A Quantitative Framework for Predicting Resource Usage and Load in Real-Time Systems based on UML Models

Vahid Garousi, Lionel Briand and Yvan Labiche
Software Quality Engineering Laboratory (SQUALL)
www.sce.carleton.ca/squall
Department of Systems and Computer Engineering, Carleton University
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
{vahid, briand, labiche}@sce.carleton.ca

Abstract

This paper presents a quantitative framework for predicting resource usage and load in Real-Time Systems (RTS). The prediction is based on an analysis of UML 2.0 sequence diagrams, augmented with timing information, to extract timed-control flow information. It is aimed at improving the predictability of a RTS by offering a systematic approach to predict system behavior in each time instant during its execution. Since behavioral models such as sequence diagrams are available in earlier design phases of the software life cycle, the framework enables resource analysis at a stage when design decisions are still easy to change. We use network traffic as an example resource to illustrate the approach. Usage and load analysis of other resources (such as CPU, memory and database) can be performed in a similar fashion. A case study illustrates the feasibility of the approach.

Keywords: Resource usage prediction, load analysis, load forecasting, resource overuse detection, real-time systems, UML.
# Table of Contents

## Table of Contents

- **1 Introduction** .................................................................................................................. 4
- **2 Background** .................................................................................................................. 4
  - 2.1 UML Profile for Schedulability, Performance, and Time ................................................. 5
  - 2.2 Control Flow Analysis ................................................................................................... 5
- **3 Related Works** ................................................................................................................ 7
- **4 Model-Based Predictability Analysis: An Overview** ..................................................... 8
- **5 Resource Usage Analysis** .............................................................................................. 9
  - 5.1 Definitions ........................................................................................................................ 9
    - 5.1.1 Formalizing Sequence Diagram Messages ................................................................. 9
    - 5.1.2 Network Deployment Diagram and Network Interconnectivity Tree ......................... 10
  - 5.2 Resource Usage Definition ............................................................................................. 11
  - 5.3 Resource Usage Measure .............................................................................................. 12
    - 5.3.1 Effect of Inheritance ................................................................................................. 13
    - 5.3.2 Indeterminism in Messages Sizes ........................................................................... 13
  - 5.4 Resource Usage Query .................................................................................................. 14
    - 5.4.1 Location: Objects, Nodes or Networks .................................................................... 14
    - 5.4.2 Direction (for nodes only): In, Out, or Bidirectional .............................................. 14
    - 5.4.3 Traffic Types: Data Traffic or Number of Messages .............................................. 14
    - 5.4.4 Duration: Instant or Interval .................................................................................... 14
  - 5.5 Resource Usage Analysis Functions ............................................................................. 15
    - 5.5.1 Naming Convention ................................................................................................. 15
    - 5.5.2 Traffic Location: Network ...................................................................................... 16
    - 5.5.3 Examples ................................................................................................................ 16
  - 5.6 Resource Usage Analysis of Other Resource Types ...................................................... 17
    - 5.6.1 CPU ....................................................................................................................... 17
    - 5.6.2 Memory .................................................................................................................. 19
- **6 Load Forecasting** ........................................................................................................... 20
  - 6.1 Load Forecasting Query .............................................................................................. 21
  - 6.2 Load Forecasting Heuristics .......................................................................................... 21
  - 6.3 Load Forecasting Functions .......................................................................................... 22
    - 6.3.1 Naming Convention ................................................................................................. 22
    - 6.3.2 Load Location: Network ....................................................................................... 23
- **7 Reusing MBRUA Principles for Additional Applications** ............................................. 23
  - 7.1 Detecting Resource Overuse .......................................................................................... 23
  - 7.2 Detecting Illegal Concurrent Access to Critical Objects .............................................. 24
- **8 Case Study** .................................................................................................................... 24
  - 8.1 The Case Study System and its UML Model ................................................................. 25
  - 8.2 Predicting Resource Usage Pattern ............................................................................... 26
    - 8.2.1 Control Flow Analysis of SCAPS SD ...................................................................... 26
    - 8.2.2 Traffic Prediction and Measurement ...................................................................... 26
    - 8.2.3 Comparison Results .............................................................................................. 28
  - 8.3 Detecting Resource Overuse ....................................................................................... 29
- **9 Conclusions** .................................................................................................................. 30

## Acknowledgements

Carleton University TR SCE-06-05 June 2006
1 INTRODUCTION

Real-Time Systems (RTS) are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plan systems [1]. However as discussed in [1], their development and testing is difficult as engineers have to account for real-time constraints.

An important part of RTS verification and testing is to analyze how different resources (e.g. network bandwidth, memory, CPU) are utilized after deployment in the field. If a system overuses a resource (overload conditions) or uses it in an invalid manner (e.g. violating mutual exclusion) at any time instant, functional or non-functional (e.g. performance) failures are inevitable. Furthermore, controlling the way that software consumes resources is an important concern for software executing on embedded devices such as smart cards where memory is limited [2].

Predictability Analysis (PA) [3-5] aims at analyzing a RTS behavior before the system is functioning in the field. It entails various activities that RTS developers perform to assure that the systems they develop are safe, reliable, maintainable, and satisfy their time constraints. Examples of such activities are: analysis of mutual exclusion, resource usage analysis, resource usage management, load forecasting, and schedulability analysis. Several model-based predictability analysis techniques have been proposed in the literature [3, 5, 6]. However, few of them [6] are based on Unified Modeling Language (UML) [7] models. Since the UML is gaining popularity in the RTS community, model-based PA techniques exploiting UML models are needed. However, to the best knowledge of the authors, there has been no such work reported, which enables early PA by exploiting UML models developed for other purposes (e.g. documentation, testing and code generation).

This paper presents a quantitative framework, referred to as Model-based Resource Usage Analysis (MBRUA), based on an analysis of control flow in UML 2.0 sequence diagrams [8], to predict resource usage and load in a RTS. These behavioral models, augmented with timing information, are used to extract timed-control flow information and improve the predictability of a RTS by offering a systematic approach to predict system behavior in each time instant during its execution.

The UML 2.0 Sequence Diagram (SD) metamodel [7] provides various program-like constructs such as conditions (using alt combined fragment operator), loops (using loop operator), and procedure calls (using interaction occurrence construct). These constructs result into different Control Flow Paths (CFP), resulting in different system behavior (i.e., sequences of messages or events). Therefore, a comprehensive model-based PA technique which aims at predicting system behavior should take into account the different CFPs of SDs. Predictability analysis in our approach is achieved by first analyzing this control flow and then the timing information of the SD messages to perform a time-based predictability analysis, which can be used to predict a RTS’s behavior in each time instant during its execution. Two predictability analysis methods are discussed in this paper: resource usage analysis and load forecasting. Discussions are illustrated with examples and a case study is performed to demonstrate the feasibility of the proposed approach.

The rest of this article is structured as follows. Section 2 provides background information, which includes an overview of the sequence diagram control flow analysis technique we use. A survey of related works is presented in Section 3. Section 4 presents an overview of our predictability analysis technique. Prediction of resource usage is discussed in Section 5. Load forecasting is described in Section 6. Section 8 presents the set up and results of a case study. Finally, Section 9 concludes the article and discusses some of the future research directions.

2 BACKGROUND

To model timing information in behavioral models, the Object Management Group (OMG) has proposed the UML Profile for Schedulability, Performance, and Time (UML-SPT) [9]. We provide an overview of this profile in Section 2.1. We then summarize in Section 2.2 the technique we have developed in [8] to perform control flow analysis of UML 2.0 sequence diagrams.
2.1 UML Profile for Schedulability, Performance, and Time

Since its adoption as a standard, the UML has been used in a large number of time-critical and resource-critical distributed systems, for instance [10-14]. Based on this experience, a consensus has emerged that, while being useful, UML is lacking some modeling notations in key areas that are of particular concern to Real-Time distributed system designers and developers. In particular, it was noticed that the lack of a quantifiable notion of time and resources was an obstacle to its broader use in the distributed and embedded domains. To further standardize the use of UML in modeling complex distributed systems, the OMG adopted a new UML profile named “UML Profile for Schedulability, Performance and Time” (UML-SPT) [9].

The UML-SPT profile proposes a framework for modeling real-time systems using UML. The profile is now popular in the research community [6, 15-17] and is getting accepted in industry [18]. “The profile provides a uniform framework, based on the notion of quality of service (QoS), for attaching quantitative information to UML models” [9]. Specifically, QoS information represents, either directly or indirectly, the physical properties of the hardware and software environments of the application represented by the model. An example SD annotated with timing information using the UML-SPT is shown in Figure 1.

As defined by the UML-SPT, the «RTstimulus» stereotype models a timed stimulus and can be attached to action executions that generate stimuli (such as call action and method). RTArrivalPattern tagged-value is used to specify the type of the arrival pattern (e.g., a periodic stimulus) and its characteristics (e.g., the period value). RTstart and RTend tagged-values indicate the start and end time instances of a stimulus. For example, as it is modeled in Figure 1, the first message of the SD has a periodic arrival pattern which fires every 100 ms. The first getData() message is triggered at 1 ms and finishes at 2 ms. The time origin (t=0) in the UML-SPT profile [9] (and thus in the current work) is assumed to be the start time of the SD execution.

2.2 Control Flow Analysis

As we discussed in an earlier work [8], the UML 2.0 SD metamodel [7] provides various program-like constructs such as conditions (using alt combined fragment operator), loops (using loop operator), and procedure calls (using interaction occurrence construct). Furthermore, asynchronous messages and parallel combined fragments model parallelism inside SDs. These constructs result into different control flow paths (CFP), resulting in different system behavior, which should be taken into account in a comprehensive model-based PA technique.

We presented a model-based CFA technique in [8] to analyze control flow in SDs. An overview of the technique is illustrated using the activity diagram in Figure 2. A Concurrent Control Flow Graph (CCFG) is our Control Flow Model for SDs. If we consider the UML 2.0 SD metamodel [7], asynchronous messages and the par (parallel) interaction operator entail concurrency in a SD. However, such concurrency cannot be analyzed by conventional Control Flow Graphs (CFG) [19].
A CCFG is generated for each SD. A node and an edge in a CCFG correspond to a message (in the corresponding SD), and the transition of control flow between messages, respectively. In cases where a SD calls (refers to) another SD, there are control flow edges connecting their corresponding CCFGs to form an Inter-SD CCFG, which is a concept similar to the inter-procedural CFG [19]. For example, the CCFG in Figure 4 corresponds to the SD in Figure 3.

The algorithm to map the SD to the CCFG is discussed in detail in [8]. The mapping rules are described in OCL (Object Constraint Language) and relate different parts of an instance of a SD metamodel to different parts of an instance of the CCFG metamodel. For example, one of our fourteen mapping rules is defined between an InteractionFragment (usually referred to as a SD) and an Activity (usually referred to as an activity diagram) which checks if there exists an instance of activity (a CCFG) for every instance of interaction fragment (a SD).

Note that the UML-SPT notations and the node tagged-value in Figure 3 are not required by our CFA technique. They have been used to model the deployment node of each object and messages’ timing information, as this SD will be used as our running example in Section 5.

The concept of Concurrent Control Flow Paths (CCFPs) [8] is similar to conventional Control Flow Paths (CFPs), except that they consider concurrent control flows in SDs as they are derived from CCFGs [8]. Each CCPF is made of several message nodes of a CCFG. We presented a grammar in [8] to represent all the CCFPs from a CCFG. An infinite number of CCFPs can be derived from the CCFG in Figure 4 because of the loop involving nodes (i.e., messages) H and I. Applying a strategy often used when deriving control flow paths from source code, we try to bypass the loop (if possible), take it only once, a representative or average number, and a maximum number of times. This procedure generates the four CCFPs shown in Figure 5. The symbol $\rho$ will denote CCFPs in the remainder of this article. These four possible paths are $\rho_1$ (the loop is bypassed), $\rho_2$ (the loop is taken once), $\rho_3$ (the loop is taken a representative number of times, specifically twice), and $\rho_4$ (the loop is taken a maximum number of times, specifically three times).
When messages of a SD are annotated with timing information, we refer to such CCFPs as Timed Concurrent Control Flow Paths (TCCFP). The concept of TCCFP is defined to emphasize the inclusion of timing information in CCFPs. All the CCFPs in Figure 5 are also TCCFPs.

### 3 Related Works

There have been several model-based predictability analysis techniques for RTS in the literature (e.g. [3, 5, 6]). However, to the best knowledge of the authors, there has been no reported work on the predictability of RTS based on the control flow of UML behavioral models to predict resource usage and load. Furthermore, none of the existing works focuses specifically on the prediction of resource usage and load based on models, but rather tackle other types of PA activities, e.g., validation and performance evaluation [3, 6], and schedulability analysis [5]. In addition, though there is a body of related works (e.g., [6]) which perform PA activities based on some form of legal behaviors of a system under analysis, e.g., based on scenario diagrams (a simplified sequence diagram which has only one CFP), PA based on a comprehensive CFA of behavioral UML models still needs to be addressed.

Bernardi et al. [6] report on the use of UML 1.x sequence diagrams and statecharts for the validation and the performance evaluation of systems. The authors assume that the system is specified as a set of statecharts and that sequence diagrams are used to represent “executions of interest”. It is argued that UML 1.x lacks a formal semantics and hence it is not possible to apply, directly, mathematical techniques on UML models for system validation. To reach this goal, the authors propose an automatic translation of sequence diagrams and statecharts into Generalized Stochastic Petri-nets, special types of Petri-nets used for performance evaluation, and a composition of the resulting Petri-net models into a set of mathematical models suitable for performing two types of analysis: (1) Correctness analysis and (2) Performance analysis. The former analysis verifies that the scenario represented by a SD is admissible in the sense that there exists at least a set of inputs which, when fed to the model, fires the path (in the composed model) specified by the scenario. Performance analysis is performed by measuring several metrics, defined in the article, which are computed by measuring the crossing time of tokens between different places. The idea of translating UML models to Generalized Stochastic Petri-nets is similar to our CFA approach, but it only considers scenario diagrams (SDs with one CFP) while the current work takes into account the program-like constructs (e.g., if, loop) in UML 2.0 SDs. Furthermore, this work does not address the issue of resource usage and load analysis.

Feiler et al. [3] present a model-based architectural approach for improving predictability of performance in embedded RTS. The approach is component-based and utilizes an automated analysis of task and communication architectures to provide insight into schedulability and reliability during design. The MetaH language and toolset from Honeywell [20] is used as the modeling language in the approach. The work is based on runtime task and communication architectural models in the MetaH language, and does not analyze
behavioral models to perform predictability analysis. The authors note that partitioning (resulting from the use of MetaH architectural models) enforces timing protection, i.e., satisfaction of timing constraints, in DRTSs, and thus yields more accurate predictability.

Yau and Zhou [5] presented an approach to incorporate schedulability analysis into existing frameworks for model-based software development. The goal is to improve the predictability of a RTS and increase the capability of model refinements and code generation using the schedulability analysis results. The authors propose a new diagram in the context of UML-based RTS development, referred to as scheduling reference diagrams. A scheduling reference diagram models the timing and schedulability aspects of distributed tasks in a system. Scheduling reference diagrams are generated from timing requirements and collaboration diagrams, and are used to conduct schedulability analysis. The work does not discuss how the complex control flow structures (such as alternatives and loops) are handled. It is thus difficult to compare it with our approach, which is based on UML 2.0 SDs. Furthermore, the authors focus on schedulability analysis and do not comment on other analyses such as resource and load forecasting.

4 MODEL-BASED PREDICTABILITY ANALYSIS: AN OVERVIEW

The activity diagram of Figure 6 provides an overview of our Model-Based Predictability Analysis (MBPA) approach. The diagram conforms to the general model-processing framework, proposed by the UML-SPT profile [9], where our technique acts as a model processor that takes a UML model as input and generates the analysis results.

Our technique takes as inputs the UML model of a system and a set of resource-dependent inputs (for resource usage analysis and load forecasting). The behavioral model (SDs augmented with timing information using UML-SPT profile modeling features [9]) is used to predict the behavior of the System Under Analysis (SUA). The structure model (class diagrams) is used to find out about the generalization (inheritance) relationships among classes to be able to appropriately handle polymorphic behaviors of objects in SD lifelines when analyzing control flow [8]. Furthermore, as it will be discussed in Section 5, resource usage analysis will use the internal structure of classes in the SUA.

The technique then analyzes the control flow in the input model (Section 2.2) and uses the resulting Timed Concurrent Control Flow Paths (TCCFP) for resource usage analysis and load forecasting. The concepts of resource usage definition, measure and query are specific input parameters for driving the resource usage analysis and will be further discussed in Section 5. A resource usage definition (RUD) is a set of criteria which defines how the usage of a resource should be quantified from behavioral models. For example, a possible RUD for network traffic is to only consider messages sent across different nodes in a distributed system. A
resource usage measure (RUM) defines how to quantify the usage of a specific resource entailed by executing source code corresponding to a set of model elements. A RUM can be considered as a function from a set of model elements to Real values. For example, considering network traffic or memory as a resource, a possible RUM is to compute data size of objects (sum of attribute sizes) transmitted by a message over a network. A Resource Usage Query (RUQ) is a querying mechanism to filter the RUA results. Network Deployment Diagrams (NDD) in Figure 6 are models to describe the distributed architecture (Section 5.1.2) of a SUA, a piece of information required for network traffic analysis and prediction, which is the type of RUA we focus on in this paper. A load forecasting query and a sequence diagrams schedule are specific input parameters for the load forecasting activity, and will be discussed in Section 6. A Sequence Diagrams Schedule is a set of specific SD’s DTCCFPs and their start time, i.e., the time at which they are triggered during system execution.

Our MBPA technique is flexible as it can be easily adapted and applied to other resource types (e.g. CPU, memory, disk, and database) as it is defined in terms of the concepts of Resource Usage Definition (RUD) and Resource Usage Measure (RUM), which can be specified for each specific resource type. For example, we will present in Section 5.6 the RUD and RUM for resource types CPU and memory.

5 RESOURCE USAGE ANALYSIS

As discussed in Section 2.2, TCCFPs of a SD are generated by control flow analysis. TCCFPs are then used by the Resource Usage Analysis (RUA) along with a set of resource-specific inputs: Resource Usage Definition (RUD), Resource Usage Measure (RUM) and Resource Usage Query (RUQ).

We discuss in the next sections how the usage of network traffic, as an example resource, can be analyzed in a RTS. Section 5.1 provides a set of fundamental definitions and concepts which will be used for network traffic usage analysis. The specific RUD, RUM, and RUQ for network traffic are presented in Section 5.2, Section 5.3, and Section 5.4, respectively. We then define in Section 5.5 a set of traffic usage measures which are classified based on the query attributes in a RUQ. To demonstrate the applicability of the approach to other resource types, Section 5.6 briefly presents the RUD and RUM for CPU and memory.

5.1 Definitions

A set of definitions are presented in this section, which will be used for network traffic usage analysis.

5.1.1 Formalizing Sequence Diagram Messages

As discussed in Section 4, in order to perform RUA, our technique needs to manipulate SD messages. Thus, in order to precisely define how we perform RUA, we provide a formal representation for SD messages. Similar to the tabular notation for UML 2.0 SDs in Appendix D.1 of [7], each SD message, in the design model of a distributed system, can be denoted as a tuple:

\[ \text{message}=(\text{sender, receiver, methodOrSignalName, msgSort, parameterList, returnList, startTime, endTime, msgType}) \]

where

- **sender** denotes the sender of the message and is itself a tuple of the form \( \text{sender}=(\text{object, class, node}) \), where:
  - **object** is the object (instance) name of the sender.
  - **class** is the class name of the sender.
  - **node** is where the sender object is deployed.
- **receiver** denotes the receiver of the message and is itself a tuple of the same form as **sender**.
- **methodOrSignalName** is the name of the method or the signal class name on the message.
- **msgSort** (message sort) is the type of communication reflected by the message, as defined in UML 2.0 [7]. It can be either **synchCall** (synchronous call), **synchSignal** (synchronous signal), **asynchCall** (asynchronous call), or **asynchSignal** (asynchronous signal).
- **parameterList** is the list of parameters for call messages. **parameterList** is a sequence of the form \( <(p_1, C_1, \text{in/out}), ..., (p_n, C_n, \text{in/out})> \), where \( p_i \) is the \( i \)-th parameter name of class type \( C_i \) and \( \text{in/out} \) defines the kind of
the parameter. For example if the call message is \( m(o_1:C_1, o_2:C_2) \), then the ordered parameters set will be \(<(o_1, C_1, \text{in}), (o_2, C_2, \text{in})>\). If the method call has no parameter, this set is empty.

- **returnList** is the list of return values on reply messages. It is empty in other types of messages. UML 2.0 assumes that there may be several return values for a reply message. We show returnList in the form of a sequence \(<(\text{var}_1=\text{val}_1, C_1), \ldots, (\text{var}_n=\text{val}_n, C_n)>\), where \(\text{val}_i\) is the return value for variable \(\text{var}_i\) with type \(C_i\).

- **startTime** is the start time of the message (modeled by UML-SPT profile’s \(\text{RTstart}\) tagged value).

- **endTime** is the end time of the message (modeled by UML-SPT profile’s \(\text{RTend}\) tagged value).

- **msgType** is a field to distinguish between signal, call and reply messages. Although the \(\text{messageSort}\) attribute of each message in the UML metamodel can be used to distinguish signal and call messages, the metamodel does not provide a built-in way to separate call and reply messages. Further explanations on this and an approach to distinguish between call and reply messages can be found in [8].

As an example of this formalism, message \(\text{getData()}\) in the SD in Figure 1 is represented as \((sdc, \text{SensorDataCollector}, -), (s1, \text{Sensor}, -), \text{getData}, \text{synchCall}, \text{null}, \text{null}, \text{'1ms'}, \text{'2ms'}, \text{'Call'})\). Using this formalism, different fields of a message are accessed using the record notation. For example, given a message \(m\), \(m.\text{sender.object}\) refers to the sender object of message \(m\).

5.1.2 Network Deployment Diagram and Network Interconnectivity Tree

Network Deployment Diagram (NDD) is a model to describe the distributed architecture of a SUA and is specifically needed for the RUA in this article (network traffic resource type). NDDs are an extension to UML package structures, where the entire system network is the root (high level) package and other networks and nodes are the sub-packages modeled in a hierarchical manner. An example of a hierarchical distributed system and the corresponding NDD is shown in Figure 7-(a) and (b), respectively.

A **Network Interconnectivity Tree (NIT)** is a data structure built from a NDD. The motivation for NITs is to easily identify the subset of nodes and networks that are relevant for deriving dependency between any two given nodes and the network path between them. The root of the tree is always the entire system network while system networks and nodes are its children. In a NIT, networks and nodes are shown as rectangles and circles, respectively. For example, the NIT of the NDD in Figure 7-(b) is shown in Figure 7-(c).

![Figure 7-(a): An example of a hierarchical distributed system. (b): The corresponding Network Deployment Diagram (NDD). (c): The corresponding Network Interconnectivity Tree (NIT). (d): Deriving the network path between two nodes.](image)

To identify the network path between any two given nodes, we define the network path function \(\text{getNetworkPath}(n_s, n_r)\) where \(n_s\) and \(n_r\) are the sender and the receiver nodes of a message, respectively. An
algorithm for this function can be found in [8, 21]. For example, the derivation of the network path between \(n_1\) (the sender) and \(n_4\) (the receiver) is depicted in Figure 7-(d) and is formally represented as:

\[
\text{getNetworkPath}(n_1, n_4) = \langle \text{network}_1, \text{SystemNetwork}, \text{network}_2 \rangle
\]

A simplification that has been made in the \text{getNetworkPath} function is that we assume there is only one path between any two given nodes. But there can be several paths (routes) between two nodes in a network. Routing algorithms are usually used to maximize the network bandwidth and/or balance load in large networks by sending data through various paths between two nodes. This issue will be addressed in future work so as to account for several paths between any two given nodes.

5.2 Resource Usage Definition

Resource Usage Definition (RUD) is a set of criteria to select relevant model elements for a specific resource. One possible RUD for network traffic usage is to filter TCCFPs by removing their local messages (sent between two objects on a local node). A formal definition of this RUD is given in Equation 1.

\[
\text{RUD}_{\text{network}} : \text{TCCFP} \rightarrow \text{DTCCFP} \\
\forall \rho \in \text{TCCFP}, \text{RUD}_{\text{network}}(\rho) = \rho - \{ \text{msg} \mid \text{msg} \in \rho \land \text{msg.sender.node} = \text{msg.receiver.node} \}_{\text{Local messages in } \rho}
\]

Equation 1-RUD of the network traffic usage analysis technique.

A Distributed TCCF (DTCCFP) is a TCCFP where all messages are distributed. A DTCCFP is built from a given TCCFP \(\rho\) by removing all local messages and keeping the distributed ones. As an example, let us assume the CCFPs in Figure 5. In order to derive their DTCCFPs, we should first determine if each message is local or distributed. According to the corresponding SD (Figure 3), all the messages except \(A\) and \(B\) are distributed. Hence, using the RUD in Equation 1, the DTCCFPs corresponding to the TCCFPs in Figure 5 are shown in Figure 8. Note that TCCFPs and DTCCFPs are types of CCFPs, which are ordered sequences of concurrent message.

To better explain the output of a RUD and to visualize DTCCFPs, we present a timed inter-node representation of DTCCFPs. UML 2.0 introduces a new interaction diagram called Timing Diagrams (Section 14.4 in [7]). As defined by UML 2.0: “Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Timing diagrams focus on conditions changing within and among lifelines along a linear time axis.” We use the basic concepts of UML 2.0 timing diagrams and propose a model for timed inter-node representation of DTCCFPs. The representation is a 2-dimentional chart where the x-axis is a linear time axis and the y-axis is the set of all nodes referenced at least once by the messages of a given DTCCFP.

For example, let us consider the SD in Figure 3 and the DTCCFP(\(\rho_1\)) in Figure 8. A timed inter-node representation of DTCCFP(\(\rho_2\)) is shown in Figure 9, where arrowheads correspond to the type of messages (synchronous/asynchronous call or reply). The start and end times of the messages are extracted from the SD.

![Figure 9-Timed inter-node representation of DTCCFP(\(\rho_2\)) in Figure 8.](image-url)
5.3 Resource Usage Measure

A Resource Usage Measure (RUM) is a function to measure the usage of a resource by the model elements selected by a RUD. A RUM can be considered as a function from the set of model elements to Real values. We define a RUM for network traffic usage in Equation 2.

\[ RUM_{\text{network}}(msg) \] returns the network data traffic value entailed by message \( msg \) (regardless of traffic location on a specific network). The most data centric parts of a call, a reply, and a signal message are \( \text{parameterList} \), \( \text{returnList} \), and the attributes of the corresponding signal class, respectively. \( \text{CallDT} \), \( \text{ReplyDT} \) and \( \text{SignalDT} \) are the amount of Data Traffic (DT) for a call, reply, or signal message. \( \text{CallDT} \), DT for a call message, is calculated by the summation of data sizes of all the attributes of each parameter class. For a reply message, DT is calculated by the summation of data sizes of all the attributes of each class in the return list. \( \text{SignalDT} \), DT for a signal message, is equal to the data sizes of all attributes of the signal class referred to by the message. The data carried by a signal message is represented as attributes of the signal instance.

To estimate the data size of a set of objects, we add up data sizes of all classes in the set. Let us define the data size of a class to be the total sizes of its attributes in bytes. Therefore the total size of the classes in a \( \text{parameterList} \) and \( \text{returnList} \) can be an estimate for the data sizes of call and reply messages. Admittedly, other measures (perhaps more accurate) of network traffic can be considered. For example, a more accurate estimate would account for the extra data added by the lower layers of the OSI (Open Systems Interconnection) network model—such as data link and physical—to the data submitted by the application layer of the OSI model. Such an estimation requires a detailed, platform-specific analysis of packet and frame structures in different layers of the OSI model. We however expect the extra data to represent a small percentage of the network traffic, an assumption that will be verified in our case study. (Section 8).

\[
\begin{align*}
\forall msg \in \text{Message} : RUM_{\text{network}}(msg) &= \begin{cases} 
\text{CallDT}(msg) & \text{if } msg.\text{msgType} = \text{"Call"} \\
\text{ReplyDT}(msg) & \text{if } msg.\text{msgType} = \text{"Reply"} \\
\text{SignalDT}(msg) & \text{if } msg.\text{msgType} = \text{"Signal"}
\end{cases} \\
\text{CallDT}(msg) &= \sum_{C|\exists i.c_{i}\in \text{msg.parameterList}} \text{dataSize}(C_i) \\
\text{ReplyDT}(msg) &= \sum_{C|\exists i.c_{i}\in \text{msg.returnList}} \text{dataSize}(C_i) \\
\text{SignalDT}(msg) &= \text{dataSize}(msg.\text{methodOrSignalName}) \\
\forall C \in \text{classDiagram} \text{: dataSize}(C) &= \sum_{a_i\in C.\text{attributes}} \text{dataSize}(a_i)
\end{align*}
\]

Equation 2-RUM to analyze network traffic usage.

Dash (-) in Equation 2 indicates that a field can take any arbitrary value (a “don’t care” field). Note the format of \( \text{parameterList} \) and \( \text{returnList} \), as mentioned in Section 5.1. \( msg.\text{parameterList} \) and \( msg.\text{returnList} \) is the sequence of parameters (returns) for a call (reply) message. The function \( \text{dataSize}(C_i) \) returns the data size of the class \( C_i \). \( \text{classDiagram} \) is the set of classes (it can be extracted from the system’s class diagram). \( \text{attributes} \) denotes the set of attributes of class \( C \) (again, it can be extracted from the system’s class diagram). \( \text{size}(a_i) \) is the size of an attribute \( a_i \) of class \( C \), which can be calculated based on attribute types. If the attribute type is an atomic type, e.g., \( \text{int}, \text{long}, \text{bool} \), its size (in bytes) can be found in the specification of the programming language used to develop the system. For example, the data sizes of primitive Java data types \( \text{short}, \text{int}, \text{long} \) are two, four and eight bytes in Java, respectively. In case an attribute \( a_i \) of a class is itself an object, the size of that attribute, \( \text{size}(a_i) \), will be the size of its class and can be calculated recursively using Equation 2. As an example, suppose a call message \( msg_1 \) with \( \text{parameterList} = \langle o_1.A, o_2.B \rangle \), where classes \( A \) and \( B \) are defined in the class diagram of Figure 10. Using the class specifications of \( A \) and \( B \), we can estimate the size of the message \( msg_1 \) as:

\[
\text{size}(msg_1) = \text{size}(A) + \text{size}(B) = (8*(100+500)) + (8*(100+500)+8*400) = 12.8\text{KB}
\]
5.3.1 Effect of Inheritance

When estimating the data size of a class (and the messages using it), it is important to take into account the inheritance relationships in which the class is involved. This might affect the size of the messages making use of that particular class because of dynamic binding.

For example, suppose the method signature of a method \( m \) to be \( m(o_1,o_2:A):A \), which basically means that two parameters of class type \( A \) (defined in Figure 10) are passed to method \( m \) and an object of type \( A \) is returned. Since \( B \) and \( C \) are both sub-classes of \( A \), an object of type \( B \) or \( C \) can also be an argument of method \( m \), which in this case will cause the message to have different data sizes depending on which subclass becomes an argument of the method in a particular execution. This is because classes \( B \) and \( C \) both define an extra attribute and there is uncertainty on which subclass, if any, might be an argument. At least two approaches can be taken to analyze the data size of such a message:

1. One may calculate the data size of all the classes in such inheritance relationships (classes \( A \), \( B \) and \( C \) in the above example) and then pick the maximum value as data size. For example, the maximum data size of the message \( m \) (above) using this approach can be calculated as follows (\( \text{dataSize}(m) \) only counts the two parameters, thus multiplication factor 2, since the return value is not part of the size of the call message, but part of the return message):
   \[
   \text{dataSize}(m) = 2 \times \max(\text{dataSize}(A), \text{dataSize}(B), \text{dataSize}(C))
   \]
   \[
   = 2 \times (8 \times (100+500), 8 \times (100+500+400), 8 \times (100+500+200)) = 16 \text{ KB}
   \]

2. An approach where the probabilities of a superclass and its subclasses to be the run-time type of an argument (given in an operational profile for instance) are taken into account. For example, assume an operational profile which specifies the following probabilities: \( p(A)=0.6, p(B)=0.3, p(C)=0.1 \). The expected data size of the message \( m \) (above) using this approach can then be calculated as:
   \[
   \text{dataSize}(m) = p(A) \cdot \text{dataSize}(A) + p(B) \cdot \text{dataSize}(B) + p(C) \cdot \text{dataSize}(C)
   \]
   \[
   = 0.6 \times 8 \times (100+500) + 0.3 \times 8 \times (100+500+400) + 0.1 \times 8 \times (100+500+200) = 5.92 \text{ KB}
   \]

5.3.2 Indeterminism in Messages Sizes

There can be two possibilities regarding the data size of a class: fixed and variant (indeterministic). A class has a fixed data size if it has attributes with fixed data sizes such as \( \text{int} \) (four bytes) and \( \text{short} \) (two bytes) in C++. We saw that the data sizes of messages with such classes as parameters or return values can be estimated using Equation 2. Conversely, there might also be parameters or return values whose sizes can not be measured precisely. For example, an input parameter of a call message might be of type, say, \( \text{String} \) in C++. The size of such an object may change depending on the length of the string assigned to it. Therefore, Equation 2 can not be applied to estimate the data size of a message in those cases. Other data structures such as linked lists, hash tables, and trees also belong to the set of data types with indeterministic data sizes.

One simple approach to estimate data size in this latter case is to make use of statistical distributions of size values instead of single values. Statistical distributions of the size of such messages can be estimated by monitoring size across a representative set of runs. The larger the number of runs, the closer the sample distribution to the actual, unknown statistical distribution. Runtime monitoring techniques (such as [22]) can
be applied to monitor and derive such distributions. Depending on context, the expected value, maximum, or any other quantile of the distribution can be used as the estimated data size of a message.

5.4 Resource Usage Query

A Resource Usage Query (RUQ) is a querying mechanism for RUA used to focus on some particular aspects of the resource usage in a SUA. For the network traffic usage analysis, we define four query attributes:

- Traffic location: objects, nodes or networks (Section 5.4.1)
- Traffic direction (for nodes only): in, out, or bidirectional (Section 5.4.2)
- Traffic type: data traffic or number of messages (Section 5.4.3)
- Traffic duration: instant or interval (Section 5.4.4)

A RUQ can be a combination of the above four query attributes. For example, a RUQ can be the following: What is the data traffic (type) over the system network (location) in time interval 1ms to 10ms (duration)? We discuss the above four query attributes in more detail next. We then define a class of traffic usage functions for DTCCFPs which are classified based on the query attributes. These functions compute the output of our RUA technique.

5.4.1 Location: Objects, Nodes or Networks

If we leave out the intermediate network nodes (such as routers and gateways), network traffic can essentially go through two places in a system: nodes or networks. In a typical distributed message scenario, the message is initiated by the sender node and then travels along a network path from the sender to the receiver node. A network path (Section 5.1.2) consists of one or more networks in the system. Finally, the message arrives at the destination node, where it is supposed to be handled appropriately. Furthermore, since each node can have more than one object, the source/destination of each message can be different objects on different nodes. Therefore, we see objects as a nested level of traffic inside nodes.

5.4.2 Direction (for nodes only): In, Out, or Bidirectional

As discussed above, traffic location can be either a network, an object, or a node. In case of a node or an object, we can think of three measurements in terms of traffic direction: in, out or bidirectional. This is due to the fact that a node or an object is an end point of traffic in the system. Since a network in the system only relays the traffic, i.e., it transmits the traffic to other networks/nodes, we consider only one traffic direction for networks: bidirectional. For brevity, when we talk about traffic for networks in this article, we implicitly refer to bidirectional traffic.

5.4.3 Traffic Types: Data Traffic or Number of Messages

In our context, network traffic can be characterized by at least two relevant types:

1. The amount of data, and
2. The number of distributed messages

For example, consider a simple system made up of two nodes $n_A$ and $n_B$. Node $n_A$ might rarely communicate with $n_B$, but when sending a message, $n_A$ sends a large amount of data to $n_B$. On the other hand, $n_B$ frequently sends queries to $n_A$, and gets replies. However each request from $n_B$ to $n_A$ and the corresponding reply have small data sizes. Therefore, both the volume of data and the frequency of messages are relevant to us. For example, in stress testing distributed systems based on RUA [23], our results from past case studies have revealed that different nodes and networks in a SUT might exhibit faults related to network traffic if stress test requirements are derived with either of the above two traffic types.

5.4.4 Duration: Instant or Interval

Network traffic analysis can be performed in two time-related fashions: network traffic in a time instant or a time interval.
As a DTCCFP may include several messages, it can have different usage levels of network traffic in different time instants, depending on which of its message is executing. We can also add up instant duration traffic values over a given time interval. Analyzing traffic over a time interval might be interesting, for instance when the saturation of a specific resource over a time interval is the main concern for developers. For example, performing a RUA over time intervals can be used to assure that network buffers of a SUT will never overflow.

5.5 Resource Usage Analysis Functions

We define a set of traffic usage functions for DTCCFPs which are classified based on these query attributes defined in 5.4. The naming convention of the functions is discussed in Section 5.5.1. For brevity, only formal definitions of resource usage functions for network usage location are presented in Section 5.5.2. The functions for node and object usage locations are derived in a similar fashion and are presented in Appendix A. Section 5.5.3 illustrates the definitions with examples.

5.5.1 Naming Convention

A tree structure denoting the traffic functions’ naming convention and their input parameters is shown in Figure 11. The root node of the tree has a null label. A function name is determined by traversing the tree from the root to a leaf node and concatenating all the node labels in order.

Figure 11-Naming convention and input parameters of the traffic usage analysis functions.

Five layers are shaded in the tree. The top four layers correspond to the four query attributes discussed in Section 5.4. By counting the number of paths from the root node of the tree to leaf nodes, we get 28 paths (4 for networks, and 12 for node and object categories each), resulting in 28 different traffic functions.

The bottom layer in Figure 11 specifies the input parameters of a traffic function whose name is determined by traversing from the root to a leaf node. For example, consider the path specified by the bold line in Figure 11. This path represents function NetInsDT. According to the bottom layer, the input parameters of this function is \((p, net, t)\). This function returns the instant \(\text{Ins}\) data traffic value \(\text{DT}\) of a given DTCCFP \(\rho\) for a given network \(\text{net}\) at a given time \(t\). Input parameters including \(\text{int}\) in the bottom layer of Figure 11 correspond to the functions with interval duration. The start and end times of an interval, i.e., \(\text{int}=(\text{start, end})\), must be provided for such functions. For functions with node or object traffic location, the input parameters include either \(\text{nod}\) or \(\text{obj}\) as traffic location, respectively.

More detailed descriptions of the functions are given next. Functions with network traffic location are described in Sections 5.5.2. Similar functions can be defined for node and object traffic location.
5.5.2 Traffic Location: Network

1. \(\text{NetInsDT}(\rho, \text{net}, t)\) returns the instant data traffic for DTCCFP \(\rho\) on network \(\text{net}\) at time \(t\).

\[
\text{NetInsDT}(\rho, \text{net}, t) = \begin{cases} 
\sum_{\substack{\text{msg, node} \\
\text{msg, start} \leq t \leq \text{msg, end} \land \text{net} \in \text{getNetworkPath} \left( \text{msg, sender.node, msg, receiver.node} \right) \land \text{dur}(\text{msg}) \geq 0}} 
\end{cases}
\]

where \(\text{dur}(\text{msg})\) returns the time duration of a message: \(\text{dur}(m) = m\text{.endTime} - m\text{.startTime}\). Since a message can span over several time units, data traffic value of a message within a time unit is its total data size divided by its duration, which yields the message’s traffic per time unit. This is an approximation as packets could be sent at various rates for a message. As a simplification, we assume a uniform distribution of traffic during the duration of a message.

2. \(\text{NetInsMT}(\rho, \text{net}, t)\) returns the instant message traffic for DTCCFP \(\rho\) on network \(\text{net}\) at time \(t\).

\[
\text{NetInsMT}(\rho, \text{net}, t) = \begin{cases} 
\text{msg} \mid \text{msg, node} \land \text{msg, start} \leq t \leq \text{msg, end} \land \text{net} \in \text{getNetworkPath} \left( \text{msg, sender.node, msg, receiver.node} \right) \land \text{dur}(\text{msg}) \geq 0 \end{cases}
\]

3. \(\text{NetIntDT}(\rho, \text{net}, \text{int})\) returns the data traffic for DTCCFP \(\rho\) on network \(\text{net}\) over time instant interval \(\text{int}\). \(\text{NetIntDT}\) can be calculated using \(\text{NetInsDT}\).

\[
\text{NetIntDT}(\rho, \text{net}, \text{int}) = \sum_{\text{t=\text{int.start}..\text{int.end}}} \text{NetInsDT}(\rho, \text{net}, t)
\]

4. \(\text{NetIntMT}(\rho, \text{net}, \text{int})\) returns the message traffic for DTCCFP \(\rho\) on network \(\text{net}\) over time interval \(\text{int}\).

\[
\text{NetIntMT}(\rho, \text{net}, \text{int}) = \sum_{\text{t=\text{int.start}..\text{int.end}}} \text{NetInsMT}(\rho, \text{net}, t)
\]

5.5.3 Examples

An example is given here to illustrate how the value of a network traffic usage function is calculated. Suppose DTCCFP \(\rho=\text{cm}1\text{cm}2\text{rm}1\text{rm}2\) where \(\text{cm}\) and \(\text{rm}\) are call and reply messages, respectively, and are defined in the messages’ format (Section 5.1.1) as follows:

\[
\begin{align*}
\text{cm}1 &= ((o1,c1,n1), (o2,c2,n2), t(), \text{synchCall}, <(p1:-),(p2:-)>, \text{null}, 1, 2, \text{‘Call’}) \\
\text{cm}2 &= ((o2,c2,n2), (o3,c3,n3), u(), \text{synchCall}, <(p3:-),(p4:-)>, \text{null}, 3, 5, \text{‘Call’}) \\
\text{rm}1 &= ((o3,c3,n3), (o2,c2,n2), \text{null}, \text{null}, <(x=u(),-)>, 8, 9, \text{‘Reply’}) \\
\text{rm}2 &= ((o2,c2,n2), (o1,c1,n1), \text{null}, \text{null}, <(y=t(),-)>, 12, 13, \text{‘Reply’})
\end{align*}
\]

A dash (-) in the above messages definitions indicates that a field can take any arbitrary value (a “don’t care” field). Let us assume the system NIT in Figure 12. Further assume that the sizes of the four messages of DTCCFP \(\rho\) are calculated using the RUM in Equation 2 yielding the following values: \(\text{cm}1\) (90 KB), \(\text{cm}2\) (80 KB), \(\text{rm}1\) (30 KB), and \(\text{rm}2\) (50 KB). The following traffic functions can then be calculated.

- \(\text{NetInsDT}(\rho, \text{Network1}, t)\):
5.6 Resource Usage Analysis of other Resource Types

We briefly mention here how the RUA technique described in the previous sections for network traffic can be adapted to other resource types (e.g. CPU, memory, disk, and database). The RUA activity of our Model Based Predictability Analysis (MBPA) approach is flexible as it identifies specific concepts (Resource Usage Definition (RUD) and Resource Usage Measure (RUM) which can be tailored to each specific resource type, thus providing a framework for all types of resources. For example, we present below the RUDs and RUMs for resource types CPU and memory. Note that the time-based RUA functions (similar to the ones in Section 5.5 for network usage) for these resource types are not addressed here and should be investigated later based on the presented RUDs and RUMs.

5.6.1 CPU

Our heuristics for calculating CPU usage using SD messages are as follows. Among call, reply and signal messages, only call and signal messages consume CPU power. This is a simplification of the fact that reply (return) messages only return values to the caller of a message and the CPU usage entailed by reply messages is considered negligible. As an example, let us consider a call message \(cm\) and its corresponding reply message \(rm\). The receiver object starts to execute a method body upon receiving the message \(cm\) from the caller. Such an execution consumes CPU and finally the receiver returns a reply message \((rm)\) to the caller. The CPU usage entailed by \(rm\) is negligible compared to the execution of \(cm\), since it is basically copying the return results to a stack and returning back the control to the caller. The practical impact of such simplification and more accurate measures should however be investigated in future work.

The CPU usage of each call message depends on the processing complexity of the operation of the message, which can be either (1) predicted (calculated by a performance tool), (2) measured (if an executable implementation of the SUA is available), (3) required (coming from the system requirements or from a performance budget based on a message, e.g., a required response time for a scenario), or (4) assumed (based on experience) by modelers. These four alternatives denote the source of the information and are defined in the Performance Modeling section of the UML–SPT [9]. The prediction of processing complexity of an operation is
a challenging task in the early design phase. Existing works such as [24, 25] have proposed ways to predict CPU utilization using different analysis models. The prediction in [24] is specific to rule-based systems\(^1\) (for instance used in telecommunication applications) and is done using a model of the CPU cost per rule, referred to as a capacity model. The work in [25] uses fuzzy logic and the concept of stochastic processes to predict CPU utilization of mainframe systems based on a log file from the recent system behavior. Thus, the technique requires the implementation or at least an earlier version of the system to be available. The authors propose the predicted CPU utilization values to help system programmers tune a system’s performance by moving out the unimportant jobs during the peak time. Another CPU utilization estimation heuristic was proposed by Gomaa [26] in which a developer tries to implement pieces of code that are representative enough of the foreseen implementation. The CPU utilization of the “early” code is then measured to estimate the final code’s operation utilizations.

The Performance Modeling section of the UML–SPT [9] discusses ways to model CPU usage in behavioral models. For example consider the SD in Figure 13, where message \texttt{op()} is annotated with the \texttt{PAdemand} stereotype from the Performance Modeling section of the UML–SPT. \texttt{PAdemand} (PA for Performance Analysis) is used to model the resource demand of a scenario step (e.g., a message in a SD, or an activity in an activity diagram). The right hand side of a \texttt{PAdemand} equality should be of type \texttt{PAperfValue} (Performance Value), which is used to specify a complex performance value as defined below.

\[
\text{PAperfValue}=(\text{source-modifier} \?, \text{type-modifier} \?, \text{time-value} \?)
\]

where: \texttt{<source-modifier> ::= \text{req} | \text{assm} | \text{pred} | \text{msr}} is a string that defines the source of the value meaning respectively: required, assumed, predicted, and measured. \texttt{<type-modifier> ::= \text{mean} | \text{sigma} | \text{kth-mom}, <Integer> | \text{max} | \text{percentile}, <real> | \text{dist} is a specification of the type of value meaning: average, variance, kth-moment (integer identifies value of k), percentile range (real identifies percentage value), probability distribution. \texttt{<time-value>} is a time value described by the \texttt{RTtimeValue} type (defined by in the General Time Modeling section of the UML–SPT). Thus, \texttt{PAdemand} annotation of the message \texttt{op()} in Figure 13 means that it is predicted that this message utilizes 90% of the CPU (on \texttt{n}2).

Based on our heuristics for CPU usage by SD messages, we present the RUD and RUM for CPU resource analysis in Equation 3, where function \texttt{CPUUsage(message)} returns the processing complexity value associated with a message. CPU usage is specific to messages and not operations involved in those messages. The reason is that the same operation used in different sequence diagrams for instance may entail different CPU usages as

\footnote{Rule-based systems represent knowledge in terms of a set of rules that tell the system what it should do or what it could conclude in different situations. A rule-based system consists of a set of If-Then-Else rules, a set of facts, and some interpreter controlling the application of the rules, given the facts [27].}
different computations may be triggered. As discussed above, for the sake of simplification, the \( RUD_{CPU} \) does not consider reply messages, but only call messages.

The \( RUM_{CPU} \) of a message is simply equal to the processing complexity value of the operation associated with the message. It is important when analyzing CPU usage to consider event occurrences\(^2\) in SDs. For example consider the SD in Figure 13, and assume \( op() \) is the only message considered in the RUA according to \( RUD_{CPU} \). Since the processing of \( op() \) starts on object \( o_2 \) at event occurrence \( e_{o_1} \) and finishes at \( e_{o_2} \), the message consumes CPU power only in the time period between two event occurrences. Note that this notion of resource usage over time is not incorporated in Equation 3, since RUD and RUM are by nature not time-based. The issue should be further investigated when designing a set of time-based RUA functions for CPU usage (similar to the ones in Section 5.5 for network usage).

\[
\begin{align*}
RUD_{CPU}: & \ TCCFP \to TCCFP \\
\forall \rho \in TCCFP: & \ RUD_{CPU}(\rho) = \rho - \{msg \mid \text{msg} \in \rho \land \text{msgType} = \text{'reply'}\} \\
\forall \text{msg} \in \text{Message}: & \ RUM_{CPU}(\text{msg}) = \text{CPUUsage(msg)}
\end{align*}
\]

Equation 3-RUD and RUM for CPU resource.

Another important consideration when analyzing CPU usage in distributed systems is the locality of the usage. Similar in concept to the location attribute of network traffic (Section 5.4.1), the CPU usage location denotes the particular CPU on which a message is processed. For example, as shown in Figure 13, \( o_1 \) and \( o_2 \) are deployed on nodes \( n_1 \) and \( n_2 \). Therefore, the actual execution of operation \( op() \) takes place on \( n_2 \) and leads to CPU usage on \( n_2 \) only. The locality aspect of CPU usage is important because it is crucial for engineers to determine the host CPU which must handle the processing load of a message in a DRTS. Furthermore, we made a simplification in this section that the CPU utilization of message during its execution is uniform. A more realistic approach will be to define a time-based function, which will predict a message’s CPU utilization at each time instance during its execution.

### 5.6.2 Memory

Our heuristics for calculating memory usage by SD messages is as follows. Memory is used by messages in two ways:

- Messages which associated method or signal name is create or destroy, or
- Temporary (heap) memory used by local variables as a result of message invocations.

For example, consider the SD in Figure 14. Object \( o_1 \) creates an object of class \( C_3 \) and destroys it after sending a message \( (m_2) \) to it and receiving a reply \( (r_2) \). Thus, temporary (heap) memory corresponding to the data size of \( C_3 \) is allocated and then de-allocated. Furthermore, assume the source-code implementation of messages \( m_1 \) and \( m_2 \) results in 10 and 20 integer local variables, respectively. Assuming that each integer variable consumes four bytes of memory, invocation of \( m_1 \) and \( m_2 \) will consume 40 and 80 bytes of heap memory. Estimating such information may be possible in late design stages by using, for example, heuristics similar to the ones used in the COMET (Concurrent Object Modeling and Architectural Design with UML) [26] object-oriented life cycle, where Gomaa proposed a heuristic to estimate time durations of messages based on benchmarks made of previously-developed similar messages. Such an approach can be adapted to the estimation of number and types of local variables of a method by comparing the functionality/role of a method at hand to benchmarks of similar messages.

previously-developed methods local variables (in the same target programming language). It should be acknowledged that this is in general a complex task which would need lots of experience and skills from developers. Such information should then be provided by modelers in an appropriate way, for example by using specific tagged-values.

![Figure 14-Memory usage analysis example.](image)

Based on our heuristics for CPU usage by SD messages, we present the RUD and RUM for memory usage in Equation 4, where function `dataSize(class)` returns the data size of a class (Section 5.3). We consider the amount of memory allocated/de-allocated in `RUM_Memory`. A `create` message allocates memory space (denoted with +), while a `destroy` message releases memory (denoted with -). Note that, for simplicity, the temporary (heap) memory used by local variables resulting from message invocations has not been incorporated into RUD or RUM. The temporary memory allocated in the beginning of an operation by its local variables will be de-allocated upon return from the operation. Based on this simplification, the granularity of the RUA which will use the RUD and RUM in Equation 4 is assumed to be at the message level, and thus the invocation of operations with local memory usage will not cause any change in the amount of memory space consumed. However if a time-based RUA is to be performed, time-based RUD and RUM should be defined where the intra-message-invocation memory usage should also be accounted for. Furthermore, a part of such a RUA technique should make sure that there is enough temporary memory space to invoke such messages. Similar to the locality aspect of CPU usage, memory usage analysis should also take locality into account in the context of distributed systems.

\[
RUD_{Memory} \cdot TCCFP \rightarrow TCCFP
\]

\[
\forall \rho \in TCCFP : RUD_{Memory}(\rho) = \rho - \{msg \mid msg \in \rho \land msg.methodOrSignalName \notin \{create,destroy\}\}
\]

\[
\forall msg \in Message : RUM_{Memory}(msg) = \begin{cases} +dataSize(msg.receiver.class) & \text{if } msg.methodOrSignalName = create \\ -dataSize(msg.sender.class) & \text{if } msg.methodOrSignalName = destroy \end{cases}
\]

Equation 4-RUD and RUM for memory resource.

### 6 Load Forecasting

We define Model-Based Load Forecasting (MBLF) as the process to predict the amount of load on different entities of a system using models. In our context, an entity can be an object, a node, or a network in a distributed RTS. Load on an entity with respect to a resource type can be informally phrased as the total usage amount of that resource deployed on that particular entity given a schedule of system scenarios. We intend to estimate the total load entailed on an entity by triggering a set of SDs (our chosen notion for scenarios). The motivations for predicting load in a RTS at the design stage are:

1. Analyzing the load of each entity to check whether it conforms to specifications at a stage of development where design decisions can still be easily changed. For example, our MBLF technique will help predict load on a specific node and determine if it exceeds its specified maximum value.
2. Finding the entities with highest loads and applying Software Performance Engineering [28] practices to balance load, if needed.

3. Stress testing: Using the load values at each time instant, stress testing techniques can be devised to schedule the execution of SDs such that the peak load values of all SDs all occur at the same time [23].

In our context, MBLF is closely related to RUA because RUA results are used to perform MBLF (recall Figure 6). We discussed in Section 5 how to measure resource usage of each DTCCFP. MBLF, on the other hand, considers a set of DTCCFPs and determines how much total load is imposed on an entity by triggering the DTCCFPs. MBLF is performed on a set of SD’s DTCCFPs that have been scheduled, i.e., we know at what time each DTCCFP is triggered. MBLF can be performed on different resource types, and, similarly to the RUA in Section 5, we select network traffic in this section as an example resource.

The concept of load forecasting query (LFQ), to filter the MBLF results for an entity, is described in Section 6.1. We describe the load forecasting heuristics in Section 6.2. We then define a class of load forecasting functions in Section 6.3 which are classified based on four load attributes (Section 6.1). The functions are similar to the traffic usage analysis functions (Section 5.5), except that the parameters of a typical load forecasting function are a schedule of a set of DTCCFPs, an entity, and a time instance (or interval), instead of a DTCCFP, an entity, and a time instance (or interval).

6.1 Load Forecasting Query

The following four load attributes determine the specifics of the MBLF to perform:

- **Load location**: nodes, objects or networks
- **Load direction** (for nodes only): into, from, or bidirectional
- **Load type**: data traffic or number of messages
- **Load duration**: instant or interval

For example, a LFQ can be the following: *what is the total (meaning by all SDs) number of requests towards object o at time instant t?* Note that there is a difference between a LFQ and a RUQ (Section 5.4): the former is a query for the amount of load entailed by triggering a set of TCCFPs, while the latter is a query for the amount of resource usage entailed by triggering a TCCFP. Such a difference originates from the difference between MBLF and RUA. RUA performs an analysis for a particular DTCCFP, while MBLF performs an analysis for a set of DTCCFPs.

6.2 Load Forecasting Heuristics

To forecast load in terms of time, our MBLF technique requires a Sequence Diagrams Schedule (SDS), which is a set of specific TCCFPs from each SD and their start time. A formal definition of a SDS is as follows. Assuming that a SUA has $n$ TCCFPs ($\rho_1, \ldots, \rho_n$), a SDS is a schedule of a selected set of TCCFPs in the form of: $\langle (\rho_1, \alpha_1), (\rho_1, \alpha_2), \ldots, (\rho_m, \alpha_m) \rangle$, where the value of $m$ is independent of the value of $n$ and each entry of the sequence is a tuple $(\rho, \alpha)$ such that $\alpha$ is the start time of TCCFP $\rho$, i.e., the time to trigger $\rho$. Note that zero, one or more different schedules of a TCCFP can appear in a SDS. For example, in a SUA which has three TCCFPs ($\rho_1$, $\rho_2$, $\rho_3$), $sds=\langle (\rho_1, 2ms), (\rho_1, 5ms), (\rho_3, 1ms) \rangle$ includes two schedule of $\rho_1$ (at 2ms and 5ms), one schedule of $\rho_3$ (at 1ms) and no schedule of $\rho_2$.

The heuristic of our load forecasting technique is illustrated using the example in Figure 15, which shows how the example LFQ in Section 6.1 can be answered. In order to forecast load, the RUA in Section 5 is performed first (Figure 15-(a)). It is assumed that the SUA has three SDs, and one DTCCFP has been selected for each SD: $\rho_1$, $\rho_2$ and $\rho_3$. Using the start times from the SDS, MBLF schedules RUA results of DTCCFPs (Figure 15-(b)) and then calculates the amount of load at a given time instance by adding up the usage values in all selected DTCCFPs (shown by a dashed line in Figure 15-(c)).
6.3 Load Forecasting Functions

The naming convention of the functions is given in Section 6.3.1. For brevity, only formal definitions of load functions for network load location are presented in Section 6.3.2. The functions for node and object load locations are derived in a similar fashion and are presented in Appendix B.

6.3.1 Naming Convention

A tree structure denoting the convention for naming load forecasting functions is shown in Figure 16. The root node of the tree has a null label. A function name is formed by traversing the tree from the root to a leaf node and concatenating all the node labels in order.

The bottom layer in Figure 16 specifies the input parameters of each load forecasting function. For example, consider the path specified by the bold line in Figure 16. This path represents function NetInsDL. According to the bottom layer, the input parameters of this function is \((sds, net, t)\). This function returns the instant (Ins) data load value (DL) of a given SDS \((sds)\) for a given network \((net)\) at a given time \((t)\). Input parameters with \(int\) in the bottom layer of Figure 16 correspond to the functions with interval duration. The start and end times of an interval, i.e., \(int=(start, end)\), should be provided for such functions. For functions with node or object traffic location, the input parameters include either \(nod\) or \(obj\) as traffic location, respectively.

![Figure 16-Naming convention and input parameters of load forecasting functions.](image-url)
6.3.2 Load Location: Network

The formulas to calculate the load forecasting functions with network load location are presented next. Functions for node and object load locations are presented in Appendix B.

1. \( \text{NetInsDL}(sds, net, t) \) returns the instant data load for SDS \( sds \) on network \( net \) at time \( t \) (time \( t=0 \) is the beginning of the SDS).

\[
\text{NetInsDL}(sds, net, t) = \sum_{p, ap \in sds} \text{NetInsDT}(p, net, t - ap)
\]

2. \( \text{NetInsML}(sds, net, t) \) returns the instant message load for SDS \( sds \) on network \( net \) at time \( t \).

\[
\text{NetInsML}(sds, net, t) = \sum_{p, ap \in sds} \text{NetInsMT}(p, net, t - ap)
\]

3. \( \text{NetIntDL}(sds, net, int) \) returns the interval data load for SDS \( sds \) on network \( net \) over time interval \( int \). \( \text{NetIntDL} \) can be calculated using \( \text{NetInsDL} \).

\[
\text{NetIntDL}(sds, net, int) = \sum_{t_{\text{start}}, t_{\text{end}} \in \text{int}} \text{NetInsDL}(sds, net, t)
\]

4. \( \text{NetIntML}(sds, net, int) \) returns the interval message load for SDS \( sds \) on network \( net \) over time interval \( int \). \( \text{NetIntML} \) can be calculated using \( \text{NetInsML} \).

\[
\text{NetIntML}(sds, net, int) = \sum_{t_{\text{start}}, t_{\text{end}} \in \text{int}} \text{NetInsML}(sds, net, t)
\]

7 Reusing MBRUA Principles for Additional Applications

We describe in this section how the prediction of resource usage and load enables developers to perform two additional, important activities:

- Detecting resource overuse (Section 7.1)
- Detecting illegal access to mutually exclusive objects (Section 7.2)

7.1 Detecting Resource Overuse

Given a resource \( R \) with capacity value \( C_R \), we say that \( R \) is overused if the total usage of \( R \), referred to as \( U_R \), by processes of a system exceeds \( C_R \) (\( U_R > C_R \)). Note that resource usage and capacity are measured using the same unit but the unit depends on the resource type. For example, capacity and usage of a network are both usually measured in Mbps (mega bits per second). The capacity of a CPU is usually fixed at 100% and CPU usage is measured relatively to this capacity (e.g., 80%).

Considering network traffic as resource type and using the formalisms in Sections 5 and 6, the formula in Equation 5 can be used to detect network traffic overuse based on load analysis information. The function \( \text{DetectNetworkTrafficOveruse}(sds, net, C_{net}) \) returns true if the amount of traffic on network \( net \) entailed by a set of TCCFPs with a specific schedule \( sds \) is superior to the network's capacity value \( (C_{net}) \) in at least one time instance.

\[
\text{DetectNetworkTrafficOveruse}(sds, net, C_{net}) = \begin{cases} 
\text{true} & \text{if } \exists t \mid \text{NetInsDL}(sds, net, t) > C_{net} \\
\text{false} & \text{otherwise} 
\end{cases}
\]

Equation 5- Detecting network traffic overuse based on load analysis information.

Note that we have used the \( \text{data} \) load forecasting function \( \text{NetInsDL} \) in Equation 5, since network capacities are usually measured in terms of maximum amount of data which can be transmitted over a network. However, we acknowledge that detection of network traffic overuse can also potentially be performed based on message traffic. Message capacity of a network depends on many factors (e.g., buffer sizes) and measuring a network's message capacity requires additional analysis using communication network theories (e.g., queueing theory [29]).
7.2 Detecting Illegal Concurrent Access to Critical Objects

In RTSs there are often objects that should be accessed in a mutually exclusive manner. In other words, such objects (referred to as critical) cannot be accessed by more than one client at a time. For example, a robot arm (actuator) can be sent only one move signal at a time. Sending two move signals (in different directions) can lead to physical damage to the arm. This indicates a risk of violating mutual exclusion. But if the robot arm operations are synchronized, mutual exclusion will be handled correctly. Our load forecasting technique can be used to detect such illegal accesses to mutually exclusive objects based on SDs. This can be useful when validating SD schedules, generated using schedulability analysis, in terms of access to mutually exclusive objects.

For example, consider a robot arm class RobotArm which appears in behavioral models (SDs) of a RTS. Different control objects of such a system can often send messages (control messages) to an object of type RobotArm. Depending on the business logic of the RTS under study, a subset of those messages can be mutually exclusive messages, and cannot be sent at the same time. (An alternative design decision to enforce mutual exclusion among such messages could be to incorporate a queuing mechanism for messages sent concurrently.) For example, in the above example, [moveLeft, moveRight, moveUp, moveDown] is a set of mutually exclusive messages and denotes that a robot arm cannot be asked to, for example, move up and down at the same time. Mutually exclusive messages can be defined by modelers in a class diagram by using UML concurrency=sequential\(^1\) tagged-values on the operations of a class.

Our heuristic to check for illegal (mutually exclusive) access to a critical object is to count the number of control messages to this object at each time instance during a given SDS and make sure that there exists no time instance in which there are two or more control messages being sent. Such a heuristic is formalized in Equation 6, where TotalInControlLoad(sds, obj, t) returns the number of control messages which are sent towards object obj at time t by triggering SDS sds.

\[
\text{DetectIllegalMutualExclusiveAccess(sds, obj)} = \begin{cases} 
\text{true} & \text{if } \exists t \mid \text{TotalInControlLoad(sds, obj, t)} > 1 \\
\text{false} & \text{otherwise}
\end{cases}
\]

Equation 6-Detecting illegal concurrent accesses to a critical object obj.

The formal definition of TotalInControlLoad(sds, obj, t) is provided in Equation 7, where the number of all messages in all TCCFPs of the given SDS at a time instance are counted. MutuallyExclusiveMessages is the set of control messages which can not be sent concurrently to object obj.

\[
\text{TotalInControlLoad(sds, obj, t)} = \sum_{\forall \rho, ap \in \text{sds}} \text{msg} \mid \text{where msg} = \begin{cases} 
\text{msg}_{i} \in \rho \land \\
\text{msg}_{i}.\text{start} \leq t - \alpha_{\rho} \leq \text{msg}_{i}.\text{end} \land \text{msg}_{i}.\text{receiver.object} = \text{obj} \land \\
\text{msg}_{i}.\text{methodOrSignalName} \in \text{MutuallyExclusiveMessages}
\end{cases}
\]

Equation 7-Calculating the total number of control messages towards an object based on a SDS.

8 Case Study

We apply our MBRUA approach to a distributed system to demonstrate its feasibility and to illustrate the variety of relevant MBRUA activities that can be performed on this system. Our case study system and its UML model are described in Section 8.1. We then report two applications of our MBRUA approach here: (1) Predicting resource usage pattern (Section 8.2), and (2) Detecting resource overuse (Section 8.3).

\(^1\) If the concurrency of an operation is set to sequential, it indicates that “callers (of the operation) must coordinate so that only one call to an object (on any sequential operation) may execute at once. If concurrent calls occur, then the semantics and integrity of the system cannot be guaranteed” \([30]\) J. Rumbaugh, I. Jacobson, and G. Booch, Unified Modeling Language Reference Manual, 2nd ed: Addison-Wesley Professional, 2004.
8.1 The Case Study System and its UML Model

Our case study system is a prototype SCADA-based power system (Supervisory Control and Data Acquisition Systems [31]). The system is referred to as SCAPS (a SCADA-based Power System) [23]. SCAPS is a system to control the power distribution grid across a nation consisting of several provinces. Each province has several cities and regions. There is one central server in each province which gathers the SCADA data from Tele-Control units (TCs) from all over the province and sends them to the central national server. The national server performs the following real-time, data-intensive safety-critical functions as part of the Power Application Software: (1) Overload monitoring and control, (2) Detection of separated (disconnected) power system, and (3) Power restoration after grid failure. The UML design model of SCAPS is presented in [21]. The two analyses in Sections 8.2 and 8.3 use different parts of the SCAPS UML model. Thus, we present next the parts of the model used in this article. The complete UML model can be found in [21].

A subset of the SCAPS UML diagrams required by the analysis in this article is shown in Figure 17. The SD in Figure 17-(a) corresponds to the overload monitoring (OM) control of the province of Ontario (ON) in SCAPS. By using the interaction occurrence construct of UML 2.0 SDs, the SD OM_ON references the SD in Figure 17-(c) to retrieve data from the provincial Tele-Control units. An object of type ASA (Automatic System Agent), deployed in one of the national servers (SEV_CA1), is the active actor in those SDs.

Figure 17-(d) is the Network Deployment Diagram (NDD), as defined in Section 5.1.2, of the system. We do not show the SCAPS class diagram here due to its large size. Parameter dataType in Figure 17, used in call messages queryONData, is an instance of class LoadStatus. An instance of the LoadStatus class stores information about the load levels of different parts of the grid served by a SCAPS Tele-Control unit. The data size of this class, as required in our predictability analysis and calculated using Equation 2 based on the attributes of the LoadStatus class in the SCAPS class diagram, is 4 MB. Note that such an estimate is a realistic value, according to the SCADA literature (e.g., [31]). Furthermore as we discuss in detail in [21], the real time constraints in Figure 17 are realistic estimates of message duration times used in SCADA power systems.

Figure 17-Parts of the SCAPS design UML model.
8.2 Predicting Resource Usage Pattern

To demonstrate the capability of our MBRUA approach to predict resource usage, we apply the technique to SCAPS using the partial design model in Figure 17 and considering network bandwidth as an example resource type. The predicted network bandwidth of executing SD OM_ON is compared to the real observed resource usage values derived by running this SD and measuring the amount of network traffic using the Network Traffic Monitor (NTM) tool [32] in each time instant. Different steps of the prediction process are discussed next.

8.2.1 Control Flow Analysis of SCAPS SDs

The first step of the MBRUA is to analyze the control flow in SDs. The CCFG of the SD OM_ON is shown in Figure 18. To ease references to CCFG nodes in our coming discussion, CCFG nodes have been annotated with $A_x$ labels, where $x$ is a sequential number. The next step is to derive TCCFPs from the CCFG. Since there are no decision nodes in the CCFG, there is only one TCCFP called $\rho$ in Figure 19. Using RUD (Section 5.2), the RUA process converts $\rho$ to a DTCCFP, as shown in Figure 19. Only local message $A_{13}$ is removed in this process, as the other messages are all distributed.

8.2.2 Traffic Prediction and Measurement

We show next how to predict the interval data traffic over the SCAPS national network (Canada) during the execution of SD OM_ON. The RUA function $NetInsDT(\rho_{OM\_ON}, \text{"Canada"}, t)$ is computed for different values of $t$ in the time interval [0 ms, 13 ms (duration of $\rho_{OM\_ON}$)], as depicted in Figure 22. Referring to the messages in the SD OM_ON and the SCAPS deployment structure (Figure 17-(a)), only the reply message from an object of type ProvController (in SD OM_ON) to an object of type ASA (Automatic System Agent), i.e., message $A_{12}$ in Figure 18, is sent over the national network (Canada) and is considered in the RUA. The data size of this return message is 20 MB: 5 (number of TCs in Ontario) $\times$ 4 MB (data size of the LoadStatus class). Since the duration of this message is 7 seconds (12-5 as specified in Figure 17-(a)), the estimated traffic per time unit is $20\ MB/7 \approx 2.85\ MB/s$.

We discuss now how we compared the above predicted network bandwidth with the measured network bandwidth entailed by executing CCFP $\rho_{OM\_ON}$. The runtime resource usage values were measured by executing CCFP $\rho_{OM\_ON}$ and recording the amount of network traffic using the Network Traffic Monitor (NTM) tool [32] in each time instant. Note that we have the same time precision (1 second) in the NTM tool, the time annotations in the SDs, and our formulas. We used the heuristics [33] of summing up the traffic on all the nodes in a particular network to measure the total traffic on that network. We tried to make our measurements as accurate as possible by turning off all network services on the machines involved except the SCAPS application. In this way, all the measured traffic corresponded only to the traffic exchanged by the SCAPS applications on different nodes.

Four PCs were used to play the roles of SEV_CAI (one PC), SEV_ON (one PC), TC_YOWx (one PC) and TC_YYZx (one PC) nodes. The last two deployment decisions (related to TCs) were made to simplify the
system’s deployment, controllability (less nodes to control at runtime) as well as observability. The actual network deployment of our case study is illustrated in Figure 20: Three intranets in our institution (servers, SCE department, and squall.sce.carleton.ca) were chosen to simulate the network architecture of SCAPS. This network deployment is not exactly the one specified for SCAPS but this decision, motivated by practical considerations, had however a negligible effect (if any) on our case study. Indeed, as discussed above, our goal in this section of the case study is to show how the predicted values (calculated using the RUA functions defined in Section 5.5) of the interval data traffic over the SCAPS national network (Canada) during the execution of SD OM_ON compare to the measured values. As illustrated in Figure 17-(d) and Figure 20, the traffic over the national network (Canada) should not change when several TCs of the system are merged into one physical node. Note these TCs only return data (i.e., to be transmitted over the network) and having one merged physical node or several nodes should not differ from a network traffic point of view.

![Figure 20-The actual network deployment of our case study.](image)

The Operating System (OS) and hardware configurations of the machines used are detailed in Table 1. All the network connections had a bandwidth of 100 Mbps.

<table>
<thead>
<tr>
<th>Machine hosting</th>
<th>OS</th>
<th>CPU</th>
<th>RAM</th>
<th>Network Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEV_CA1</td>
<td>Windows 2000</td>
<td>2.8 GHz Intel Xeon</td>
<td>2 GB</td>
<td>Intel PRO/1000 XT</td>
</tr>
<tr>
<td>SEV_ON</td>
<td>Windows XP</td>
<td>2.8 GHz Intel Pentium 4</td>
<td>2 GB</td>
<td>3COM Fast Ethernet Controller</td>
</tr>
<tr>
<td>TC_YOWx</td>
<td>Windows 2000</td>
<td>863 MHz Intel Pentium III</td>
<td>1 GB</td>
<td>3COM Fast Ethernet Controller</td>
</tr>
<tr>
<td>TC_YYZx</td>
<td>Windows 2000</td>
<td>863 MHz Intel Pentium III</td>
<td>1 GB</td>
<td>3COM Fast Ethernet Controller</td>
</tr>
</tbody>
</table>

Table 1- OS and hardware configurations of the machines used in our case study.

The NTM [32] was installed on each machine and was “turned on” to monitor and log the measured traffic per second. Note that we conducted a careful analysis of the NTM output results in order to calculate only the traffic values on the Canada network, e.g., network traffic measurements between TC_YOWx and SEV_ON nodes (Figure 20) were not used in the analysis. Screenshots are provided in Figure 21 for the main screen of the NTM (a), while only SCAPS is running as a network application, the main screen of NTM while other network application are running (b), a traffic log view in NTM (c), and SCAPS (d). As explained above, we stopped all network services on the involved machines except the SCAPS application. This can be seen by comparing Figure 21-(a) and (b). Thus all network traffic values (either incoming or outgoing traffic) reported by NTM in case of Figure 21-(a) corresponds to that entailed by SCAPS. Figure 21-(c) shows a traffic log view in NTM where the grid shows the measured traffic value (triggered by SCAPS) in each second. Note that although NTM is powerful enough to let us extract the traffic values resulting from a network application among several running applications on a machine (controllable by the process section in Figure 21-(c)), we nevertheless decided to turn off other network applications (and services) on the machines to simulate a dedicated network as it is the case in real DRTSs, and to minimize any network-related effect by other applications on our case study. Figure 21-(d) shows a snapshot in which the SCAPS application is running on the SEV_CA1 node and has just accepted a connection from TC_YOW1.
Figure 21-Screenshots from (a): the main screen of Network Traffic Monitor (NTM) [32] while only SCAPS is running as a network application, (b) the main screen of NTM while other network applications are running, (c) a traffic log view in NTM, and (d): SCAPS.

8.2.3 Comparison Results

The average values of observed data traffic in each time instant over 10 runs are depicted in Figure 22. The overall average across all time instants is 3.02 MB/s and is slightly larger than the predicted value (2.85 MB/s; calculated using the RUA functions defined in Section 5.5). We believe that this is most probably due to the fact that extra data is added by the lower layers of the OSI (Open Systems Interconnection) network model—such as data link and physical—to the data submitted by the application layer of the OSI model. Estimating such a difference requires a detailed analysis of packet and frame structures in different layers of
the OSI model as discussed in Section 5.3. As we can see in Figure 22, the difference is small in our case study, providing evidence that our estimates of data traffic are reasonably accurate. The close correspondence between the predicted and observed values in the SCAPS case study suggests that the proposed MBRUA approach is a promising way of predicting network traffic in the early design stages.

![NetInsDT(pOM_ON,"Canada",t) (MB)](image)

Figure 22-An example of predicted and observed resource usage analysis.

### 8.3 Detecting Resource Overuse

To demonstrate the capability of our MBRUA approach to detect resource overuse, we apply it to SCAPS by attempting to determine if any network traffic overuse occurs when triggering a SDS of two SDs OM_ON, Figure 17-(a), and DSPS_ON (Detection of Separated Power System), Figure 17-(b).

Since both SDs OM_ON and DSPS_ON are initiated by ASA (SCAPS Automatic System Agent) and both query specific provincial power grid data (load and connectivity, respectively), the entailed traffic goes through the national network (Canada) (Figure 17-(b)). As a given SDS, we consider \( sds = (\rho_{OM\_ON},0ms), (\rho_{DSPS\_ON},0ms) >, \) meaning that the only CCFPs of both SDs start at the same time. Our goal is to determine if any network overuse situation occurs in the SCAPS national network (Canada) during the execution of this SDS. According to Section 7.1, we have to evaluate function \( \text{DetectNetworkTrafficOveruse}(sds, "Canada", C\_Canada) \), where \( C\_Canada = 100 \text{ Mbps} \) (mega bits per second)=12.5 MBps (mega bytes per second).

As shown in Equation 5, in order to do so, \( NetInsDL(sds, "Canada", t) \) should be analyzed first. This function is the summation of \( NetInsDT \) function values across all CCFPs in the SDS. An example of calculations for a \( NetInsDT \) function for different time values based on a given CCFP was presented in Section 8.2, which corresponded to the only CCFP \( (\rho_{OM\_ON}) \) of SD OM_ON. By using a similar procedure, we calculated the \( NetInsDT(\rho_{DSPS\_ON} "Canada", t) \) values as shown in Figure 23-(b). In order to calculate the predicted resource usage values of \( \rho_{DSPS\_ON} \), in Figure 23-(b), we assumed that the size of the Ontario grid connectivity data \( (connectivityON \) in Figure 17-(b) is 50 MB (a realistic estimation based on the SCADA literature [31]). Therefore, the resource usage value per time unit will be \( 50/3 = 16.6 \text{ MB} \). 3 \( ms \) here is the duration of the return message from the \( queryONData \) interaction occurrence in Figure 17-(b).

Figure 23-(a) corresponds to \( NetInsDT((\rho_{OM\_ON} "Canada", t) \) and uses the predicted values from Figure 22 (i.e., 2.85MB/s). Using the definitions in Section 6 and based on the definition of \( sds \) (above), the two \( NetInsDT \) functions in Figure 23-(a) and (b) yield the \( NetInsDL \) function values shown in Figure 23-(c), where the values at each time instance from each of the two \( NetInsDT \) functions have been added to yield \( NetInsDL \) values.

Recall that our objective in this section was to determine if any network traffic overuse occurs in the SCAPS national network (Canada) during the execution of \( sds \). By substituting the variables in Equation 5 with the values in this section (network name and its capacity), we can write:

\[
\text{DetectNetworkTrafficOveruse}(sds, "Canada",12MB) = \begin{cases} 
\text{true} & \text{if } \exists t \mid NetInsDL(sds,"Canada",t) > 12.5 \text{MB} \\
\text{false} & \text{otherwise}
\end{cases}
\]

\[
= \text{true} \quad \text{because for } t \in [8,9,10]: NetInsDL(sds, "Canada", t) > 12.5 \text{ MB}
\]

The above function returns true (meaning that a network traffic overuse is detected) since load values in three time instances (8, 9 and 10 s) exceed \( C\_Canada = 12.5 \text{ MB} \) (the capacity of network Canada). Such an analysis is
shown graphically in Figure 23-(c), where the 12.5 MB capacity is depicted by a horizontal bold line. The resource overuse region is marked by a dashed rectangle.

To investigate if the network traffic overuse detected by our technique really occurs for a specific execution of SCAPS, we executed SDs OM_ON and DSPS_ON according to the schedule specified in sds. The entailed traffic on network Canada by this execution was recorded using a strategy similar to what we reported in Section 8.2. A comparison between the predicted values (from Figure 23-(c)) and the observed network traffic load is reported in Figure 23-(d). Note the discrepancy (during time interval [8, 12]) between the predicted and the observed values of traffic usage when a resource overuse occurs. Since the network capacity is 100 Mbps, the network can not transmit more than 12.5 MB of traffic per second. Therefore, data buffering techniques are employed in the network (by routers and network interfaces) to prevent data loss. This leads to the discrepancy situation in time instances 11 and 12 as reported in Figure 23-(d), in which neither of the return messages in SD OM_ON nor DSPS_ON complete in their specified time (i.e., time=12 and 11 seconds, respectively). In other words, we have found a network traffic overuse situation which has led to real-time constraint violations in SCAPS.

When detecting a network traffic overuse, the developers must take necessary actions to fix the problem. Typical suggestions are: (1) increase the capacity of the network under investigation, (2) assess whether an increase in duration of relevant SD messages is acceptable, and (3) decrease the amount of data exchanged. Such corrective actions are however easier to undertake in early design stages, before the implementation is completed.

9 CONCLUSIONS

This paper presents a quantitative framework for the early prediction of resource usage and load in Real-Time Systems (RTS) during the design phase. The prediction is based on an analysis of UML 2.0 sequence diagrams, augmented with timing information, to extract timed-control flow information. It is aimed at improving the predictability of a RTS by offering a systematic approach, based on plausible and standard early model representations, to predict system behavior in each time instant during its execution. The results of a case study on an actual RTS have shown that the approach is promising as it yields reasonably accurate results (estimates are on average 6% below actual values).
We furthermore reuse our Model-based Resource Usage Analysis (MBRUA) principles and develop automated techniques to perform two important activities in the context of RTSs: (1) Detecting resource overuse, and (2) Detecting illegal access to mutually exclusive objects. The former activity is applied to our case study system and we show how it helps us to detect a network traffic overuse in the system under analysis before its deployment.

Some of our future works include: (1) applying the approach on more complex RTSs and assess its effectiveness in improving the predictability of RTSs; (2) using the load forecasting information to develop model-based load balancing techniques; (3) investigating further the resource usage analysis of other resource types; (4) developing a complete RUA activity for CPU and memory resource types (similar to the one presented for network traffic in this article) which includes a set of time-based RUA functions based on the presented RUDs and RUMs; and (5) developing more accurate network usage measures which will account for the extra data added by the lower layers of the OSI (Open Systems Interconnection) network model—such as data link and physical—to the data submitted by the application layer of the OSI model.

ACKNOWLEDGEMENTS

This work was in part supported by Siemens Corporate Research, Princeton, NJ and a Canada research chair.

REFERENCES


APPENDIX A. RESOURCE USAGE ANALYSIS FUNCTIONS FOR NODE AND OBJECT USAGE LOCATIONS

Traffic Location: Node

1. *NodInInsDT(ρ, nod, t)* returns the instant data traffic that node *nod* receives for DTCCFP *ρ* at time *t*. “In” denotes that the traffic direction is towards the node as explained in Section 5.4.2.

   \[
   NodInInsDT(\rho, \text{nod}, t) = \begin{cases} 
   \sum_{\text{msg} \in \text{msg}} \frac{\text{RUM}_{\text{loc}}(\text{msg})}{\text{dur}(\text{msg})} \cdot |\text{msg}| > 0, \text{where} \\
   0 \quad \text{otherwise}
   \end{cases}
   \]

   \[
   \text{where} 0 < \text{msg} = \begin{cases} 
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   0 \quad \text{otherwise}
   \end{cases}
   \]

2. *NodInInsMT(ρ, nod, t)* returns the instant message traffic that node *nod* receives for DTCCFP *ρ* at time *t*.

   \[
   NodInInsMT(\rho, \text{nod}, t) = \text{msg} \text{ where msg} = \begin{cases} 
   \text{msg} \text{, msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   0 \quad \text{otherwise}
   \end{cases}
   \]

3. *NodInIntDT(ρ, nod, int)* returns the data traffic that node *nod* receives for DTCCFP *ρ* over a time interval *int*.

   \[
   NodInIntDT(\rho, \text{nod}, \text{int}) = \sum_{\text{t}=\text{start.t}, \text{int}, \text{start.t}+1, \ldots, \text{end.t}} \text{NodInInsDT(\rho, nod, t)}
   \]

4. *NodInIntMT(ρ, nod, int)* returns the message traffic that node *nod* receives for DTCCFP *ρ* over time interval *int*.

   \[
   NodInIntMT(\rho, \text{nod}, \text{int}) = \sum_{\text{t}=\text{start.t}, \text{int}, \text{start.t}+1, \ldots, \text{end.t}} \text{NodInInsMT(\rho, nod, t)}
   \]

5. *NodOutInsDT(ρ, nod, t)* returns the instant data traffic that node *nod* sends for DTCCFP *ρ* at time *t*. “Out” denotes that the traffic direction is from the node as explained in Section 5.4.2.

   \[
   NodOutInsDT(\rho, \text{nod}, t) = \begin{cases} 
   \sum_{\text{msg} \in \text{msg}} \frac{\text{RUM}_{\text{loc}}(\text{msg})}{\text{dur}(\text{msg})} \cdot |\text{msg}| > 0, \text{where} \\
   0 \quad \text{otherwise}
   \end{cases}
   \]

6. *NodOutInsMT(ρ, nod, t)* returns the instant message traffic that node *nod* sends for DTCCFP *ρ* at time *t*.

   \[
   NodOutInsMT(\rho, \text{nod}, t) = \text{msg} \text{ where msg} = \begin{cases} 
   \text{msg} \text{, msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   \text{msg}, \text{msg} \in \rho \wedge \\
   \text{msg}, \text{msg} \in \text{nod} \\
   0 \quad \text{otherwise}
   \end{cases}
   \]

7. *NodOutIntDT(ρ, nod, int)* returns the data traffic that node *nod* sends for DTCCFP *ρ* over time interval *int*.

   \[
   NodOutIntDT(\rho, \text{nod}, \text{int}) = \sum_{\text{t}=\text{start.t}, \text{int}, \text{start.t}+1, \ldots, \text{end.t}} \text{NodOutInsDT(\rho, nod, t)}
   \]

8. *NodOutIntMT(ρ, nod, int)* returns the message traffic that node *nod* receives for DTCCFP *ρ* over time interval *int*.

   \[
   NodOutIntMT(\rho, \text{nod}, \text{int}) = \sum_{\text{t}=\text{start.t}, \text{int}, \text{start.t}+1, \ldots, \text{end.t}} \text{NodOutInsMT(\rho, nod, t)}
   \]
9. $NodBiInsDT(\rho, nod, t)$ returns the instant data traffic that node $nod$ sends or receives for DTCCFP $\rho$ at time $t$.

$$
NodBiInsDT(\rho, nod, t) = \begin{cases}
\sum_{msg, \rho} RUM_{\rho}(msg, \rho) / \text{dur}(msg) & |msg| > 0,\text{where} \\
msg = \begin{cases}
msg & msg_1 \in \rho \\
msg_1, start \leq t \leq msg_1, end \\
(msg, \text{sender.node} = nod \lor msg, \text{receiver.node} = nod)
\end{cases} \\
0 & \text{otherwise}
\end{cases}
$$

10. $NodBiInsMT(\rho, nod, t)$ returns the instant message traffic that node $nod$ sends or receives for DTCCFP $\rho$ at time $t$.

$$
NodBiInsMT(\rho, nod, t) = \{msg\} \text{ where } msg = \begin{cases}
msg_1 & msg_1 \in \rho \\
msg_1, start \leq t \leq msg_1, end \\
(msg, \text{sender.node} = nod \lor msg, \text{receiver.node} = nod)
\end{cases}
$$

11. $NodBiIntDT(\rho, nod, int)$ returns the data traffic that node $nod$ sends or receives for DTCCFP $\rho$ over time interval $int$.

$$
NodBiIntDT(\rho, nod, int) = \sum_{t=start, int, start+1, \ldots, int} NodBiInsDT(\rho, nod, t)
$$

12. $NodBiIntMT(\rho, nod, int)$ returns the message traffic that node $nod$ sends or receives for DTCCFP $\rho$ over time interval $int$.

$$
NodBiIntMT(\rho, nod, int) = \sum_{t=start, int, start+1, \ldots, int} NodBiInsMT(\rho, nod, t)
$$

Traffic Location: Object

1. $ObjInInsDT(\rho, obj, t)$ returns the instant data traffic object $obj$ receives for DCCFP $\rho$ at time $t$.

$$
ObjInInsDT(\rho, obj, t) = \begin{cases}
\sum_{msg, \rho} RUM_{\rho}(msg, \rho) / \text{dur}(msg) & |msg| > 0,\text{where} \\
msg = \begin{cases}
msg_1 & msg_1 \in \rho \\
msg_1, start \leq t \leq msg_1, end \\
msg, \text{receiver.object} = obj
\end{cases} \\
0 & \text{otherwise}
\end{cases}
$$

2. $ObjInInsMT(\rho, obj, t)$ returns the instant message traffic object $obj$ receives for DCCFP $\rho$ at time $t$.

$$
ObjInInsMT(\rho, obj, t) = \{msg\} \text{ where } msg = \begin{cases}
msg & msg_1 \in \rho \\
msg_1, start \leq t \leq msg_1, end \\
msg, \text{receiver.object} = obj
\end{cases}
$$

3. $ObjInIntDT(\rho, obj, int)$ returns the data traffic that object $obj$ receives for DTCCFP $\rho$ over a time interval $int$.

$$
ObjInIntDT(\rho, obj, int) = \sum_{t=start, int, start+1, \ldots, int} ObjInInsDT(\rho, obj, t)
$$

4. $ObjInIntMT(\rho, obj, int)$ returns the message traffic that object $obj$ receives for DTCCFP $\rho$ over time interval $int$.

$$
ObjInIntMT(\rho, obj, int) = \sum_{t=start, int, start+1, \ldots, int} ObjInInsMT(\rho, obj, t)
$$

5. $ObjOutInsDT(\rho, obj, t)$ returns the instant data traffic that object $obj$ sends for DTCCFP $\rho$ at time $t$. “Out” denotes that the traffic direction is from the object as explained in Section 5.4.2.
6. $\text{ObjOutInsDT}(\rho, \text{nod}, t) = \sum_{\text{msg} \in \text{msg}} \frac{\text{RUM}_{\text{obj}}(\text{msg}, i) / \text{dur}(\text{msg}, i)}{\text{msg}} ; |\text{msg}| > 0$, where
$$\text{msg} = \begin{cases} \text{msg} | \text{msg} \in \rho \land & \text{msg}_\text{start} \leq t \leq \text{msg}_\text{end} \land \\
\text{msg}, \text{sender.object} = \text{obj} \
\end{cases}$$

7. $\text{ObjOutIntDT}(\rho, \text{obj}, \text{int})$ returns the data traffic that object $\text{obj}$ sends for DTCCFP $\rho$ over time interval $\text{int}$.

8. $\text{ObjOutIntMT}(\rho, \text{obj}, \text{int})$ returns the message traffic that object $\text{obj}$ receives for DTCCFP $\rho$ over time interval $\text{int}$.

9. $\text{ObjBiInsDT}(\rho, \text{obj}, t)$ returns the instant data traffic that object $\text{obj}$ sends or receives for DTCCFP $\rho$ at time $t$.

10. $\text{ObjBiInsMT}(\rho, \text{obj}, t)$ returns the instant message traffic that object $\text{obj}$ sends or receives for DTCCFP $\rho$ at time $t$.

11. $\text{ObjBiIntDT}(\rho, \text{obj}, \text{int})$ returns the data traffic that object $\text{obj}$ sends or receives for DTCCFP $\rho$ over time interval $\text{int}$.

12. $\text{ObjBiIntMT}(\rho, \text{obj}, \text{int})$ returns the message traffic that object $\text{obj}$ sends or receives for DTCCFP $\rho$ over time interval $\text{int}$.
APPENDIX B. LOAD FORECASTING FUNCTIONS FOR NODE AND OBJECT LOAD LOCATIONS

Load Location: Node

1. \( \text{NodInInsDL}(sds, \text{nod}, t) \) returns the instant data load for SDS \( sds \) that node \( \text{nod} \) receives at time \( t \).
   \[
   \text{NodInInsDL}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodInInsDT}(p, \text{nod}, t - ap)
   \]

2. \( \text{NodOutInsDL}(sds, \text{nod}, t) \) returns the instant data load for SDS \( sds \) that node \( \text{nod} \) sends at time \( t \).
   \[
   \text{NodOutInsDL}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodOutInsDT}(p, \text{nod}, t - ap)
   \]

3. \( \text{NodBiInsDL}(sds, \text{nod}, t) \) returns the instant data load for SDS \( sds \) that node \( \text{nod} \) sends or receives at time \( t \).
   \[
   \text{NodBiInsDL}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodBiInsDT}(p, \text{nod}, t - ap)
   \]

4. \( \text{NodInInsML}(sds, \text{nod}, t) \) returns the instant message load for SDS \( sds \) that node \( \text{nod} \) receives at time \( t \).
   \[
   \text{NodInInsML}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodInInsMT}(p, \text{nod}, t - ap)
   \]

5. \( \text{NodOutInsML}(sds, \text{nod}, t) \) returns the instant message load for SDS \( sds \) that node \( \text{nod} \) sends at time \( t \).
   \[
   \text{NodOutInsML}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodOutInsMT}(p, \text{nod}, t - ap)
   \]

6. \( \text{NodBiInsML}(sds, \text{nod}, t) \) returns the instant message load for SDS \( sds \) that node \( \text{nod} \) sends or receives at time \( t \).
   \[
   \text{NodBiInsML}(sds, \text{nod}, t) = \sum_{\forall p, ap \in sds} \text{NodBiInsMT}(p, \text{nod}, t - ap)
   \]

7. \( \text{NodInIntDL}(sds, \text{nod}, \text{int}) \) returns the interval data load for SDS \( sds \) that node \( \text{nod} \) receives over time interval \( \text{int} \). \( \text{NodInIntDL} \) can be calculated using \( \text{NodInInsDL} \).
   \[
   \text{NodInIntDL}(sds, \text{nod}, \text{int}) = \sum_{\text{start} \to \text{end} \in \text{int}} \text{NodInInsDL}(sds, \text{nod}, t)
   \]

8. \( \text{NodOutIntDL}(sds, \text{nod}, \text{int}) \) returns the interval data load for SDS \( sds \) that node \( \text{nod} \) sends over time interval \( \text{int} \). \( \text{NodOutIntDL} \) can be calculated using \( \text{NodOutInsDL} \).
   \[
   \text{NodOutIntDL}(sds, \text{nod}, \text{int}) = \sum_{\text{start} \to \text{end} \in \text{int}} \text{NodOutInsDL}(sds, \text{nod}, t)
   \]

9. \( \text{NodBiIntDL}(sds, \text{nod}, \text{int}) \) returns the interval data load for SDS \( sds \) that node \( \text{nod} \) sends or receives over time interval \( \text{int} \). \( \text{NodBiIntDL} \) can be calculated using \( \text{NodBiInsDL} \).
   \[
   \text{NodBiIntDL}(sds, \text{nod}, \text{int}) = \sum_{\text{start} \to \text{end} \in \text{int}} \text{NodBiInsDL}(sds, \text{nod}, t)
   \]

10. \( \text{NodInIntML}(sds, \text{nod}, \text{int}) \) returns the interval message load for SDS \( sds \) that node \( \text{nod} \) receives over time interval \( \text{int} \). \( \text{NodInIntML} \) can be calculated using \( \text{NodInInsML} \).
    \[
    \text{NodInIntML}(sds, \text{nod}, \text{int}) = \sum_{\text{start} \to \text{end} \in \text{int}} \text{NodInInsML}(sds, \text{nod}, t)
    \]

11. \( \text{NodOutIntML}(sds, \text{nod}, \text{int}) \) returns the interval message load for SDS \( sds \) that node \( \text{nod} \) sends over time interval \( \text{int} \). \( \text{NodOutIntML} \) can be calculated using \( \text{NodOutInsML} \).
    \[
    \text{NodOutIntML}(sds, \text{nod}, \text{int}) = \sum_{\text{start} \to \text{end} \in \text{int}} \text{NodOutInsML}(sds, \text{nod}, t)
    \]
12. **NodBiIntML**(sds, nod, int) returns the interval message load for SDS sds that node nod sends or receives over time interval int. NodBiIntML can be calculated using NodBiInsML.

\[
NodBiIntML(sds, nod, int) = \sum_{\text{start, end} \in \text{int}} NodBiInsML(sds, nod, t)
\]

**Load Location: Object**

1. **ObjInInsDL**(sds, obj, t) returns the instant data load for SDS sds that object obj receives at time t.

\[
ObjInInsDL(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjInInsDT(p, obj, t - ap)
\]

2. **ObjOutInsDL**(sds, obj, t) returns the instant data load for SDS sds that object obj sends at time t.

\[
ObjOutInsDL(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjOutInsDT(p, obj, t - ap)
\]

3. **ObjBiInsDL**(sds, obj, t) returns the instant data load for SDS sds that object obj sends or receives at time t.

\[
ObjBiInsDL(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjBiInsDT(p, obj, t - ap)
\]

4. **ObjInInsML**(sds, obj, t) returns the instant message load for SDS sds that object obj receives at time t.

\[
ObjInInsML(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjInInsMT(p, obj, t - ap)
\]

5. **ObjOutInsML**(sds, obj, t) returns the instant message load for SDS sds that object obj sends at time t.

\[
ObjOutInsML(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjOutInsMT(p, obj, t - ap)
\]

6. **ObjBiInsML**(sds, obj, t) returns the instant message load for SDS sds that object obj sends or receives at time t.

\[
ObjBiInsML(sds, obj, t) = \sum_{p, ap \in \text{sds}} ObjBiInsMT(p, obj, t - ap)
\]

7. **ObjInIntDL**(sds, obj, int) returns the interval data load for SDS sds that object obj receives over time interval int. ObjInIntDL can be calculated using ObjInInsDL.

\[
ObjInIntDL(sds, obj, int) = \sum_{\text{start, end} \in \text{int}} ObjInInsDL(sds, obj, t)
\]

8. **ObjOutIntDL**(sds, obj, int) returns the interval data load for SDS sds that object obj sends over time interval int. ObjOutIntDL can be calculated using ObjOutInsDL.

\[
ObjOutIntDL(sds, obj, int) = \sum_{\text{start, end} \in \text{int}} ObjOutInsDL(sds, obj, t)
\]

9. **ObjBiIntDL**(sds, obj, int) returns the interval data load for SDS sds that object obj sends or receives over time interval int. ObjBiIntDL can be calculated using ObjBiInsDL.

\[
ObjBiIntDL(sds, obj, int) = \sum_{\text{start, end} \in \text{int}} ObjBiInsDL(sds, obj, t)
\]

10. **ObjInIntML**(sds, obj, int) returns the interval message load for SDS sds that object obj receives over time interval int. ObjInIntML can be calculated using ObjInInsML.

\[
ObjInIntML(sds, obj, int) = \sum_{\text{start, end} \in \text{int}} ObjInInsML(sds, obj, t)
\]

11. **ObjOutIntML**(sds, obj, int) returns the interval message load for SDS sds that object obj sends over time interval int. ObjOutIntML can be calculated using ObjOutInsML.

\[
ObjOutIntML(sds, obj, int) = \sum_{\text{start, end} \in \text{int}} ObjOutInsML(sds, obj, t)
\]
12. ObjBiIntML(sds, obj, int) returns the interval message load for SDS sds that object obj sends or receives over time interval int. ObjBiIntML can be calculated using ObjBiInsML.

\[
\text{ObjBiIntML}(sds, obj, int) = \sum_{t=int \text{ start} \text{ int } \text{ start} + 1, \ldots \text{ int } \text{ end}} \text{ObjBiInsML}(sds, obj, t)
\]