Ha, E. A. Lee, D. G. Messerschmitt, Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, International Journal of Computer Simulation, Special Issue on Simulation Software Development Component Development Strategies 4.