DREMS ML: A Wide Spectrum Architecture Design Language for Distributed Computing Platforms[☆]

Daniel Balasubramanian[a], Abhishek Dubey[a], William Otte[a], Tihamer Levendovszky[a], Aniruddha Gokhale[a], Pranav Kumar[a], William Emfinger[a], Gabor Karsai[a,*]

[a] *Institute for Software-Integrated Systems, Dept of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37212, USA*

Abstract

Complex sensing, processing and control applications running on distributed platforms are difficult to design, develop, analyze, integrate, deploy and operate, especially if resource constraints, fault tolerance and security issues are to be addressed. While technology exists today for engineering distributed, real-time component-based applications, many problems remain unsolved by existing tools. Model-driven development techniques are powerful, but there are very few existing and complete tool chains that offer an end-to-end solution to developers, from design to deployment. There is a need for an integrated model-driven development environment that addresses all phases of application lifecycle including design, development, verification, analysis, integration, deployment, operation and maintenance, with supporting automation in every phase. Arguably, a centerpiece of such a model-driven environment is the modeling language. To that end, this paper presents a wide-spectrum architecture design language called DREMS ML that itself is an integrated collection of individual domain-specific sub-languages. We claim that the language promotes "correct-by-construction" software development and integration by supporting each individual phase of the application lifecycle. Using a case study, we demonstrate how the design of DREMS ML impacts the development of embedded systems.

Keywords:

[☆]This work was supported by the DARPA System F6 Program under contract NNA11AC08C. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of DARPA. The authors thank Olin Sibert and Graham O'Neil, and all the team members of our project for their invaluable input and contributions to this effort.

[*]Corresponding Author

Email addresses: `daniel@isis.vanderbilt.edu` (Daniel Balasubramanian), `dabhishe@isis.vanderbilt.edu` (Abhishek Dubey), `wotte@isis.vanderbilt.edu` (William Otte), `tihamer@isis.vanderbilt.edu` (Tihamer Levendovszky), `gokhale@isis.vanderbilt.edu` (Aniruddha Gokhale), `pkumar@isis.vanderbilt.edu` (Pranav Kumar), `emfinger@isis.vanderbilt.edu` (William Emfinger), `gabor@isis.vanderbilt.edu` (Gabor Karsai)

Architecture description language, model-driven development, fractionated spacecraft

1. Introduction

Today's large-scale distributed real-time embedded systems are enormously complex and there is an ever-increasing need for engineering tools to support their design, development, integration, deployment, operation, and maintenance. Often these systems are running on a mobile distributed platform with wireless connectivity. Such platforms are expected to host many different applications running side-by-side, possibly in different security domains. The platform is highly resource-constrained thus resource management is an issue. Applications are often mission-critical and the system is expected to provide some guaranteed level of service in all situations, hence fault tolerance is a requirement. As an example one can consider a swarm of UAVs that act as a sensor network with in-network processing (while also flying in formation) or a fractionated spacecraft.

The DARPA System F6 program[1] was concerned with developing a cluster of small, independent spacecraft modules that interact wirelessly to maintain coordinated flight and support the functions usually performed by a monolithic satellite. This hardware platform is considered a reusable resource, a 'global space commons', on which various distributed software applications can be deployed and executed such that they share cluster resources. For a variety of reasons, many challenges arise in hosting these applications so that they can deliver services with the expected level of quality. For instance, it is expected that multiple organizations and users whose diverse software applications have varying demands for computational and communication resources are to be supported on this distributed hardware which can experience highly fluctuating connectivity. Moreover, the success of every mission depends on the capability of autonomous fault management, the delivery of desired real-time services and the availability of separate domains and levels of security.

To support these needs, we have developed an architecture called Distributed Real-Time Managed System (DREMS) [1], and its core, the Information Architecture Platform (*IAP*) for F6 that comprises (i) a novel operating system, (ii) a middleware layer that supports different communication-interaction patterns including request-response and publish/subscribe, and (iii) a component model that is used to develop DREMS applications [2, 3]. Note that an entire F6 cluster of satellites is considered an open (hardware) platform whose services are available through the software platform (IAP) to applications developed by various customers. In this sense it is similar to a cloud-computing platform that follows a 'Platform as a Service' model.

[1]F6 stands for 'Future, Fast, Flexible, Fractionated, Free-Flying spacecraft'

The architecture provides first class support for multiple levels of security (MLS) that is enforced by the operating system kernel, the core ingredient of the Trusted Computing Base (TCB). Multi-level fault management is also supported by the IAP, with different functions, such as detection, mitigation, recovery distributed across the architecture.

Despite the elaborate and elegant runtime architecture to support DREMS applications, developers face a number of complex inherent and accidental challenges in constructing their applications. The inherent challenges pervade all phases of the application lifecycle, including design, development, deployment, resource scheduling, security provisioning, verification, determining the right testing strategies, runtime resource and fault management, and dealing with evolution in requirements and maintenance. The accidental challenges stem from the mundane and error prone activities of composing application components, providing the glue code to interact with the middleware capabilities, deploying the applications on the resources of the cluster, configuring the resources according to the partition schedule, provisioning monitoring and fault detection capabilities, testing, and dealing with all these complexities when iterating over the development cycle due to changes in requirements [4].

Clearly, there is a need for tools that application developers can use to handle all these challenges. Although a variety of tools that handle individual aspects of the problem space exist, such a disparate and disconnected tooling capability is not desired for a number of reasons. First, every different tool implies a learning curve on the part of developers and having to deal with the vagaries of individual tools. Second, the developers are now responsible for connecting these disparate tools into a tool chain requiring a number of transformations from the output of one tool to another. While these challenges are predominantly accidental, a more serious challenge stems from the fact that most of the existing tools do not provide domain-specific architectural reasoning capabilities desired for the IAP.

Architectural description languages, such as the Architecture Analysis and Design Language (AADL) [5, 6], OMG SysML [7, 8], and OMG MARTE profile for UML [9], are geared towards addressing the problem of disconnected and disparate tooling. Architecture description languages enable the proper decomposition of the system into manageable parts with well-defined interfaces (and thus contracts) between them, which ease system integration problems. Overall, an architectural model defined in these languages helps to capture in a single place details about the system's requirements, architecture and implementation details. A significant advantage for developers is that they can generate a variety of artifacts: analytical models to conduct timing, reliability, security, performance, etc. analysis from a single source. When language capabilities are offered in the context of a model-driven engineering process, particularly in the form of model-integrated computing (MIC) [10], application developers can validate their system using domain-specific artifacts that promote the realization of systems that are "correct-by-construction", and utilize the model transformation capabilities of MIC that can automate most of the mundane, accidental complexities faced by developers.

This paper presents DREMS modeling language (ML), which is an architecture description language (ADL) and its associated tooling. DREMS ML is a "wide spectrum" ADL because it provides a single source for DREMS application developers and the system integrator to address all the inherent and accidental challenges described above in realizing DREMS applications. Our recent publications describing DREMS ML [11] have focused on showing how it helps developers use the underlying component abstractions, configure the components and deploy applications that are composed of interacting components. Moreover, the focus of these papers was to describe the language in terms of its support for reusability, property configuration at various levels and integration with textual languages. The key topics we cover in this paper are the following:

- We briefly review the main features of the IAP with a focus on the software component model. Our aim here is to highlight the development lifecycle of DREMS applications that influences the design of the DREMS ML.
- We present the design of DREMS ML including all the sub-languages it provides that address the inherent challenges in the different phases of an application's lifecycle.
- We illustrate how the language can be used throughout various system development activities.
- We show through a case study how DREMS ML is used to develop applications and evaluate how the design of DREMS ML addresses the challenges.

The rest of this paper is organized as follows: Section 2 provides an overview of the IAP, Section 3 delves into the details of DREMS ML, articulating how it addresses the inherent and accidental complexities faced by developers; Section 4 describes the development activities supported by DREMS ML. Section 6 describes related research and compares it to DREMS ML; Section 5 evaluates the DREMS ML design in the context of a use case; and finally Section 7 offers concluding remarks alluding to future work in this area.

2. The DREMS Information Architecture Platform

In this section, we briefly present the DREMS Information Architecture Platform (IAP), which is the the run-time software platform and framework for System F6. IAP also offers a component model [2, 3], which we also describe.

The IAP has a layered architecture [2] (shown in Figure 1, for a single host) that comprises a novel operating system (DREMS OS), a modified and extended Linux kernel, a middleware layer(DREMS ORB) and the component-based applications. Instead of traditional processes, DREMS executes applications in the context of **actors**. Actors are temporally and spatially isolated processes that are extended in the following ways. (i) They are unique identity across multiple hosts, (ii) can be migrated from one host to another, and (iii) can be persisted and restored. The operating system provides primitives for concurrency,

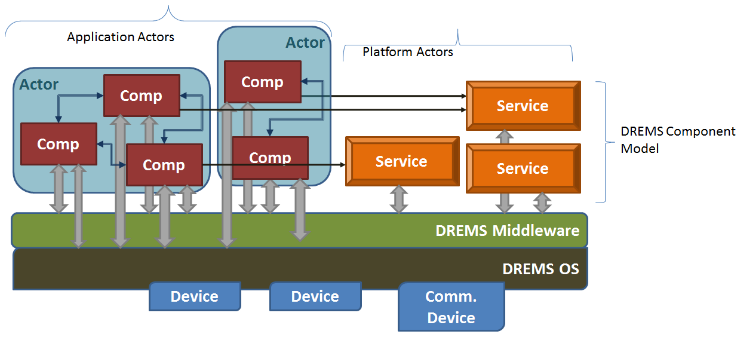


Figure 1 Information Architecture Platform

synchronization, file operations and secure information flows among actors and hosts; it also enforces the temporal and spatial separation of actors, and resource management policies. The middleware provides higher-level services supporting request/response and publish/subscribe interactions for distributed software.

A group of one or more actors deployed together to work collaboratively forms an **Application** and are called *Application actors*. One application may be split across application actors potentially distributed across on different hosts. **Platform actors** are actors that provide system-level services, such as system management, component deployment, and fault management.

2.1. Component Model

The component model (DREMS COM) facilitates the creation of software applications from modular and reusable components that are deployed in the distributed system. Components are the basic units of composition for creating distributed software applications on the IAP. Components are hosted inside *containers*, the portion of the component middleware responsible for managing the lifecycle of component instances. Actors (and thus applications) are constructed from *interacting* components, and all synchronization and data exchange among components happens through interactions. The component model specifies the execution semantics of a component, the interaction semantics between any two components and the interaction patterns between the component and the services provided by the framework to manage the life cycle of the component.

2.1.1. Interaction

Figure 2b shows the different communication and interaction patterns that are currently supported by the DREMS middleware. All data exchanged in these patterns are strongly typed. These patterns can be grouped into two categories:

- Group Publish-Subscribe: A publisher is a point of data production and a subscriber is a consumer of data. This is a group interaction pattern, wherein multiple publishers and subscribers communicate within a specified 'domain' over a specified 'topic name'. All data exchanged between a publisher and a subscriber can be either stateful (i.e there are different instances of the data) or stateless, a.k.a events (i.e. there is only one instance of the data). Events are like singleton classes where there is only one instance of the class, but the value of the data members can change. Stateful data implies that there can be different object instances of the topic with a different value assigned to each instance. These interactions are specified in the OMG Data Distribution Services standard [12].

- Point to point: A point to point interaction is between two components; one client (that has a receptacle) and the other server (that provides facet). Facets and receptacles are typed with an interface that defines a collection of methods. A facet port (of a server) is attached to the implementation of the methods defined in the interface and it services the requests issued through a receptacle (of the same type) of another component (a client) for these interface methods.

Table 1 lists the communication patterns that can be realized using the extended ports that are currently part of the platform. Though the interaction semantics of these ports is pre-specified, a middleware framework can choose multiple implementations for them. For example, synchronous call/return and asynchronous messaging can be supported using OMG's CORBA infrastructure while the publish/subscribe mechanism can be supported using OMG DDS. Consequently, the DREMS COM implementation decouples the transport mechanism from the structural artifacts of a component by using the notion of connectors [13]. A connector is a pluggable unit of container functionality; they are similar to components in that they have a well defined interface, but are entirely generated or implemented by the component middleware. Connectors are deployed much like copmonents, and serve as the interface between a component and the underlying communications middleware or other container services. This design choice also enables the extension of available interaction patterns by allowing the creation of new extended ports without requiring changes to key portions of the component model implementation.

2.1.2. Component Architecture

Figure 2a provides an overview of the component model. Only the business logic of the component needs to be manually written, the rest of the architecture can be achieved by configuring the services provided by DREMS.

While the execution semantics are controlled by the component executor, i.e. the business logic of the component, the interaction patterns are realized via *extended ports* which support many distributed interactions patterns that are commonly used. These ports are called *extended* because unlike traditional single interface ports, these ports provide a collection of semantically related interfaces that are used together.

Figure 2 shows a number of extended ports – each of them represent a requested middleware service. It also shows the service interaction patterns that

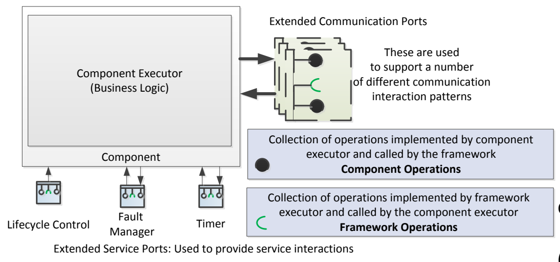
Pattern	Scope	Description
1. Asynchronous remote method invocation	Point to point	Client component with receptacle makes the call but does not block waiting for a reply from the server. The reply from the server is handled as a separate component operation on the client component.
2. Synchronous remote method invocation	Point to point	Client component with receptacle makes the call and is blocked while waiting for a reply from the server.
3. Stateless Data publisher and pull subscription (Event Publisher and Read or Get Subscriber)	Group Publish Subscribe	An event can be published and received by a number of subscribers. A new publication replaces any old existing sample of the data in the buffers. The subscribers are responsible for polling the middleware periodically for new events.
4. Stateful Data publisher and pull subscription (State Publisher and Read or Get Subscriber)	Group Publish Subscribe	The publisher controls whether a new instance (distinguised by a key field) of the topic is created or whether the value of an existing instance is updated. The publisher can also control whether an existing instance is completely removed from the distributed system. The subscriber is responsible for polling the middleware for updated instances.
5. Stateless Data publisher and push subscription (Event publisher and Push Consumer)	Group Publish Subscribe	The publication side in this pattern is similar to item 3 above, but a callback (invocation of a component operation) is made by the middleware to the subscriber when new data is available.
6. Stateful Data publisher and push subscription (State Publisher and State Notify Subscriber)	Group Publish Subscribe	The publication side in this pattern is similar to item 4 above, but a callback (invocation of a component operation) is made by the middleware to the subscriber when new data is available. A component can also control if it receives an invocation on a separate operation if the stateful publisher changes the lifecyle of a topic instance. This is typically used to indicate the ingress or an egress activity from a group of publishers and subscribers.

Table 1 Communication patterns that can be realized using the available extended ports

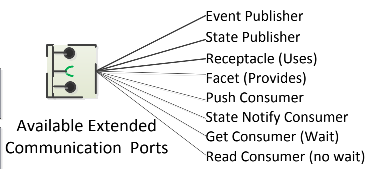
can be realized by using the service extended ports. The service patterns are used to interact with the framework managing the component and can also be used to support periodic and aperiodic time-based triggers that initiate component operations. Additionally, they can support state variables: component attributes with (limited) history, which are often needed in software interacting with physical phenomena. These interaction patterns can be also used to support fault management and component life cycle control, i.e. activating or shutting down a component.

As shown in Figure 2 functions associated with all extended ports, both service and communication, can be grouped into two categories (a) component [provided] operations and (b) framework [provided] operations. It is key to state the difference between two.

The component operations are written by the component developers and provide implementations that state the component's response to either a service


Extended Communication Ports
These are used to support a number of different communication interaction patterns
Component Executor (Business Logic)
Component
Collection of operations implemented by component executor and called by the framework
Component Operations
Collection of operations implemented by framework executor and called by the component executor
Framework Operations
Lifecycle Control
Fault Manager
Timer
Extended Service Ports: Used to provide service interactions

(a) DREMS Component



(b) Communication Interaction Patterns provided by the IAP

Figure 2 Component (left) and interaction patterns (right)

interaction (including a timer expiration, see container services) or a communication interaction. These operations have access to the component state directly and can alter it. The framework provided operations are called from within a component operation.

The framework operations typically do not block and return immediately, except in two cases: (a) synchronous remote method invocation and (b) a get or a waiting read performed by a subscriber. In both cases, the framework operation call blocks until either a response is available or a timeout occurs. To specify the timeout, all framework operations are marked with a timeout parameter. This parameter is specified and configured using the modeling language.

Besides the extended ports, there are three container services: (i) Lifecycle Control, (ii) Fault Manager, and (iii) Timer. While lifecycle control is responsible for delivering lifecycle events such as the initialization and shutdown, Fault Manager handles fault-related events. The timer service is a periodic timer that can be configured to periodically invoke a component operation. It is a completely separate concept from the service/system timeouts.

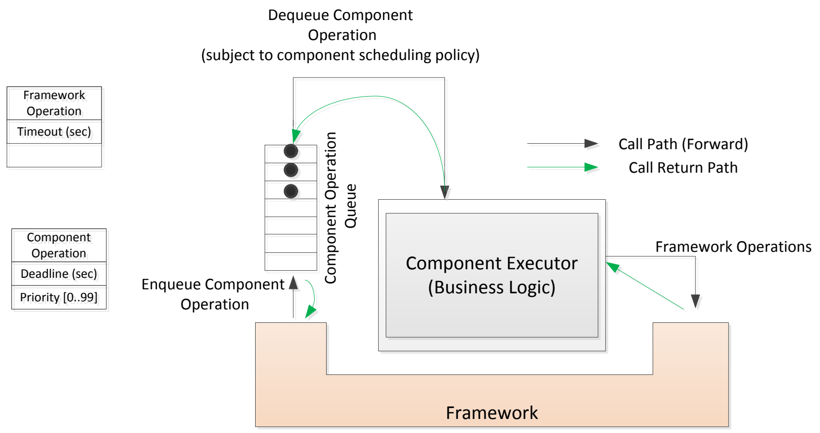

Dequeue Component Operation (subject to component scheduling policy)
Framework Operation
Timeout (sec)
Component Operation
Deadline (sec)
Priority [0..99]
Component Operation Queue
Call Path (Forward)
Call Return Path
Framework Operations
Component Executor (Business Logic)
Enqueue Component Operation
Framework

Figure 3 The Component Scheduler

Figure 3 shows the interactions between the framework and the component executor i.e. the business logic. Component operations invoked by the framework are queued in a component operation queue. By default, the size of this queue is only bounded by available memory in the actor. In the event that a lower upper bound on the queue size is specified, any failure to insert an operation into the queue would be indicated to the Fault Manager, and if appropriate throwing an exception to the client that initiated the operation. The framework issued enqueue command returns immediately to the framework. The component operation queue selects the next component operation to be executed based on the a configurable scheduling policy. Currently, we support two policies: Priority First in First Out, and Earliest Deadline First. Both of these policies are non-preemptive i.e. an operation once started cannot be interrupted. However, it is possible to monitor for deadline violation for the operation. The fault management mitigation action to be issued upon deadline violation is a subject of ongoing research and implementation; possible mitigation strategies might include (1) indicating an exceptional condition to the client (in the case of AMI/RMI), (2) failing over to a replica, (3) restarting the actor entirely, or (4) an application specific response supplied by the application developer.

To support these scheduling policies two attributes for each component operation must be configured: deadline (i.e. anticipated worst-case execution time), and an unsigned integer priority. For normal operations priorities must fall within an closed interval (e.g. [0..99]), and while priorities greater than the larger value of the interval are possible, they are reserved for component operations used to implement the lifecycle and fault management service interaction patterns (see figure 2a).

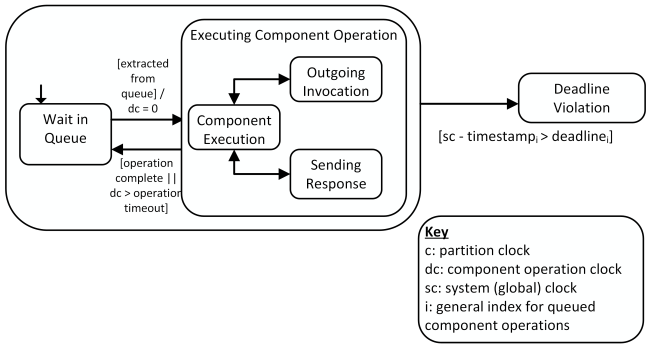

Executing Component Operation
[extracted from queue] / dc = 0
Wait in Queue
Component Execution
Outgoing Invocation
Sending Response
[operation complete || dc > operation timeout]
Deadline Violation
[sc - timestamp$_i$ > deadline$_i$]
Key
c: partition clock
dc: component operation clock
sc: system (global) clock
i: general index for queued component operations

Figure 4 Scheduling of component operations

Figure 4 depicts the semantics of component scheduling from the perspective of the Framework (Figure 3). The Figure uses hierarchy only as a syntactic convenience to omit drawing transitions to/from every state contained by a higher-level state; otherwise, our notation has the same semantics as a timed-automata. The framework takes an operation from the Component Operation Queue (transition from the state $WaitinQueue$). Then the execution of component operation takes place. If executing the operation takes more time than the specified timeout, the framework aborts the execution of the operation. It may involve an outgoing invocation (network communication) or sending response to a network communication query. During this whole process if a timeout expires on an operation in the operation queue (this timeout is different from the one for the operation execution), the framework handles it.

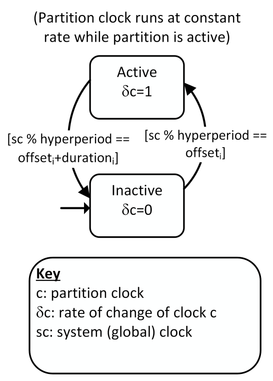
2.2. Task Scheduling

In this section, we briefly summarize the scheduling services provided by DREMS. It groups tasks into different criticality levels: (a) *Critical* tasks are those tasks which are required for system and mission management; (b) *Application* tasks perform mission-specific, non-critical work; (c) *Best Effort* tasks are those low priority tasks that are scheduled only when there are no runnable tasks from the previous two categories.

The DREMS OS scheduler provides the ability to manage computation time for tasks at the three different criticality levels: *Critical*, *Application* and *Best Effort*. The *Critical* tasks provide kernel level services and system management services. These task will be scheduled based on their priority whenever they are ready. *Application* tasks are mission specific and are isolated from each other. These tasks are constrained by temporal partitioning and can be preempted by tasks of the *Critical* level. Finally, *Best Effort* tasks are executed whenever no tasks of any higher criticality level are available.

Note that actors in an application can have different criticality levels, but all tasks associated with an actor must have the same criticality level, *i.e.* an actor cannot have both *Critical* tasks and *Application* tasks.

Therefore, the scheduling policy must be configured for the Application Actors. Their temporal isolation is provided via ARINC-653 [14] style partitions – a periodically repeating fixed interval of the CPU's time exclusively assigned to a group of cooperating actors of the same application. Figure 5 depicts the partition scheduler as a stopwatch automaton [15] from a partition's perspective. A partition can be in an active or inactive state. As described in Section 3.2.9, partitions have an associated period and duration such that each partition is run for the length of its duration every time an amount of time equal to its period elapses. In order to find a schedule that satisfies the period and duration constraints of all partitions, a constraint satisfaction problem [16] is formulated such that a solution to this problem assigns to each partition a starting and ending time relative to a calculated *hyperperiod*. The hyperperiod is the smallest interval of time after which the periodic patterns of all the tasks is repeated. Because the period of the scheduling is the hyperperiod, the time within the current hyperperiod is equal to the system clock (sc) modulo the hyperperiod. $Offset_i$ marks the start of $partition_i$ within the hyperperiod.



(Partition clock runs at constant rate while partition is active)
Active
δc=1
[sc % hyperperiod == offset_i+duration_i]
[sc % hyperperiod == offset_i]
Inactive
δc=0
Key
c: partition clock
δc: rate of change of clock c
sc: system (global) clock

Figure 5 Partition Scheduling

3. DREMS ML Architecture Design Language

3.1. Rationale for the Design of DREMS ML

This section describes the DREMS ML ADL [2]. Our design of DREMS ML is based on the argument that an ADL has to be expressive enough to support all developmental activities shown in Figure 6 that ultimately produce deployable software products.

As Figure 6 shows, the component implementor creates individual components, including the definition of their interfaces and their business logic. The application developer is responsible for creating applications using these components. The system integrator then combines multiple applications to create a deployment package that is run on the target system. In order for the system integrator to perform verification and validation tasks, such as security analysis (Section 4.3.2) and scheduling analysis (Section 4.3.5), the application models must contain all information relevant to each analysis.

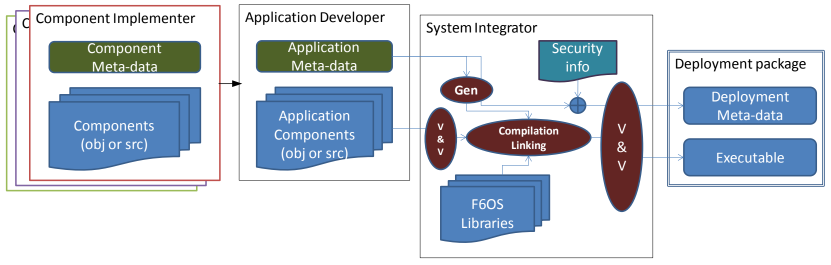


Figure 6 DREMS Application Development Workflow and Roles

[2]The DREMS package, which contains the language and examples, is available at https://drems.isis.vanderbilt.edu/

In other words, models have to represent interfaces, the components and the architecture of applications, together with details about the software platform and how applications are to be deployed on the platform. What an ADL should *not* capture is the internal behavior of the components – this is best left in the hands of skilled developers. An example internal behavior could be the algorithm that a component uses to calculate data that is then sent to other components.

When a component-based development process is followed as in DREMS, the platform has to clearly delineate what a component is. We argue that a component model (supported and enforced by a run-time framework) is essential. When the ADL is used to model an application in terms of interacting components, the designer has to clearly understand what the assembly of components means and how it works.

Another aspect of component-based development is that of deployment: at run-time, applications are created on-the-fly by activating and 'wiring up' components. Information needed for such deployment activities must be generated from the ADL and processed by a special platform component: a deployment and configuration engine that manages all applications and their components running on the platform. Finally, the ADL also serves as a system integration tool. For complex systems, e.g. the complete avionics suite of an aircraft, applications are developed in parallel, by different organizations. A distinct system integrator has to integrate all the applications into a coherent software package that is then deployed. Hence, the ADL should be capable of modeling the complete set of distributed applications, which allows the integrator to (1) make admittance decisions that ensure that the resources required by all applications will be available on the system, and (2) perform the 'systems engineering' by enabling various interactions (e.g. data flows) among applications.

We call such ADLs *wide spectrum*, as they support a wide variety of development activities. Each of the considerations mentioned above are incorporated in the design of DREMS ML and explained in the rest of this section.

Before delving into the details of the modeling language, we provide an overview of what is deployed onto the runtime system (the IAP) and how the modeling language fits in. Figure 7 shows a high-level view of the overall workflow. Three items are generated directly from a model: code, build system files and a deployment plan. The build system files describe how to compile the source files into the executable binaries that are eventually deployed onto the system. The compilable code consists of IDL that describes the component interfaces, which is translated into C++ code. This is combined with the component business logic (i.e., the internal implementation) to produce the executable binaries. The business logic and external libraries are not directly subject to the timing analysis described in Section 4.3.5. Instead, users describe an abstraction of the business logic's timing behavior inside the model, and this abstraction is analyzed for properties such as deadline violations and response times; Section 4.3.5 provides details.

The final product deployed onto the IAP consists of two items: a package of binary executables and a *deployment plan*. The deployment plan (physically:

an XML file) contains a declarative description of the software components implemented in the binary executables, along with a list of which components to instantiate, where those components should be instantiated and the communication connections that must be established between the components.

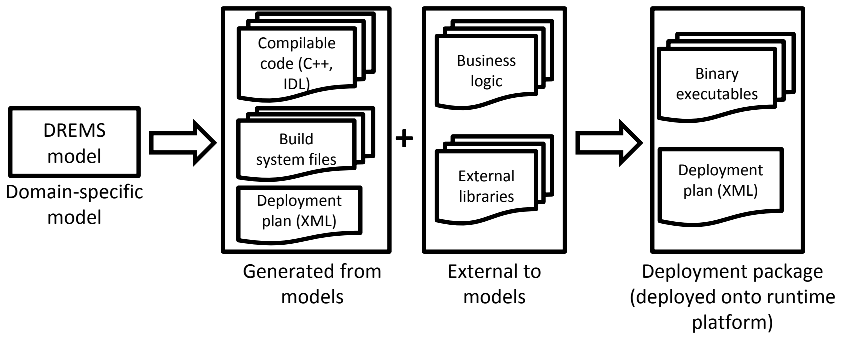


Figure 7 Workflow showing the automatically generated artifacts and their use in the development process.

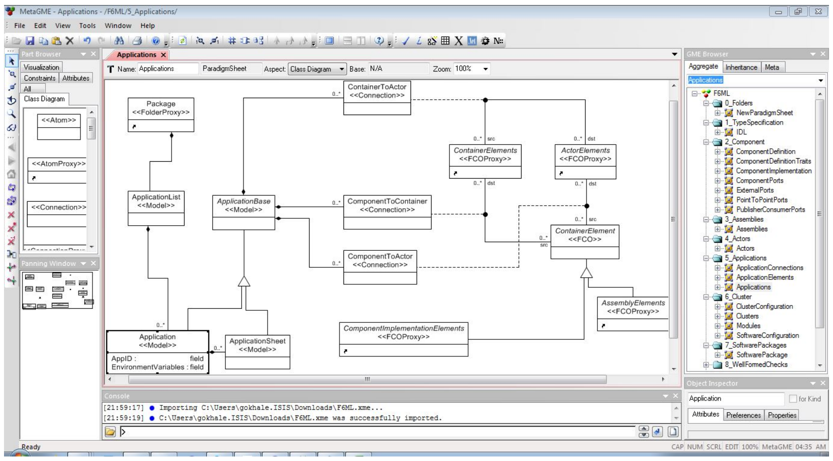
3.2. Design of DREMS ML

The DREMS ML is a wide spectrum modeling language that supports the entire development process. Below we describe the design of DREMS ML and show how it supports all the phases of the development lifecycle. In doing so it describes the design of the different sub-languages of DREMS ML that were designed to address the stages of the process shown in Figure 6. For each sub-language we describe (1) what is being modeled, (2) the motivation for including it in the ADL, and (3) what it contributes to in the final product of the development process. The sub-languages of the DREMS ML and the metamodel of one of these sub-languages – in this case modeling of an application – is shown in Figure 8. The syntax and semantics of the DREMS ML language are described using the MetaGME Language [17].

Shown in the right pane of the screenshot are all the sub-languages supported by the DREMS ML. These languages are numbered and this numbering effectively follows the application development lifecycle phases shown in Figure 6. As seen in the figure, a single overall modeling capability covers all the stages of the lifecycle, which is the reason for its “wide spectrum” property. The rest of this section delves into the design and justification for each of these sub-languages.

3.2.1. Initial Step in Modeling

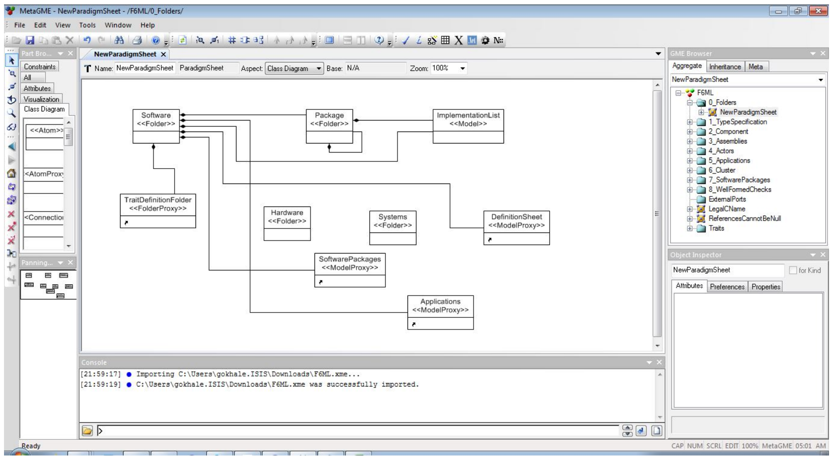
Since the DREMS ML supports a step-by-step approach to DREMS application design and deployment, the language must provide a starting point for the modeling phase. This capability appears in the sub-language numbered zero



```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 8 Overall Structure of the DREMS ML as a MetaModel Modeled in GME

and its metamodel is shown in Figure 9. The sub-language provides three top level entry points for the application designer and integrator: everything that is related with software, hardware and the system itself. The decision to separate these three modeling capabilities is driven by a desire to separate concerns and because different parties may be responsible for defining these three independent artifacts.

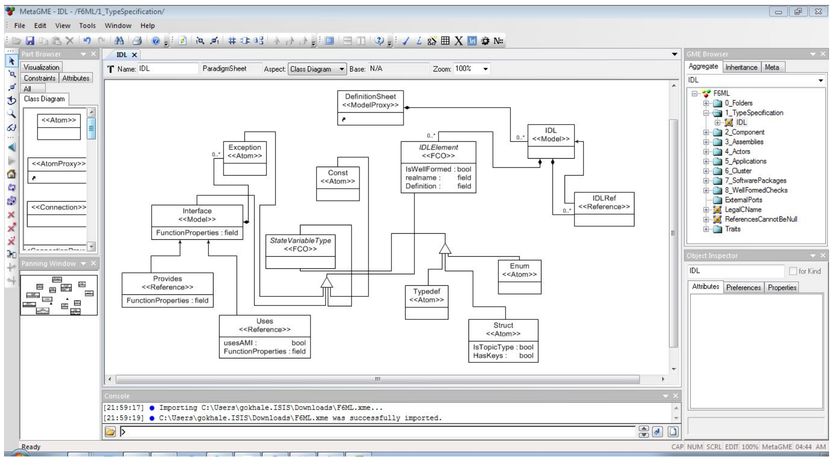


```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 9 DREMS ML-based Modeling: The Starting Point

3.2.2. Data Types and Interface Definitions

The first phase of the application lifecycle starts with component definitions and their implementations. In order to do this, one must first define the data types for all the component attributes and interfaces it supports. Thus, the next sub-language pertains to modeling the data types whose meta model is depicted in Figure 10.



```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 10 DREMS ML: Definining IDL Data Types

The sub-language for data types and interface definitions is used to model: (1) the data types that components use on their interfaces, and (2) the interfaces provided and used by components. This sub-language is present in the modeling language because in addition to primitive data types defined by OMG standards, applications can also have user-defined data types. These user-defined types include: enumerations, sequences (both unbounded and bounded), arrays[3], structures (which can contain any other primitive or user defined type), unions, and typedefs (named types).

Interfaces are named collections of method signatures that can use both built-in and user-defined data types for the parameters. This part of the modeling language provides a central "repository" where interfaces and data types can be defined and then referenced throughout multiple applications. Recall that at a high level, component communication interactions fall into one of two categories: (a) point to point interactions, and (b) publish-subscribe interactions. Point to point interactions are achieved using call-return semantics; the collection of methods that can be called is referred to as an interface.

The sub-language allows modeling the IDL types using both graphical and textual notation. The syntax of the textual description is checked and enforced

[3]Arrays are fixed size containers, while sequences are variable size containers.

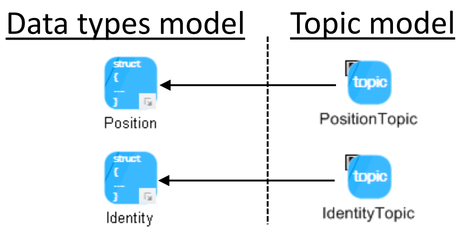


Figure 11 Example Topic definition. References (right-side) to data structures (left-side) define topics.

by including an add-on[4] to the modeling language that parses the user-written data types and interfaces, and alerts the user to any syntactical errors. From this sub-language, IDL code containing syntactically correct data types and interfaces is generated. This IDL is generated by a *model interpreter*, a program that can be invoked from inside the modeling environment and access models using APIs provided by the modeling environment. This is straightforward because the model elements contain syntactically correct IDL code (as described above).

3.2.3. Topic definitions

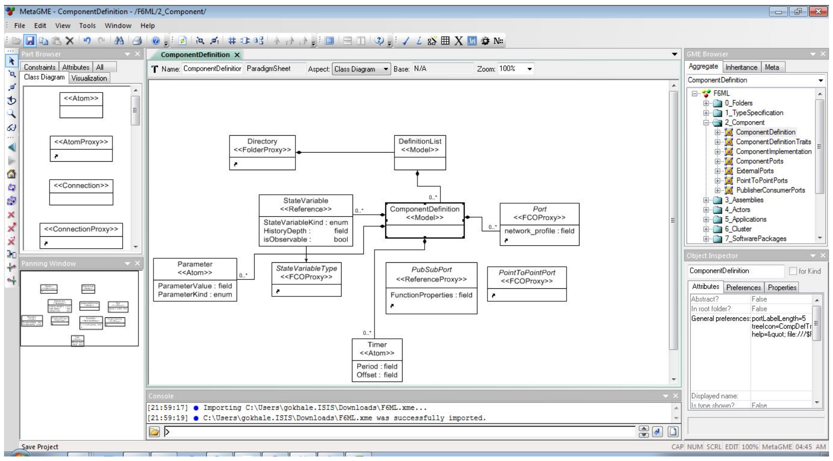
Recall that the group publish-subscribe set of component interactions is performed by the exchange of data, as opposed to the invocation of methods used by point-to-point interactions: a component "publishes" data that is consumed by some number of components that "subscribe" to that data. All data exchanged using the publish-subscribe interaction patterns is identified using *topics*. A topic provides an identifier that uniquely identifies some data items within a publish-subscribe domain. More formally, a topic is a tuple $< x, y >$, where x is a unique name, and y is a structure data type. Note that while this definition allows a single data type to be associated with multiple identifiers, a pair of publish/subscribe ports will only interact if they are assigned the same topic: the same data type alone is not sufficient.

A topic's data type is specified as a data structure using the data types modeling language. Data structures that are used as the data type of a topic can be annotated by *keys* which are used to describe different instances of the same topic. The topic definition sub-language then uses *references* to data structures defined in the data types modeling language to define the topics available for publish/subscribe interactions as shown in an example in Figure 11. From the topic sub-language, configuration files that inform the middleware about the allowed topics are produced.

[4]An *add-on* is an interactive tool: an executable extension to the graphical modeling environment that is activated when a specific editing operation is invoked.

3.2.4. Component and assembly definitions

Having modeled the data types and topic definitions, the modeler can proceed to defining the components and their implementations. The sub-language that enables component modeling is shown in Figure 12. The language is designed in a way to faithfully capture the structure and semantics of a software component. Recall that component definitions represent units of functionality that interact with one another through pre-defined interaction patterns and are the basic building block of applications. This sub-language is one of the core parts of DREMS ML: component definitions define the abstract units of functionality used by applications. From a component definition model, IDL code that describes the ports, interfaces and method properties of each component is generated.



```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 12 DREMS ML-based Modeling: Defining the Components

A component definition consists of the interfaces it provides and requires, the publisher and consumer ports it contains, and its exposed attributes. For this reason all these artifacts are contained within the "Component" model artifact. For each interface method provided by a component, a set of properties are specified, including the worst-case response time, the deadline and whether the deadline is hard (strict) or soft (flexible).

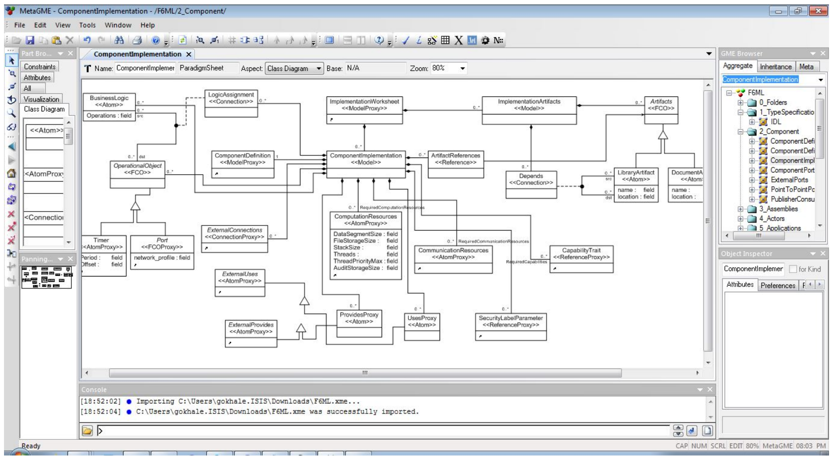
Note that the *definition* of a component is different from its *implementation* (described next). This allows the same unit of functionality (the component definition) to be implemented in multiple ways. Applications can then choose which implementation of a component definition they wish to use.

A component assembly is a group of component definitions that are packaged together to provide a functionality as a composite of simpler functions.

3.2.5. Component implementations

A component implementation represents a particular implementation of a component definition. Recall that a component definition can contain interfaces, which are a collection of methods that are used or provided by the component, as well as DDS ports that the component uses to publish data for other components or subscribe to data published by other components. Each component definition can be implemented in multiple ways, and an application may use multiple implementations of the same component definition.

The component implementation sub-language shown in Figure 13 is present so that users can specify multiple ways of implementing a component definition and override properties of that component definition. From a component implementation model, IDL code describing the implementation is generated.



```
[18:52:02] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[18:52:04] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

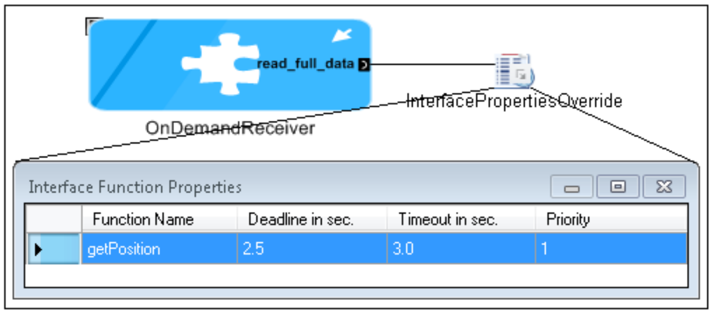
Figure 13 DREMS ML-based Modeling: Defining the Component Implementations

A component implementation in the modeling language contains a collection of artifacts, which represent required dependencies the implementation needs at runtime, such as shared libraries (i.e., .so files) and confirmation files. A component implementation can also contain timers, which are used by the implementation to invoke component operations on itself. An implementation may change the properties of ports contained in the component definition it implements.

Figure 14 shows an example of how the properties of a port can be overridden[5] in an implementation. The properties that can be overridden are the deadline, timeout and priority for individual functions contained (used) by the port. The Figure depicts an implementation of a component definition named *OnDemandReceiver*, which is one of the same component definitions described later in Section 5. This component definition contains one *uses* port named *read_full_data*, which states that the component uses the *PositionServer* interface, which is shown in Figure 27. As shown in Figure 27, the *PositionServer*

[5]While component definition may provide an expectation of function properties, the implementation specifies the actual values.

interface has one method named *getPosition*. The component implementation shown in Figure 14 overrides the properties of this method on this implementation by connecting the port to an *InterfacePropertiesOverride* element (shown on the right side of the Figure) and setting the desired values of the method attributes there.



Function Name	Deadline in sec.	Timeout in sec.	Priority
getPosition	2.5	3.0	1

Figure 14 A component implementation overriding the properties of a port of its component definition

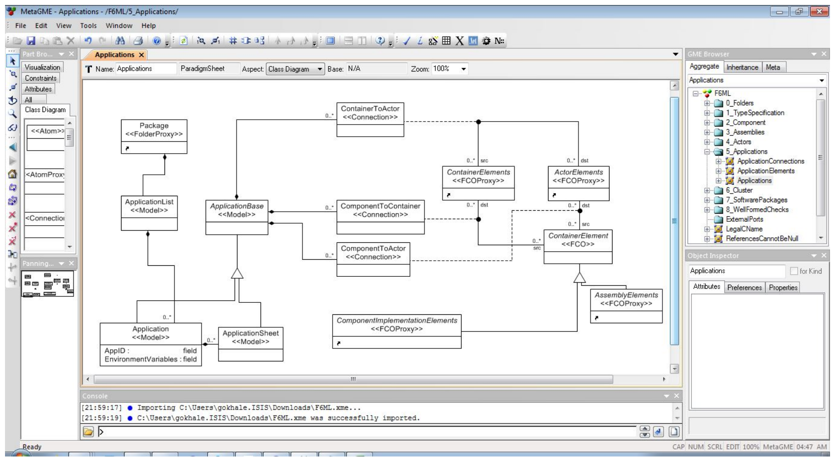
The bottom part of Figure 14 shows that this implementation has overridden the values for this method's deadline, timeout and priority. The deadline and priority attributes affect the operations on the component, and the timeout affects the framework operation as discussed in previous sections. The user interface shown at the bottom of Figure 14 was implemented as an add-on (see Section 3.2.2 for a brief definition) to the modeling environment for convenient specification.

3.2.6. Applications

The next phase in the application lifecycle is defining the applications using the components and their implementations that were defined in the earlier phase. An application consists of a group of communicating components. The application sub-language shown in Figure 15 defines applications by combining elements from the sub-languages above and extending them with additional information. An application consists of one or more component definitions, possibly grouped into assemblies, each of which is associated with a particular implementation of that component definition. The component definitions are then assigned to *actors*, which are similar to operating system processes and form the basic unit for scheduling in the system.

The application sub-language also defines the connections between the ports of components in that application. Every *uses* port of a component is connected to a *provides* port of some component. The publisher and subscriber ports of an application have topics assigned, which ensures that consumer ports using a certain topic will receive data from any publisher ports using that topic.

From an application model, a deployment plan is generated. A deployment plan is a configuration file that describes the locations of all the files and artifacts



```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 15 DREMS ML-based Modeling: Modeling the Application

needed to launch the application. This deployment plan is used by a special actor called the *Deployment Manager* to launch the actor and to configure the components and the middleware within that actor.

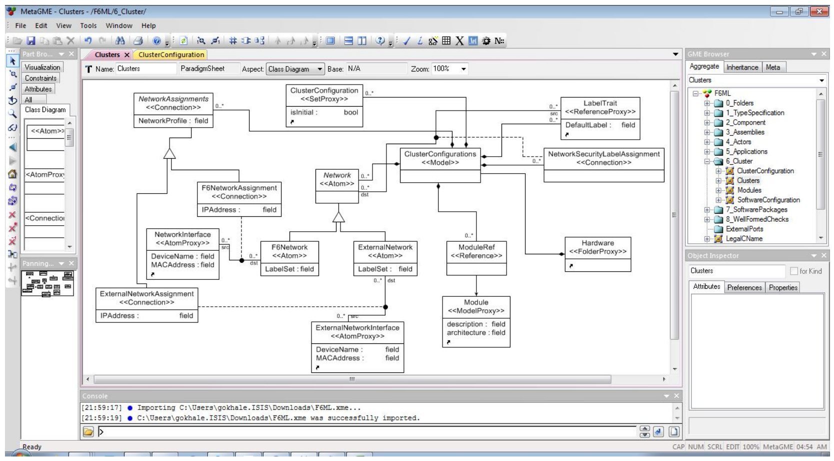
3.2.7. Platform definitions

Recall that at the top level, the DREMS ML allows a different set of modeler to define the characteristics of the platform used by the system The platform definition sub-language shown in Figure 16 describes the physical hardware devices that can be a part of the target system. The System F6 IAP is aimed at space systems, so these models describe both in-space modules and ground modules that are to communicate with space modules. The main features captured by this sub-language are the device, network and external network interfaces. Because the underlying software infrastructure has stringent requirements about the networks and devices which can be used, these networks and devices must be explicitly modeled. These models are then reused in the platform configuration models, as described below.

Figure 17 shows an example model with two modules (the left side of the Figure) and the interfaces they contain (the right side of the Figure).

3.2.8. Platform Configurations

The preceding section described the platform definition sub-language, which defines the devices that may be available during a mission. Because missions are designed to operate in a dynamic environment where both software and hardware faults may occur, all devices may not be available throughout an entire mission. Devices can fail, and both planned and unplanned network outages may occur. Additionally, the configuration of the devices may change over time: the network address of a network interface may change, or the interface may be assigned to a different network altogether.



```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 16 DREMS ML-based Modeling: Modeling the Platform

The platform configuration sub-language defines different configurations of modules and their interfaces. Each configuration consists of a set of modules and the mapping of their network interfaces onto available networks, including a network address. This allows a platform configuration to describe the state of the physical system at a given point in time. From this description, a configuration file that can establish the network configuration for each platform configuration is generated that is used at software deployment time.

Figure 18 shows a sample platform configuration that uses the modules defined in Figure 17. The modules are proxies of the modules defined in the platform definition sub-language. Notice that the network and network address change between the configurations. Alternative configurations are specified statically in the modeling language, but the switch to a new configuration on the IAP runtime platform happens dynamically in response to events, such as a device failing.

3.2.9. Software packages

The software package sub-language shown in Figure 19 describes three items: (1) the scheduling of the application's actors, (2) the domains[6] used by components for their publish and subscribe ports, and (3) the point-to-point interactions between ports in different applications. Actor scheduling is configurable within an application, and is therefore present in the modeling language. Actors are assigned to schedules in one of two ways: they can be explicitly assigned to a schedule's temporal partition with a given period and duration[7], or they can be assigned so that the operating system schedules the actors using a 'best

[6]Domains are used to specify a region of the data space within which the publish subscribe interactions can be established.

[7]Here we follow the ARINC-653 temporal partitioning model

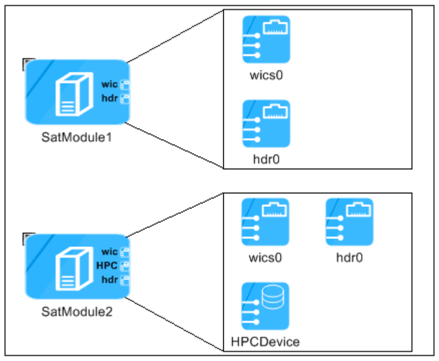


Figure 17 Example platform definition. The *SatModule1* module has two network interfaces, and the *SatModule2* module has two network interfaces and one device interface.

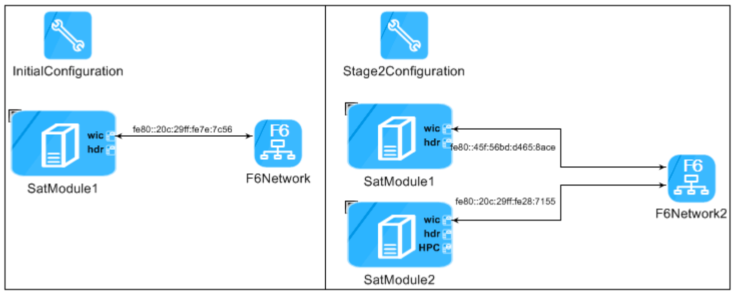


Figure 18 Two platform configurations. Only *SatModule1* is present in the first configuration (left side), but both *SatModule1* and *SatModule2* are present in the second configuration.

effort' approach. Best effort actors run in the unused slack time remaining after the actors running in the temporal partition(s) are executed.

For publish/subscribe, a domain for the publisher and subscriber ports of a component determines the scope in which produced data is visible. For point-to-point interactions connecting the point-to-point ports of components in different applications allows a component in one application to use the interfaces provided by a component in another application.

A software package model is reused in a software deployment model (described below), where the schedules are mapped onto hardware modules and the software domains are mapped onto real networks.

3.2.10. Software Deployment

Recall that the final phase of application lifecycle is handled by the system integrator who decides how the software packages are configured and deployed. The software deployment sub-language shown in Figure 20 describes how soft-

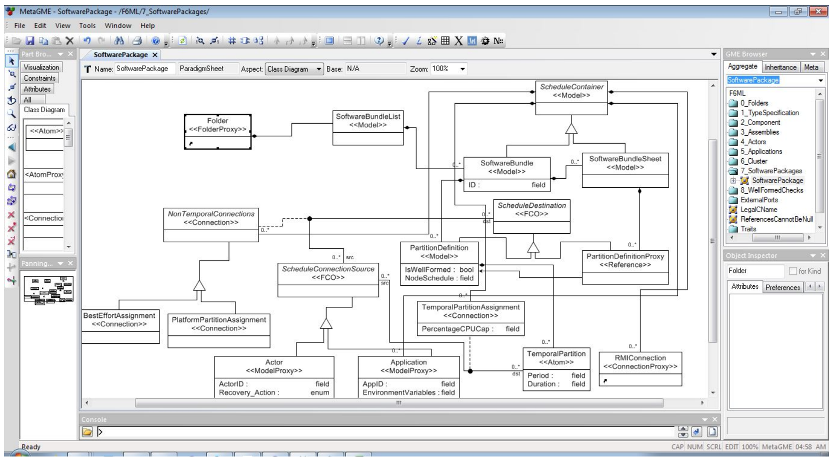
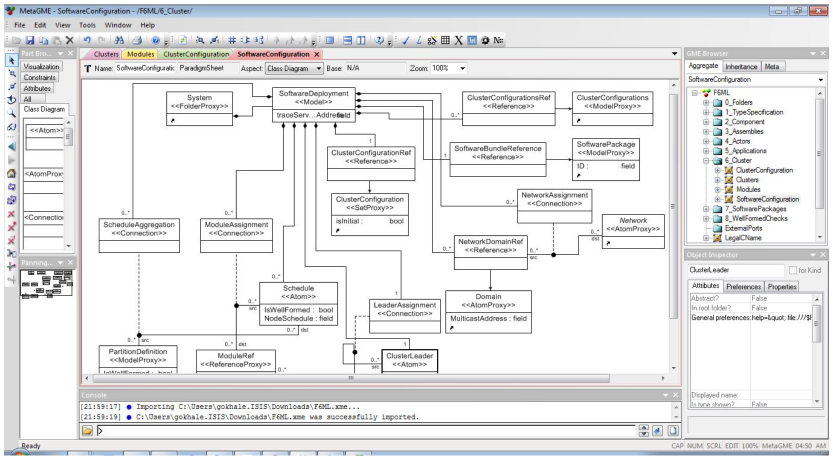

MetaGME - SoftwarePackage - /F6ML/7_SoftwarePackages/
SoftwarePackage
Folder <<FolderProxy>>
SoftwareBundleList <<Model>>
ScheduleContainer <<Model>>
SoftwareBundle <<Model>>
ID : field
SoftwareBundleSheet <<Model>>
NonTemporalConnections <<Connection>>
ScheduleDestination <<FCO>>
PartitionDefinition <<Model>>
IsWellFormed : bool
NodeSchedule : field
PartitionDefinitionProxy <<Reference>>
ScheduleConnectionSource <<FCO>>
TemporalPartitionAssignment <<Connection>>
PercentageCPUCap : field
BestEffortAssignment <<Connection>>
PlatformPartitionAssignment <<Connection>>
Actor <<ModelProxy>>
ActorID : field
Recovery_Action : enum
Application <<ModelProxy>>
AppID : field
EnvironmentVariables : field
TemporalPartition <<Atom>>
Period : field
Duration : field
RMIConnection <<ConnectionProxy>>
GME Browser
F6ML
0_Folders
1_TypeSpecification
2_Component
3_Assemblies
4_Actors
5_Applications
6_Cluster
7_SoftwarePackages
SoftwarePackage
8_WellFormedChecks
ExternalPorts
LegalCName
ReferencesCannotBeNull
Traits

Figure 19 DREMS ML-based Modeling: Modeling the Software Package

ware packages (Section 3.2.9) are mapped onto cluster configurations (Section 3.2.8). This consists of a mapping from the schedules (which describe how the actors of applications are scheduled) of a software package to the modules of a cluster configuration on which those schedules should be run. This mapping assigns a set of actors to a hardware module. This sub-language also maps the publish/subscribe domains of a software package onto the networks available in a cluster configuration. This is used to concretely specify which network carries the publish/subscribe traffic generated by software components.


MetaGME - SoftwareConfiguration - /F6ML/6_Cluster/
Clusters
Modules
ClusterConfiguration
SoftwareConfiguration
System <<FolderProxy>>
SoftwareDeployment <<Model>>
ClusterConfigurationsRef <<Reference>>
ClusterConfigurations <<ModelProxy>>
ClusterConfigurationRef <<Reference>>
SoftwareBundleReference <<Reference>>
SoftwarePackage <<ModelProxy>>
ClusterConfiguration <<SetProxy>>
isInitial : bool
NetworkAssignment <<Connection>>
ScheduleAggregation <<Connection>>
ModuleAssignment <<Connection>>
NetworkDomainRef <<Reference>>
Network <<AtomProxy>>
Schedule <<Atom>>
IsWellFormed : bool
NodeSchedule : field
LeaderAssignment <<Connection>>
Domain <<AtomProxy>>
MulticastAddress : field
PartitionDefinition <<ModelProxy>>
ModuleRef <<ReferenceProxy>>
ClusterLeader <<Atom>>

```
[21:59:17] ● Importing C:\Users\gokhale.ISIS\Downloads\F6ML.xme...
[21:59:19] ● C:\Users\gokhale.ISIS\Downloads\F6ML.xme was successfully imported.
```

Figure 20 DREMS ML-based Modeling: Configuring the Software Package for Deployment

The software deployment process is supported in the modeling language with first class concepts for schedules, modules and the mapping between the two.

4. Developing Systems with the DREMS ML

The development process of applications to be run on the IAP is similar to that of other embedded system applications: analysis, design, implementation, verification and testing; preferably in spiral progressions. The use of an ADL like DREMS ML helps in all these phases by allowing distributed development and integration of systems with specific support for following three key roles: component developers, application developers and system integrators. We describe these roles next.

4.1. Component developers

As described earlier, components are the basic units of software that can be composed together to build larger and more complex distributed software applications. Components are developed with respect to a specification and provide a small number of functionalities. Two different components built for the same specification can be used interchangeably. Part of a component developer's primary task is to model component definitions, implementations and component assemblies and to specify runtime properties for the components. These runtime properties include expected resource requirements, expected security label constraints and the type of interactions supported by a component. To support these activities, the modeling environment provides automatic generation of the necessary build scripts and framework 'glue' code. The glue code includes:

- Data type and interface code.
- Communication stubs and skeletons.
- Placeholders for insertion of component executor or business logic code.

As a component developer creates tests that exercise their code, it is likely that both the model and code will both change as bugs are discovered and the implementation is refined. During this process, it is important that the code automatically generated by the modeling environment does not accidentally overwrite code that was manually added to previously generated code. To ensure this does not happen, special markers are automatically placed in the generated code to delineate where hand-written code should be placed and thus preserved by the code generator in subsequent runs.

4.2. Application Developers

Applications are created by composing various components together, specifying the information flow, specifying resource and/or security constraints. An application developer does not have to write new code. They are able to use the tools to generate deployment plans for their application and run them in a test environment. The application developer role requires the modeling framework to support composing different component models received from different parties and combining them into a single model. This kind of model composition requires the tool to support some sanity checks to avoid duplicate type definitions and name clashes. These checks include ensuring that the different models do not define IDL data types with the same name. Once the models have been composed, the application developers can create test system deployment models to try out their applications. For this purpose, they have to use the system integration role, which is described next. The end-result of the application development process is a set of models, source code files (received along with component implementation), and tested and verified software libraries and artifacts.

4.3. System Integrators

System integration is the phase that results in a verified configuration of all the software for a specific cluster configuration. The tasks of an integrator are to specify the application instances in the system, specify the resource limits for each application instance and specify the communication constraints i.e. the topics and domains[8] being used. The system integrator also specifies the security labels at which all computing hardware and nodes will operate.

Based on these settings, a system integrator can perform a number of design constraint checks described in the following subsections. Once the model has been analyzed, the system integrator generates the application deployment plans and system configuration scripts for the test system. These are then used to deploy the test system and the test applications on the ground. Once verified, the deployment plan and software artifacts are packaged together for deployment in the production system.

4.3.1. Well-formedness checks

The modeling tools allow the system integrator to check that a model satisfies a set of constraints that are specified using the Object Constraint Language (OCL) [18], a standarized language for writing constraints on modeling languages. Listing 1 shows an example constraint written in OCL that checks whether a model satisfies the constraint that the CPU utilization of all components assigned to a partition is less than 1 (100%). These OCL constraints were developed as part of the modeling language and are included with it.

In addition to OCL constraints, the modeling language uses three additional analyses to ensure that models are semantically correct: security analysis, resource anaysis and scheduling analysis. These are described presently.

Listing 1 OCL constraint to ensure valid CPU utilization

```
// Obtain all partitions in a module
let g = self.parts("Partition") in
// Compute utilization of a partition as duration/period
```

[8]A domain can be used to isolate the communication of different applications from each other.

```
let all =g.collect(oclAsType(Partition).Duration/
oclAsType(Partition).Period) in
// Sum the utilization of all partitions
let seq = all.oclAsType(ocl::Collection)->asSequence() in
let result = seq->iterate ( i; sum : ocl::Real=0|sum +
i.oclAsType(ocl::Real)) in
// To be schedulable, the total utilization should be <=1
result <= 1
```

4.3.2. Security analysis

The modeling language supports security analysis in the following way. MLS (multi-level security) labels can be placed on all component ports, actors and hardware modules. These MLS labels are linearly ordered hierarchical classification levels [19]. A label L_a is said to *dominate* another label L_b if the classification level of L_a is greater than or equal to the classification level of L_b, and we say that the dominance relationship holds between L_a and L_b. This dominance relationship is a partial order [20].

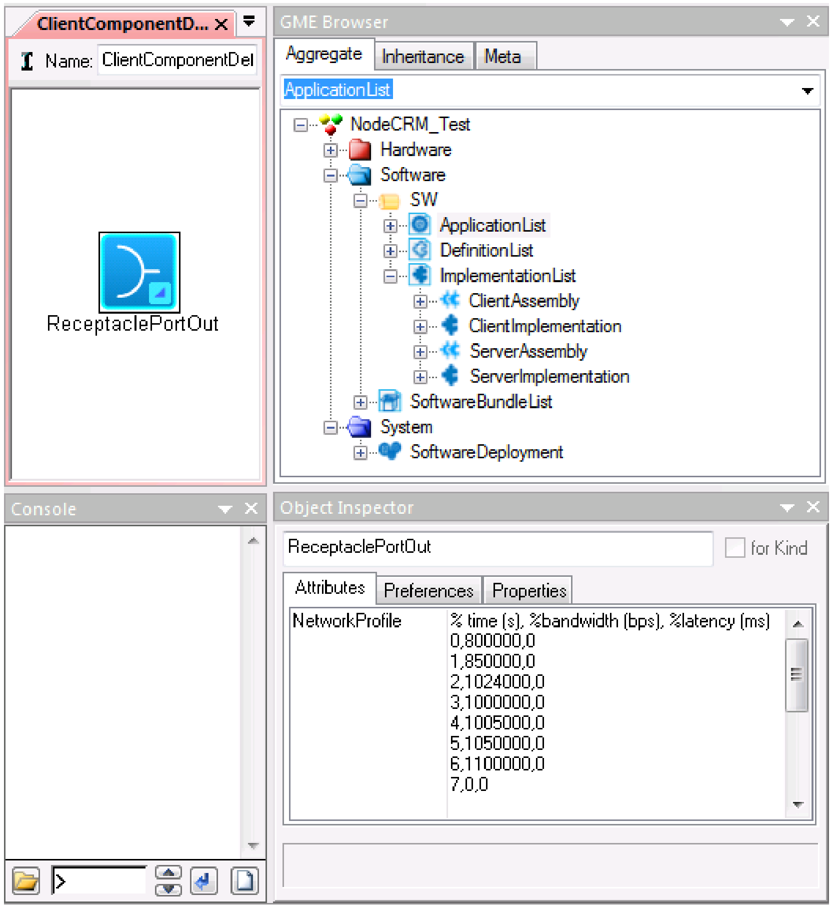
Our MLS policy states that information can flow only from lower to higher labels or between equal labels (according to the domination relation), *e.g.*, a Secret actor for mission A can read Confidential or Secret data for mission A, but not Top Secret data for mission A or Secret data for mission B. Information cannot flow from higher to lower labels or between incomparable labels.

We developed and integrated an MLS label checking library into the modeling language which automatically checks that the information flows, specified by connections between the ports of components in the modeling language, satisfy the constraints of our MLS policy (i.e., that information can only flow from lower to higher labels or between equal labels). The modeling language uses this label checking library to perform a static security analysis before a system is deployed. The static security analysis ensures at design time that (1) all intended information flows are indeed allowed, and (2) unintended information flows are disallowed.

This complex label checking is an advantage over general purpose architecture description languages, which do not support security labels and checks on both component ports and hardware modules. Additionally, the underlying platform enforces the MLS policy at runtime, ensuring that the label constraints that were checked statically in the modeling language are also enforced during execution.

4.3.3. Network resource analysis

DREMS ML includes a network resource analysis tool that allows a user to validate whether the expected network usage requirements of their application will be satisfied at run-time by the platform. Users specify the network resources required by component ports and provided by node network links. As shown in Figure 21, the network requirements of a component port are specified as time intervals of the form $time, bandwidth, latency$, where the time value is relative



Name: ClientComponentDel

ReceptaclePortOut

NetworkProfile

```
% time (s), %bandwidth (bps), %latency (ms)
0,800000,0
1,850000,0
2,1024000,0
3,1000000,0
4,1005000,0
5,1050000,0
6,1100000,0
7,0,0
```

Figure 21 A component implementation's port and network profile.

to the start of a system period. In the case of an orbiting cluster of satellites, this period would be the orbital period of the satellites and the specified bandwidth for a given interval would be constant until the next specified interval. Similarly, the latency is specified on component ports as the maximum allowable latency for port data transmission during that interval. A sequence of such network resource requirement intervals is defined as a network profile.

The developer can apply the same network profile specification to the nodes' network. As shown in Figure 22 and Figure 23, on each network link, the developer specifies the network profile, for which the bandwidth specification indicates the minimum provided bandwidth over the interval, and the latency specification represents the maximum transmission latency incurred by traffic on that link. When the entire model is interpreted, the components' node associations are resolved and each node's components' profiles are aggregated. Using the methods described in [21], we can convolve these aggregate profiles

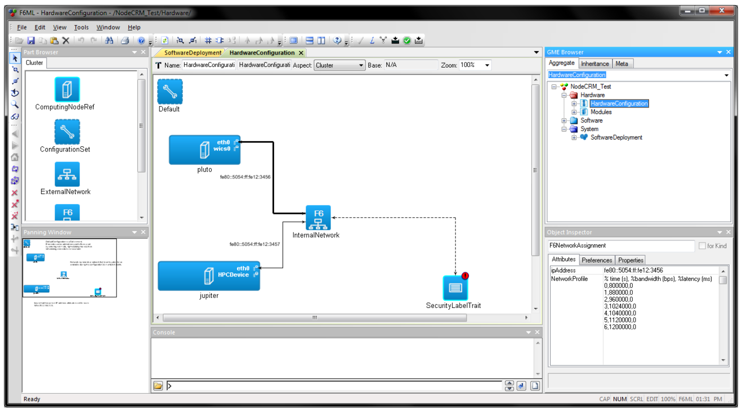
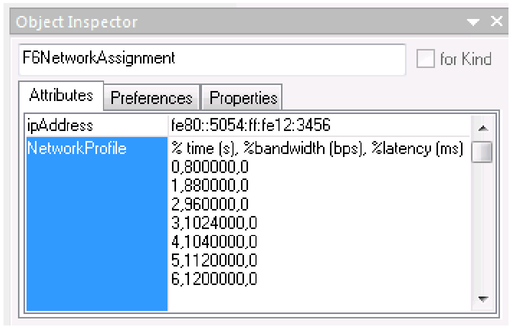


Figure 22 A node's network link and associated network profile attribute.



F6NetworkAssignment

ipAddress	fe80::5054:ff:fe12:3456
NetworkProfile	% time (s), %bandwidth (bps), %latency (ms) 0,800000,0 1,880000,0 2,960000,0 3,1024000,0 4,1040000,0 5,1120000,0 6,1200000,0

Figure 23 Zoomed-in view of the network profile in Figure 22.

with the node's link profile to determine (1) if the system can satisfy the network quality of service requirements of all the applications, (2) what are the remaining network resources, and (3) what network quality of service the applications will receive. All of this information is then reported back to the user in the form of a generated log file.

4.3.4. Scheduling analysis

Scheduling analysis is used to ensure that a valid schedule can be computed from the individual temporal paritions to which Actors are assigned. Recall that Actors are assigned to temporal partitions, each of which has a *period* and *duration*. The duration tells how long the Actor should execute, and the period tells how often the execution is repeated. For example, a partition with a period of 4ms and a duration of 2ms would execute for a total of 2ms every 4ms. Because an Application can consist of many Actors assigned to temporal partitions of different periods and durations, determining a repeating schedule for all temporal partitions that satisfies the period and duration requirements of each is a non-trivial task.

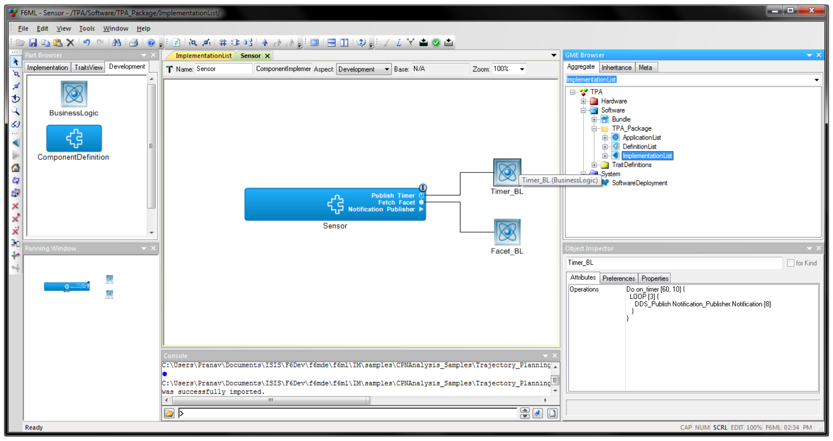
The scheduling analysis included with the modeling language computes a valid schedule by formulating a constraint satisfaction problem from the periodicity and duration requirements of all of the temporal partitions and providing this constraint satisfaction problem as input to an SMT solver [22]. If the solver finds a solution to this constraint problem, then this solution is parsed and stored inside the model. If a solution to the problem does not exist, the user is informed that the scheduling requirements of their temporal partitions cannot be satisfied.

4.3.5. Timing analysis of Component Operations

Components in DREMS communicate by requesting operations exposed through component interfaces. Component operation requests are enqueued into a component message queue from which operations are serviced one at a time. Each component thread is scheduled in a temporally partitioned OS scheduling scheme. As a real-time system, the order in which component operations execute is important. Each operation has a deadline on its execution time. Schedulability analysis at design-time ensures that every component in a deployed application completes its operations without violating deadlines.

In this regard, we have devised a Colored Petri Net-based [23] approach to modeling and analyzing component-based applications that we have integrated into DREMS ML. We assume a well-defined set of interaction semantics (such as the ones used by DREMS ML). DREMS ML captures the structural semantics of component-based applications in that a developer specifies properties such as (1) what a component is, (2) what the component ports are, (3) where each component is deployed, (4) what constitutes a process assembly. The business logic of each component operation is then written by an application developer after the modeling tools generate the necessary intermediate skeleton code for the various component operations. Our primary goal for this analysis was to be able to model these component operations within DREMS ML. In essence, the behavioral semantics of the component are captured by modeling the individual behaviors of the operations that each component would execute. Therefore, when a component A requests a remote operation on another component B, the request is enqueued on component B's message queue. When the dispatching thread of component B is scheduled, this operation is dequeued from the message queue and component B is triggered into execution. When component B is executed, it simply executes the sequence of steps written inside the business logic of the operation. Therefore, by modeling this operational behavior and the structure of the application, we have sufficient information to simulate the hierarchical scheduling nature of DREMS.

Figure 24 shows two business logic elements connected to a *Sensor* component. One of the elements connects to a component timer and contains the behavior of the timer-triggered callback executed by the Sensor. Figure 25 shows the *on_timer* operation corresponding to the on_timer business logic callback function written by a developer. This operation has a priority of 60 and a deadline of 10 ms. Once the timer triggers, the Sensor component uses the *Notification_Publisher* port to publish on the *Notification* topic taking 8



Name: Sensor
ComponentImplemer
Base: N/A

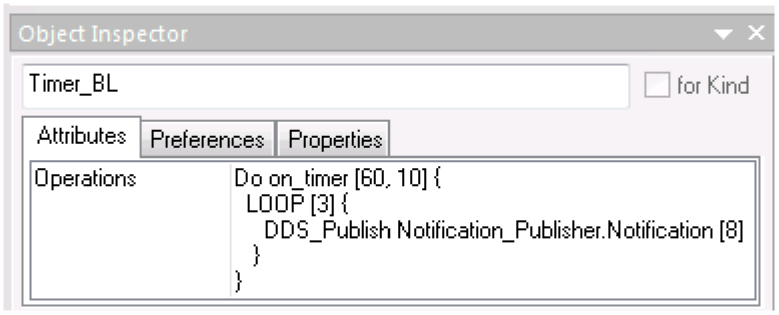
```
C:\Users\Pranav\Documents\ISIS\F6Dev\f6mde\f6ml\IM\samples\CPNAnalysis_Samples\Trajectory_Planning
●
C:\Users\Pranav\Documents\ISIS\F6Dev\f6mde\f6ml\IM\samples\CPNAnalysis_Samples\Trajectory_Planning
was successfully imported.
```

Timer_BL

Operations

```
Do on_timer [60, 10] {
  LOOP [3] {
    DDS_Publish Notification_Publisher.Notification [8]
  }
}
```

Figure 24 Component connected to two business logic elements



Timer_BL

Operations

```
Do on_timer [60, 10] {
  LOOP [3] {
    DDS_Publish Notification_Publisher.Notification [8]
  }
}
```

Figure 25 The business logic of the timer operation in Figure 24

ms. This publish operation occurs three times within the *LOOP*. This is a simple grammar-based representation of the business logic of the timer operation written by a developer.

Using model interpreters, this model is translated into a colored petri net-based analysis model. The analysis model captures the structural and behavioral properties of all applications within the DREMS ML model. The *places* in this CPN model contain *tokens* representing the state of system variables such as the component message queue, component thread states, offsets on component timers, system timer clock and component interactions. The transitions in this CPN model capture the nondeterministic set of events that can transpire in the simulation. This includes discrete events such as thread scheduling, thread blocking, operation requests and timer expiry. Using state space analysis techniques provided by the CPN Tools [24] tool suite, a bounded state space of operational behaviors is generated. User-defined queries can be generated from DREMS ML to verify system-level properties such as lack of deadline violations, deadlocks and bounded worst-case response times. This CPN-based analysis is described more fully in [25].

5. Evaluation

In this section, we will provide an evaluation of the DREMS Modeling Language through exploration of a simple but representative example from the F6 domain. This evaluation will be conducted along the development phases outlined in Section 4.

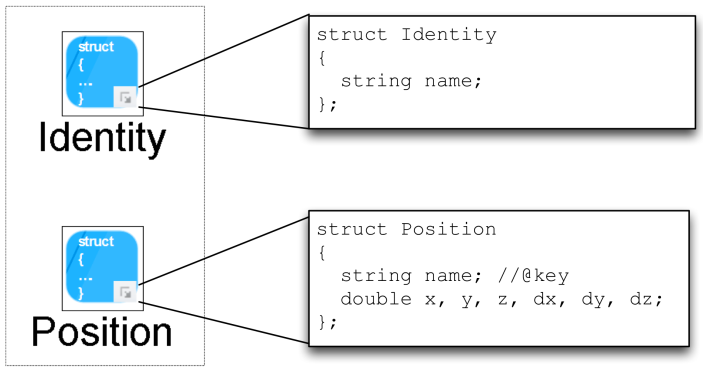
The Satellite Navigation Distribution Service (SATNAV) is a component application intended to demonstrate the salient details of the modeling language. SATNAV is intended to be a sub-application of, for example, a larger component application that calculates flight plans for the satellite to maintain cluster flight. This example has three key participants:

- **Distributor**: A component that collects readings from various sensors on the satellite bus that indicate the current position and velocity of the satellite. These readings are collated and published to interested consumers (receivers).
- **Continuous Receiver**: A component that is interested in all sensor updates published by the Distributor, and is thus provided with the full information read from the position sensors. This component consumes relatively more bandwidth than the On Demand Receiver (described below). Due to the increased bandwidth requirements, this component is most appropriately co-located on the same node as the distributor.
- **On Demand Receiver**: A component that may only be sporadically interested in new updates from the Distributor. This is intended to reduce the bandwidth requirements and thus only consumes the identifiers of sensors that have new data. When the component detects an update on a sensor it is interested in, it may call back on the distributor to obtain the full information.

The following subsections will describe the modeling and development of a system using this component application. The examples were created using the Generic Modeling Environment (GME)[26] configured to support DREMS ML. Note that in GME a model can have multiple *aspects*: visual views of the model that visualize selected subsets of the model elements.

5.1. Component development

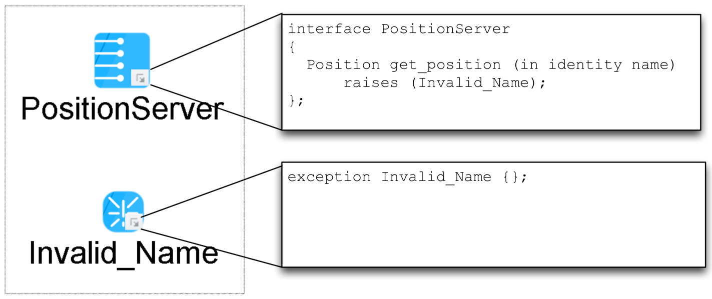
This section will describe the modeling language and implementation process from the perspective of the component developer through the following phases: data type and interface definition, topic definition, component and assembly definition, and finally component implementation.



```
struct Identity
{
  string name;
};
```

```
struct Position
{
  string name; //@key
  double x, y, z, dx, dy, dz;
};
```

Figure 26 SATNAV data types



```
interface PositionServer
{
  Position get_position (in identity name)
      raises (Invalid_Name);
};
```

```
exception Invalid_Name {};
```

Figure 27 SATNAV interfaces and exceptions

5.1.1. Data types and interface definitions

SATNAV has two key data types that are essential to its operation: a data type describing the full readings obtained from a sensor, and a data type that simply contains an identifier for a particular sensor. These are modelled by inserting a `struct` element into the model, shown in Figure 26, showing two structs: Identity, intended to identify a single sensor, and Position, intended to describe a reading from a single sensor. This element is populated through an IDL editor in which these structures may be described. The `//@key` comment following the `name` field of the `Position` structure indicates to the modeling tool that this field should be considered the key when constructing a stateful topic.

The interface used by the On Demand Receiver component to query the full update from a particular sensor, in addition to an exception that may be thrown if an invalid sensor identifier is passed to the operation, is also described in this location using a similar process: an exception and interface element is inserted into the model, and IDL defined using a similar process as before. This interface is named `PositionServer`.

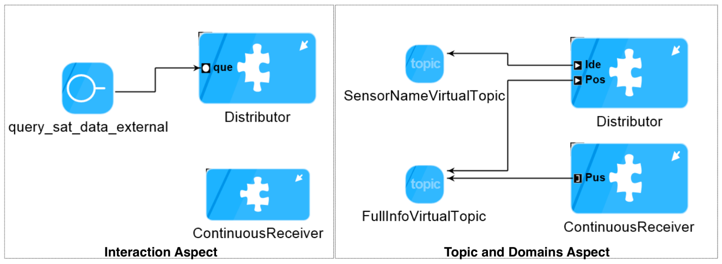


Figure 28 Component assemblies

5.1.2. Component implementation

The component implementation model describes the concrete implementations of the component definitions described earlier: this will result in the generation of skeleton code that the component developer may populate with business logic. This model is also used to associate artifacts, shared libraries and other configuration files required by the component implementations.

5.1.3. Component and assembly definitions

As described earlier, this example contains three components: a `Distributor`, a `ContinuousReceiver`, and a `OnDemandReceiver`. In the interest of brevity, we will describe modeling only the `Distributor` component; the other two are similarly modeled. We begin by inserting a component definition into the model and opening it. This component has three ports: a stateful publisher that uses the `SatFullInfo` topic defined earlier, a stateless publisher that uses the `SatName` topic, and an asynchronous RMI port that provides the `SatPosition` interface. A port representing each of these is placed into the component definition model and populated with the respective data type.

Once all three component definitions have been modeled, we create an assembly: a `LocalAssembly` that contains the `Distributor` and a `ContinuousReceiver`. The `OnDemandReceiver` will not be part of an assembly and will be modeled as part of the deployment later. This is accomplished by creating an assembly model, and inside inserting the two component definitions defined earlier that are intended to be part of this assembly. As shown in Figure 28, the ports defined inside these component definition models are now exposed on the components inside the assembly, allowing us to establish connections amongst the components.

In the Interaction aspect of the assembly model, we define connections between RMI ports of components; in this case, the `SatPosition` ports. Since the `Distributor` component, which provides this interface, and the `OnDemandReceiver`, which requires it, are not in the same assembly, we cannot directly establish this connection. Instead, we create external port on the `LocalAssembly` and delegate it to the `PositionServer` port on the `Distributor`, as shown in Figure 28.

A different aspect, 'Topics and Domains', is used to establish mapping be-

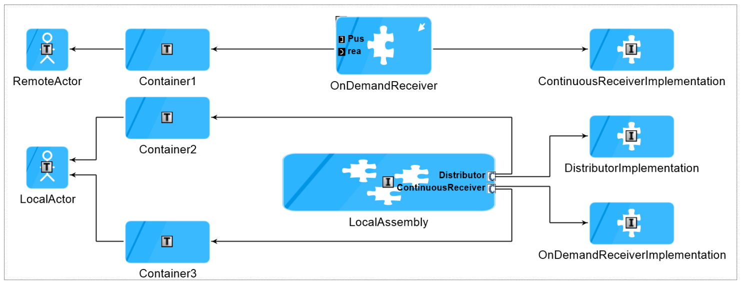


Figure 29 Application Deployment Aspect

tween publish/subscribe ports and topics. To accomplish this, we populate the assembly with two Virtual Topics, elements that are later assigned to concrete topics in the deployment model. A Virtual Topic is similar to a template parameter that has to be bound to a specific topic later. One Virtual Topic is used to represent the `SensorFullInfo` topic, and a connection is drawn from the stateful publish and subscribe ports on the `Distributor` and `ContinuousReceiver` references, respectively. Another Virtual Topic is used to represent the `SensorName` topic, and connections drawn between the stateless publication port on the `Distributor`. This is shown in Figure 28.

5.2. Application development

In this section, we describe the process of modeling an application, which is divided into three aspects:

- **Deployment**: Assemblies and components are instantiated, concrete component implementations are assigned, and components are assigned to actors.
- **Interactions**: Connections are established between external ports of components and assemblies.
- **Topics and Domains**: Concrete topics are assigned to Virtual Topics and other publish/subscribe ports.

5.2.1. Deployment aspect

In the Deployment aspect of the Application model, we accomplish two primary tasks. First, we associate concrete component implementations with the assemblies and component definitions that we wish to deploy. Second, we associate components and assemblies with containers (logical groupings of components) and actors.

We begin by inserting the assemblies and components that we wish to deploy into the model. In this example, we wish to deploy the `LocalAssembly` (containing the `Distributor` and `ContinuousReceiver`) and the `OnDemandReceiver` (which is not part of any assembly). Next, we insert implementations (modeled in Section 5.1.2). Component instances inside the assembly are exposed as

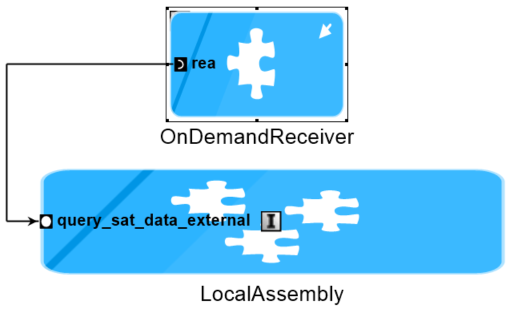


Figure 30 Application Interaction aspect

ports; we assign implementations by establishing a connection between the port and the desired implementation; for components deployed without an assembly, we establish a connection between the component and the desired implementation, as shown on the right hand side of Figure 29. Containers are assigned in a similar fashion, by creating a connection between the component (or port of an assembly) and the desired container. Containers are assigned to Actors in a similar fashion. This is shown on the left hand side of Figure 29.

5.2.2. Interactions aspect

The Interactions aspect of the Application model is shown in Figure 30. This aspect shows the components inserted into the application in Section 5.2.1, but hides the component implementations, containers, and actors. In this view, we create a connection between the `PositionServer` provided by the LocalAssembly and required by the OnDemandReceiver. The interactions aspect may also be used to expose ports to other applications in a manner similar to assemblies, described in Section 5.1.3; that capability, however, is outside the scope of this evaluation.

5.2.3. Topics aspect

The Topics aspect of the Application model is shown in Figure 31. This aspect shows pub/sub ports on components, and Virtual Topics of assemblies. In this view, we insert two concrete topics: `FullInfo` and `SensorName`. These concrete topics are associated with the subscriber port on the `OnDemandReceiver` and the virtual topics exposed by the `LocalAssembly`.

5.3. System integration

5.3.1. Software Packaging

In the software packaging model, the system integrator specifies the temporal partitions available in the system, assigns actors to temporal partitions (and by extension the components and containers within those actors), and may establish connections amongst external ports provided by the applications that are instantiated. This is accomplished with three aspects: Scheduling, which

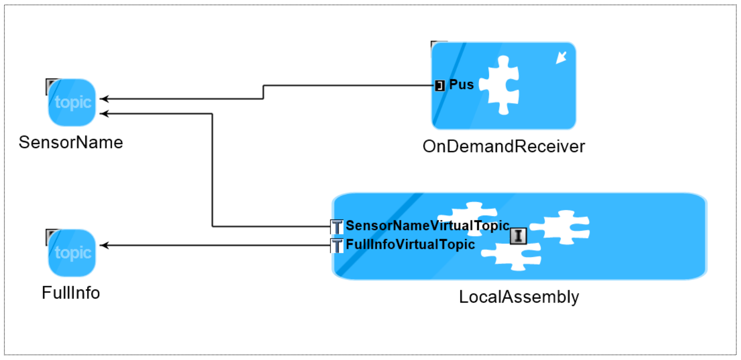


Figure 31 Application Topics aspect

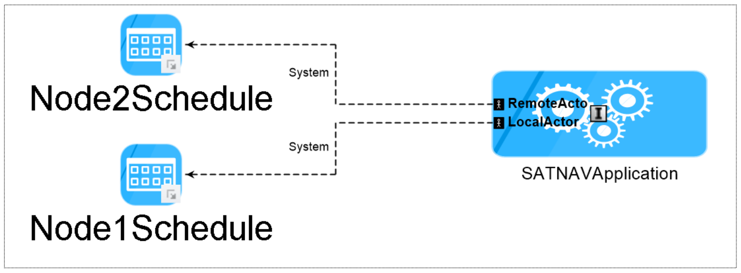


Figure 32 Software Packaging Scheduling aspect

handles specification and assignment of actors to schedules; Interactions, which handles connections between applications; and Topics, which allows further refinement of the publish/subscribe configuration of the system. The Interactions and Topics aspects are similar in function to those described in Section 5.2.2 and Section 5.2.3, and will not be described here.

In the Scheduling aspect, shown in Figure 32, partition schedules are modeled and assigned to actors. Software applications are placed in this model, and the actors contained within those application models are exposed as ports. Actors are assigned to schedules by creating a connection between the actor and the desired schedule.

5.3.2. Cluster configuration

In the Cluster Configuration model, the system integrator specifies the hardware configuration of the cluster: which hardware nodes are available, their configuration (network interfaces, available devices, etc.). Multiple cluster configuration models may be provided to represent how the cluster configuration is expected to evolve over time as satellites join and leave the cluster.

5.3.3. Software configuration

In the Software Configuration model, the system integrator specifies how software packages map to available hardware in cluster configuration models. Software packages are placed in this model; partition schedules present in the software packages are exposed as ports, which may be connected to cluster configurations present in this model. By mapping the software packages to one cluster configuration, it is possible to show how the software configuration changes when the cluster configuration is changed.

6. Related work

This section describes related work in the field of architecture description languages for real-time, embedded systems. We classify the work along two dimensions: those that pertain to standards, technologies and tools, and those that are related to research efforts that describe architecture description languages or use the standard technologies.

6.1. Standards and Technologies for Architecture Description Languages for Embedded Systems

The Architecture Analysis and Design Language (AADL) [5, 6] is a standard developed by the Society of Automotive Engineers. Originally developed for aerospace systems, the standard is applicable to the model-based specification and analysis of embedded real-time systems and systems of systems. It has comprehensive support for modeling a variety of component types and their interactions. Component abstractions in AADL consist of software components, computational hardware, and the overall system. Different interaction patterns between components are supported in AADL. Using AADL it is possible to conduct analysis for a variety of critical system properties, such as performance, schedulability and reliability.

Despite the comprehensive support offered by AADL for model-based specification and analysis of embedded real-time systems, for the system F6 we decided to address the problem by developing a completely new architecture design language. This decision stemmed from our preference for domain-specificity over generality as explained below. For instance, AADL aims to support the needs of a wide range of embedded real-time systems, making it a general-purpose architectural description language. Consequently, a component in AADL can be of different types including a process, thread, and thread group among other things. In contrast, in the IAP, a component has precise semantics, wherein an application developer understands a component to be unit of encapsulation for application business logic. In AADL it is possible to define one component type, such as a thread, and map it to the hardware resources to realize execution semantics. On the other hand, in the IAP, components cannot be directly mapped to hardware resources. They must first be composed together into actors, which in turn are allocated to the resources, and a collection of actors represents an application.

In the IAP, we support a variety of interaction patterns among components that may use different interaction paradigms such as call-return and publish/-subscribe, provide first class support for multiple levels of security and support an elaborate fault management scheme. These patterns can be easily extended with new interactions, if required. Additionally, the ports on the components support "programming by contract" with the help of pre- and post-condition checking.

Also note that the concept of an actor in IAP is slightly different from that of a traditional process. Moreover, scheduling of activities at the component-level and the actor-level occur at two different levels in our IAP. For example, although actors can support multiple threads, the run-time framework allows only one thread to execute at any given time in one component, i.e., at the component-level, the IAP enforces a per-component, one-thread-at-a-time approach to the scheduling of threads in each component. This decision was made to relieve application developers from having to use complex synchronization primitives and to avoid race conditions. These activities are in turn mapped to a partition at the actor-level and scheduled on the hardware using an ARINC653-style partition-based scheduling semantics.

Many of these key distinguishing features and semantics of the IAP are hard to realize with relative ease in AADL without substantial additional effort. We believe that although it is possible to extend as well as constrain generic modeling capabilities using techniques, such as stereotypes as used in UML or *annex* capabilities in AADL, we decided against this approach due to the additional efforts required in this process since such additional efforts and extensions may often incorporate *ad hoc* decisions, which ultimately may hinder the correct-by-construction realization of IAP applications. Therefore, our approach is based on using a domain-specific modeling language.

Other general-purpose approaches similar to AADL are OMG's SysML and OMG's MARTE profile for UML. SysML [7] is a general-purpose modeling language for systems engineering. SysML leverages a subset of the Unified Modeling Language (UML) while extending it with capabilities needed to model complex systems engineering problems, which is called the SysML profile for UML. The extensions enable engineering analysis. The Modeling and Analysis of Real-time and Embedded (MARTE) systems [9] is a UML profile to extend UML to support the model-driven development of real-time and embedded systems. Similar arguments we made on domain-specificity versus generality apply in the context of these standards, too, which made us design DREMS ML.

By no means do we discount the strengths of these standards, and our future work may involve automated transformations between DREMS ML and these standards so that we can leverage the extensive tool support and analysis capabilities that are commonly available with tools based on these standards. We believe such transformations will not be complicated since there are some similarities between DREMS ML and the standards. For example, hierarchical decomposition, and packaging are some common features available across all these technologies.

6.2. *Related Research on Architecture Description Languages for Embedded Systems*

In [27], the authors extend SysML with concepts borrowed from AADL by proposing the ExSAM profile. The key benefit derived from this exercise was the ability to model various kinds of system engineering concepts while at the same time be able to leverage the large set of existing AADL-based analysis tools. In effect their approach strengthens our argument towards building a domain-specific DREMS ML. Like the ExSAM project, interpreters in DREMS ML can transform the artifacts to AADL, wherever possible, to leverage the analysis tools in AADL.

The work described in [28] illustrates extensions to AADL using its Error Model annex feature to model and reason about errors including modeling of probalistic faults, how they propagate, recovery from failures, and degraded modes of operation ensuing from the faults. The resulting dialect of AADL developed by the authors is called SLIM (System-Level Integrated Modeling). While the area of fault modeling and reasoning as espoused and adopted in this work is very useful to our work in DREMS ML, this work also demonstrates the need for extensions to AADL to attain certain domain-specific objectives, which are intuitive to the system engineers. To that end our philosophy if DREMS ML is aligned with this work. In essence, DREMS ML is a form of architectural description language.

The EAST-ADL2 [29] project defines an architecture description language tailored towards automotive embedded systems. In particular, its goals are to capture in one place all the artefacts of an embedded systems including requirements, features, behaviors, and software and hardware components. It also includes dependencies stemming from decisions that must be made in the context of various refinements, allocation decisions, composition and communication. EAST-ADL2 has been developed to work in concert with AUTOSAR [30] to provide a complete and effective development environment for automative systems starting all the way from conception all the way to implementation. EAST-ADL2 is a domain-specific language built using a UML2 profile. EAST-ADL2 demonstrates an effort that does not use AADL, however, still supports an architectural description language for automative systems. Like EAST-ADL2, DREMS ML is also a domain-specific language built using a metamodeling language provided by our GME tool that is based largely on UML. Like EAST-ADL2, DREMS ML also works in concert with the IAP runtime architecture, which is used to provide a "cluster-as-a-service" capability to distributed, real-time and embedded systems.

A survey on architectural description languages is described in [31]. A key motivation for this work was to understand the best practices in ADLs. Among the key findings, the authors recommend that any ADL have better support for communication among different stakeholders. In tune with these recommendations, DREMS ML has support for multiple different developers to use the language to build the system as a series of enhancements. Another key recommendation is that any ADL should be simple, pragmatic, and support collaboration instead of been heavy-weight. The design of DREMS ML is aligned with this philosophy.

Although DREMS is built on OMG standards, the Fractal Initiative [32] and ProCom [33] aim at a similar language-independent component specification framework. The modeling language itself is tied to our component model, but the philosophy of the component specification chain (definition-instance-implementation) can simplify the design over these other component models as well.

With regard to analysis for architecture description languages, there are a number of formal tools that perform behavioral analysis on architecture description languages, such as [34, 35, 36, 37]. DREMS ML includes analysis for syntactic and well-formedness checks (Section 4.3.1), security analysis (Section 4.3.2), resource usage analysis (Section 4.3.3) and scheduling analysis (Section 4.3.4).

7. Conclusions

We introduced a novel, wide spectrum architecture design language for the modeling, development, integration, verification, deployment and maintenance of component-based, distributed real-time embedded applications. The salient features of the language are: (1) integrated domain-specific modeling languages to support all developmental activities, (2) a software component model with precisely defined execution semantics that allows the compositional construction of complex applications, (3) reliance on industry standards for a wide range of component communication and interaction patterns, (4) automatic generation of all implementation and deployment artifacts (except the component business logic code) from a single source, (5) support for complex system integration activities, including verification and testing. The language is defined with the help of a metamodel and a prototype implementation is in use by flight software developers today.

Further development work on the language will include: (1) models for supporting fault management in the deployed system, (2) models for quality of service properties (requirements and capabilities), (3) integration with verification and validation tools. Further application domains (beyond System F6) will be also considered, where complex distributed real-time applications are needed.

References

[1] T. Levendovszky, A. Dubey, W. Otte, D. Balasubramanian, A. Coglio, S. Nyako, W. Emfinger, P. Kumar, A. Gokhale, G. Karsai, Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems, Software, IEEE 31 (2) (2014) 62–69.

[2] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, P. Bose, A Software Platform for Fractionated Spacecraft, in: Proceedings of the IEEE Aerospace Conference, 2012, IEEE, Big Sky, MT, USA, 2012, pp. 1–20.

[3] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, J. Willemsen, F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment, in: Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13), Paderborn, Germany, 2013.

[4] I. Coutts, J. Edwards, Model-driven distributed systems, Concurrency, IEEE 5 (3) (1997) 55–63. doi:10.1109/4434.605919.

[5] P. H. Feiler, D. P. Gluch, J. J. Hudak, The Architecture Analysis & Design Language (AADL): An Introduction, Tech. Rep. ADA455842, DTIC Document (2006).

[6] P. Feiler, B. A. Lewis, S. Vestal, The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems, in: Computer Aided Control System Design, 2006, pp. 1206–1211. doi:10.1109/CACSD-CCA-ISIC.2006.4776814.

[7] Object Management Group, Systems Modeling Language (SysML), Version 1.3, Object Management Group, OMG Document formal/2012-06-01 Edition (Jun. 2012).

[8] M. Hause, et al., The SysML Modelling Language, in: Fifteenth European Systems Engineering Conference, Vol. 9, 2006.

[9] Object Management Group, UML Profile for MARTE: Modeling And Analysis of Real-Time Embedded Systems, Version 1.1, Object Management Group, OMG Document formal/2011-06-02 Edition (Jun. 2011).

[10] J. Sztipanovits, G. Karsai, Model-integrated computing, Computer 30 (4) (1997) 110 –111. doi:10.1109/2.585163.

[11] A. Dubey, A. Gokhale, G. Karsai, W. Otte, J. Willemsen, A Model-Driven Software Component Framework for Fractionated Spacecraft, in: Proceedings of the 5th International Conference on Spacecraft Formation Flying Missions and Technologies (SFFMT), IEEE, Munich, Germany, 2013.

[12] Object Management Group, DDS for Lightweight CCM Version 1.0 Beta 2, Object Management Group, OMG Document ptc/2009-10-25 Edition (Oct. 2009).

[13] W. R. Otte, A. Gokhale, D. C. Schmidt, J. Willemsen, Infrastructure for Component-based DDS Application Development, in: Proceedings of the 10th ACM international conference on Generative programming and component engineering, GPCE '11, ACM, New York, NY, USA, 2011, pp.

53–62. doi:http://doi.acm.org/10.1145/2047862.2047872.
URL `http://doi.acm.org/10.1145/2047862.2047872`

[14] ARINC Incorporated, Annapolis, Maryland, USA, Document No. 653: Avionics Application Software Standard Inteface (Draft 15) (Jan. 1997).

[15] F. Cassez, K. Larsen, The impressive power of stopwatches, in: C. Palamidessi (Ed.), CONCUR 2000 - Concurrency Theory, Vol. 1877 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2000, pp. 138–152.

[16] I. Ripoll, R. Ballester-Ripoll, Period selection for minimal hyperperiod in periodic task systems, IEEE Transactions on Computers 62 (9) (2013) 1813–1822. doi:http://doi.ieeecomputersociety.org/10.1109/TC.2012.243.

[17] A. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing domain-specific design environments, Computer 34 (11) (2001) 44–51. doi:http://dx.doi.org/10.1109/2.963443.

[18] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[19] D. E. Bell, L. J. LaPadula, Secure computer systems: Mathematical foundations, Technical Report 2547, Volume I, MITRE (1973).

[20] G. Birkhoff, Lattice Theory, 3rd Edition, Colloquium Publications, American Mathematical Society, 1967.

[21] W. Emfinger, G. Karsai, A. Dubey, A. Gokhale, Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks, in: Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems, CyPhy '14, ACM, New York, NY, USA, 2014, pp. 44–47. doi:10.1145/2593458.2593459.
URL `http://doi.acm.org/10.1145/2593458.2593459`

[22] L. M. de Moura, N. Bjørner, Z3: An efficient smt solver, in: TACAS, 2008, pp. 337–340.

[23] K. Jensen, L. M. Kristensen, Coloured Petri Nets - Modelling and Validation of Concurrent Systems, Springer, 2009.

[24] A. V. Ratzer, L. Wells, H. M. Lassen, M. Laursen, J. F. Qvortrup, M. S. Stissing, M. Westergaard, S. Christensen, K. Jensen, Cpn tools for editing, simulating, and analysing coloured petri nets, in: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets, ICATPN'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 450–462.
URL `http://dl.acm.org/citation.cfm?id=1760066.1760097`

[25] Colored Petri Net-based Modeling and Formal Analysis of Component-based Applications.
URL http://ceur-ws.org/Vol-1235/paper-10.pdf

[26] The ISIS Model Integrated Computing (MIC) Toolsuite.
URL http://www.escherinstitute.org/Plone/tools/suites/mic

[27] R. Behjati, T. Yue, S. Nejati, L. Briand, B. Selic, Extending sysml with aadl concepts for comprehensive system architecture modeling, in: R. France, J. Kuester, B. Bordbar, R. Paige (Eds.), Modelling Foundations and Applications, Vol. 6698 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 236–252. doi:10.1007/978-3-642-21470-7_17.
URL http://dx.doi.org/10.1007/978-3-642-21470-7_17

[28] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, Safety, dependability and performance analysis of extended aadl models, The Computer Journal 54 (5) (2011) 754–775. arXiv:http://comjnl.oxfordjournals.org/content/54/5/754.full.pdf+html, doi:10.1093/comjnl/bxq024.
URL http://comjnl.oxfordjournals.org/content/54/5/754.abstract

[29] P. Cuenot, P. Frey, R. Johansson, H. Lönn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. T. Kolagari, M. Törngren, et al., The EAST-ADL Architecture Description Language for Automotive Embedded Software, in: H. Geise, G. Karsai, E. Lee, B. Rumpe, B. Schatz (Eds.), Model-Based Engineering of Embedded Real-Time Systems, LNCS 6100, Springer, 2011, pp. 297–307.

[30] Autosar GbR, AUTomotive Open System ARchitecture, http://www.autosar.org/.
URL http://www.autosar.org/

[31] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, A. Tang, What Industry Needs from Architectural Languages: A Survey, Software Engineering, IEEE Transactions on 39 (6) (2013) 869–891. doi:10.1109/TSE.2012.74.

[32] G. Blair, T. Coupaye, J.-B. Stefani, Component-based architecture: the fractal initiative, Annals of Telecommunications 64 (1) (2009) 1–4.

[33] T. Bureš, J. Carlson, I. Crnkovic, S. Sentilles, A. Vulgarakis, Procom–the progress component model reference manual, Mälardalen University, Västerås, Sweden.

[34] J. Magee, J. Kramer, D. Giannakopoulou, Behaviour analysis of software architectures, in: Software Architecture, Springer, 1999, pp. 35–49.

[35] X. He, J. Ding, Y. Deng, Model checking software architecture specifications in sam, in: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM, 2002, pp. 271–274.

[36] P. Pelliccione, P. Inverardi, H. Muccini, Charmy: A framework for designing and verifying architectural specifications, Software Engineering, IEEE Transactions on 35 (3) (2009) 325–346.

[37] M. Y. Chkouri, A. Robert, M. Bozga, J. Sifakis, Translating aadl into bip-application to the verification of real-time systems, in: Models in Software Engineering, Springer, 2009, pp. 5–19.