Softw Syst Model
DOI 10.1007/s10270-008-0099-7

REGULAR PAPER

A UML-based quantitative framework for early prediction of resource usage and load in distributed real-time systems

Vahid Garousi · Lionel C. Briand · Yvan Labiche

Received: 22 June 2006 / Revised: 9 July 2008 / Accepted: 17 July 2008

Abstract This paper presents a quantitative framework for early prediction of resource usage and load in distributed real-time systems (DRTS). 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 early predictability of a DRTS by offering a systematic approach to predict, at the design phase, system behavior in each time instant during its execution. Since behavioral models such as sequence diagrams are available in early design phases of the software life cycle, the framework enables resource analysis at a stage when design decisions are still easy to change. Though we provide a general framework, we use network traffic as an example resource type to illustrate how the approach is applied. We also indicate how usage and load analysis of other types of resources (e.g., CPU and memory) 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 · Distributed systems · UML

Abbreviations

ASA	Automatic system agent
CCFG	Concurrent control flow graph
CCFP	Concurrent control flow path
CFP	Control flow path
DTCCFP	Distributed timed concurrent control flow path
LFQ	Load forecasting query
MBLF	Model-based load forecasting
MBPA	Model-based predictability analysis
MBRUA	Model-based resource usage analysis
NDD	Network deployment diagram
PA	Predictability analysis
DRTS	Distributed real-time system
RUA	Resource usage analysis
RUD	Resource usage definition
RUM	Resource usage measure
RUQ	Resource usage query
SCAPS	SCAda-based power system
SD	Sequence diagram
SDS	Sequence diagrams schedule
TC	Tele-control unit
TCCFP	Timed concurrent control flow path
UML-SPT	UML profile for schedulability, performance, and time

Communicated by Dr. Sebastien Gerard.

The initial work of this paper started when V. Garousi was a Ph.D. student at the Software Quality Engineering Laboratory at Carleton University.

V. Garousi (✉)
Department of Electrical and Computer Engineering,
Software Quality Engineering Research Group,
University of Calgary, 2500 University Drive NW,
Calgary, AB T2N 1N4, Canada
e-mail: vgarousi@ucalgary.ca

L. C. Briand
Simula Research Laboratory and University of Oslo,
P.O. Box 134, Lysaker, Norway
e-mail: briand@simula.no; briand@sce.carleton.ca

L. C. Briand · Y. Labiche
Department of Systems and Computer Engineering Software
Quality Engineering Laboratory, Carleton University,
1125 Colonel By Drive, Ottawa,
ON K1S 5B6, Canada
e-mail: labiche@sce.carleton.ca

1 Introduction

Distributed real-time systems (DRTS) (referred to as "systems" in the remainder of this paper) are becoming more important to our everyday life. Examples include command and control systems, aircraft aviation systems, robotics, and nuclear power plant systems [1]. Their development and verification and validation are difficult as engineers have to account for real-time constraints [1]. An important part of their verification and validation is to analyze how different resources (e.g., network, memory, CPU) are utilized. If a system overuses a resource (overload conditions) or uses it in an invalid way (e.g., violating mutual exclusion), functional and/or non-functional failures may be inevitable.

Predictability Analysis [2–4] aims at analyzing a system's resource usage before the system is deployed or even implemented. It is required by various activities that designers perform to assure that the systems they develop are safe, reliable, and satisfy time constraints. For example, *Resource usage analysis and verification* aims at ensuring that a system accesses resources in a valid manner as this can have an important impact on program correctness [5,6]. For example, there are often objects that should be accessed in a mutually exclusive manner. 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 [7]. *Load forecasting* is to predict server workloads [8,9], and has been used for a variety of purposes. For instance, load forecasting can be used to predict the running time of a task on a host by predicting the load the task would encounter on that host [9]. Load forecasting can also be used to predict load in the context of internet-based systems and web services (e.g., [10,11]). Load forecasting results are also usually used to perform load balancing and load sharing (e.g., [11]) to prevent load-related failures. Resource usage analysis aims at predicting the usage pattern of scenarios in a system for a resource type (e.g., CPU usage pattern of a given procedure), while load forecasting takes place at a higher level of granularity and is usually performed for hosts and distributed nodes to analyze the amount of load a node is processing (e.g., number of requests a web server is handling at a time instance).

The above analyses are typically performed late during software development, typically when the software is deployed, and one can analyze how the system actually utilizes resources. Alternatively, such analyses can be performed in early phases of software development, before any implementation is available, and the resource usage is then predicted. The advantage of the latter model-based approach is that during early stages of development, design decisions are still quite easy (and inexpensive) to change, whereas late design changes are known to be significantly more expensive [12]. One difficulty though is that any such early analysis must rely on estimates (e.g., execution times, transmission of data over the network) which are known to be difficult to obtain [1]. But approximations, simplifications, and assumptions are however necessary when the goal is to provide designers with a tool to make predictions based on what is known at design time. There are certainly many things that are not known yet regarding the implementation and execution that will affect resource usage and load. But having a tool allowing designers/architects to predict based on what they know at early design stages is a way to support early design decisions and identify potential bottlenecks that will have to be carefully watched in the remainder of development. Indeed, according to the Software Performance Engineering guidelines [13], if unacceptable resource usage and/or load are discovered late in the project, it is necessary to either abandon the system entirely or go through redefinition, redesign and redevelopment phases until the system becomes acceptable. Both of these options are much more expensive than validating resource usage and load goals from the beginning of the project.

In the context of model-based software development, the unified modeling language (UML) [14] is now the de-facto standard notation and has been adopted for building time-critical and resource-critical distributed systems. Therefore, model-based predictability analysis (MBPA) techniques (such as resource usage analysis and load forecasting) must be developed for exploiting UML models early in the design process. Several MBPA techniques have been proposed in the literature (e.g., [2,4,15–20]), from which a few make use of UML models, specifically activity and sequence diagrams. They however support analysis techniques that are different, though complementary to ours, such as response time and throughput analysis, schedulability analysis and optimal priority assignment [16]. Furthermore, no existing work addresses the problem of timed resource usage and load analysis based on the analysis of messages and their parameters and return values specified in UML models.

For practical reasons, in order to increase the applicability of our approach, we also aim at using UML models that have been built for the purpose of designing systems, as opposed to models specifically built for resource usage analysis.

This paper presents a quantitative UML-based MBPA framework. It consists in analyzing the control flow in UML 2.0 Sequence Diagrams (SD) [21] which are assumed to be augmented with timing information as it is usually the case for Real-Time (RT) systems [22]. Our ultimate goal is to investigate different aspects of MBPA. As a starting point, two MBPA methods are discussed in this paper: resource usage analysis and load forecasting; as they are among the most common analyses discussed in the literature (e.g., [2–4]), and we mainly focus on one resource type, namely network traffic. Two other resources (CPU and memory) are also discussed, though in much less details as their treatment

follows very similar principles. The approach is illustrated with examples and a case study is performed to demonstrate its feasibility.

Resource usage analysis (RUA) and load forecasting essentially relate to the behavior of a system, i.e., the executions of a system's behavioral scenarios consume resource and entail load on system entities. UML provides ways to model the behavior of an object-oriented (OO) system using interaction (e.g., sequence) diagrams. Therefore, performing RUA and load forecasting on behavioral UML diagrams should be advantageous compared to techniques based on source code (e.g., [6]) and runtime information since design decisions are significantly easier to change before the system is implemented.

The rest of this article is structured as follows. Section 2 provides background information, which includes an overview of the technique we use for control flow analysis of sequence diagrams. A survey of related works is presented in Sect. 3. Section 4 presents an overview of our predictability analysis approach: Prediction of resource usage is discussed in Sect. 5 and load forecasting is described in Sect. 6. Section 7 illustrates how our MBPA approach can be used for other purposes. Section 8 presents the set up and results of a case study. Finally, Sect. 9 concludes the article and discusses some of the future research directions.

2 Background

Section 2.1 introduces the *UML profile for schedulability, performance and time* (*UML-SPT*) [22], and Sect. 2.2 summarizes the technique we have developed to perform control flow analysis of UML 2.0 sequence diagrams [23].

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 [24–28]. 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 designers. In particular, the lack of a quantifiable notion of time was recognized as an obstacle to its broader use in the distributed and embedded domains. To address those needs, the OMG adopted the UML-SPT [22].

Note that the UML-SPT was the standard profile when the first draft of this paper was prepared. As of the time of writing the final version of this paper (March 2008), the OMG is working on the finalization stage of a new improved profile, called the *UML profile for modeling and analysis of real-time and embedded systems* (*MARTE*)[29], which is expected to replace SPT in the near future. The framework reported in this paper can be modified to be applicable with MARTE as its time modeling specification seems to be similar to that of SPT.

The UML-SPT proposes a framework for modeling real-time systems using UML. The profile is now popular in the research community [15,30–32] and is getting accepted in industry [33]. An example SD annotated with timing information using the UML-SPT is shown in Fig. 1. It illustrates some of the extensions to UML that are specified in the UML-SPT.

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). *RTstart* and *RTend* tagged-values indicate "*the start and end times of the stimulus, corresponding to the stimulus generation and stimulus reception*" [22]. In Fig. 1, message *A* is sent at time 0 ms, which gives a time origin for the sequence diagram (see below) and received at time 2 ms, and message C is sent at time 7 ms (after *A*). Both *RTstart*and *RTend* are of type *RTtimeValue* [22], which is a general mechanism for specifying time values in UML-SPT models. The general format for expressing time value expressions using *RTtimeValue* is described in [22]:

$$\langle timeValStr\rangle ::= (\langle timeStr\rangle|\langle dateStr\rangle|\langle dayStr\rangle|\langle metricTimeStr\rangle) [", "\langle clock-id\rangle]$$

where *timeStr, dateStr, dayStr* and *metricTimeStr* are used to express second/minute/hour, day/month/year, days of a week, and Probability Distribution Functions for time, respectively. The optional *clock-id* value denotes the clock mechanism used, e.g., International Atomic Time (TAI).

Note that there is no constraint in the UML-SPT profile requiring that every message in a SD be specified with timing information, and our approach does abide to this (note that this is not the case in Fig. 1). We will see later in this document that, since our technique is based on specified timing information, messages without timing information are simply not accounted for during resource usage and load analysis. As for any model-based technique, the results of our technique (e.g., accuracy) depend on the amount and quality of the information provided as input.

As specified in the UML-SPT, timing information is either absolute or relative, and in the latter case, it is "*relative to the event whose timing mark value is zero*" [22]. This is the case in Fig. 1 where the first message (i.e., A) has a starting timing mark of 0, and therefore, all the other messages' timing information are relative to this message's start time. If no such event exists, "*the interpretation of the* [*timing*] *values is application specific* (*i.e., they may be relative or absolute depending on convention*)" [22]. In this latter case, the UML-SPT does not indicate how the "application specific" semantics is specified by the designer (using the UML-SPT notation). For instance, the designer could have a

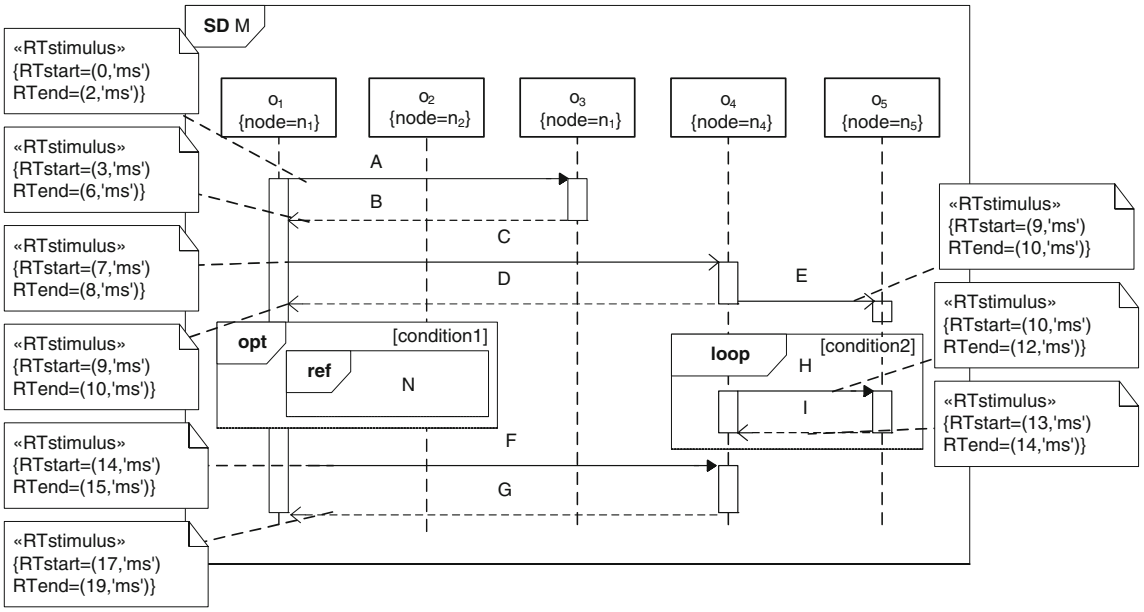


Fig. 1 An (abstract) example SD annotated with timing information using the UML-SPT

separate application specific semantics for each SD, or an application specific semantics for all the SDs of the system, and this information would likely be specified with some UML-SPT or MARTE stereotype.

In order to proceed with our approach, we adopted one possible semantics which matches standard UML-based object-oriented software development (e.g., [34]). If a SD corresponds to a use case that is directly triggered by an actor, then we require that the first message of the SD be specified with a start time equal to 0. Otherwise, the SD is triggered by another SD (using an *InteractionUse* construct [14]), which corresponds to either an *include* or an *extend* relationship between the corresponding use cases. If SD *A* is triggering SD *B*, then the triggering time of *B* (as specified in *A*) is the time origin for the time information of all the messages in *B*, and the first message in *B* does not need a timing information equal to 0.

It is worth noting that the SPT specification (for UML 1.x) does not indicate how timing information associated to messages in a loop combined fragment (UML 2.0) have to be interpreted. Recognizing that a UML 2.0 loop combined fragment is in fact a while loop, i.e., the condition is evaluated before each iteration [14], we made the following assumptions: the time it takes to evaluate the condition of a loop is the same for each iteration of the loop and is merged with the triggering time of the first message of the loop (e.g., evaluation of condition 2 in Fig. 3 takes 1 ms and is included in the duration of message *H*). This is a temporary but reasonable solution while waiting for an update in the final version of MARTE. Start and end times of messages in loop iterations (after the first one) are calculated accordingly based on the original timing information provided in the SD, e.g., in the second iteration of the loop in Fig. 3, *H* and *I* will start at 14 and 17 and end at 16 and 18, respectively (14 ms was the end time of message *I* in the first iteration).

UML 2.0 sequence diagrams and their composition are based on the semantics of Message Sequence Charts (MSCs) [35]. There has been some work on adding the notion of time to MSCs. For instance, time constraints between messages in one or different MSCs can be defined [36]. This is very similar to our assumption regarding included SDs discussed above, except that instead of asking the designer to define a time constraint (between the triggering message of the included SD and the first message of the included SD), we assume the two time instants are the same. More details on time and compositional semantics of SDs and MSCs can be found in related work (e.g., [36,37]).

Note that the current article assumes that timing information is already available, and that some form of schedulability analysis has been successfully performed (e.g., [38]). A number of techniques can be used to estimate these message execution times. Most of them either rely on some form of (semi-)automated source code analysis of the tasks (e.g., [39]) or some type of runtime monitoring of task executions (e.g., [40]). These techniques cannot be used in our context since they require that the system be implemented whereas we assume our approach is used during the design stages. Another approach to obtain such information, at design time, is to estimate the source lines of code for tasks (e.g., using past experience in developing similar tasks) and use benchmarks of the selected programming language executions on the selected hardware [41]. Early predictions necessarily imply

```
1  SD::InteractionFragment.allInstances->forAll(interFrag:InteractionFragment|
2   CCFG::InitialNode.allInstances->exits(in:InitialNode|
3     in.activity=Utility::Util.getCCFG(interFrag) and
4     in.outgoing->includes(flow:ControlFlow|
5      getCCFG(interFrag).node->exits(an:ActivityNode|
6        an.inFlow->includes(Utility::Util.getFirstMessage(interFrag).sendEvent)
7        and flow.target=an
8       )
9     )
10  )
```

Fig. 2 OCL specification of rule #2 in Table 1

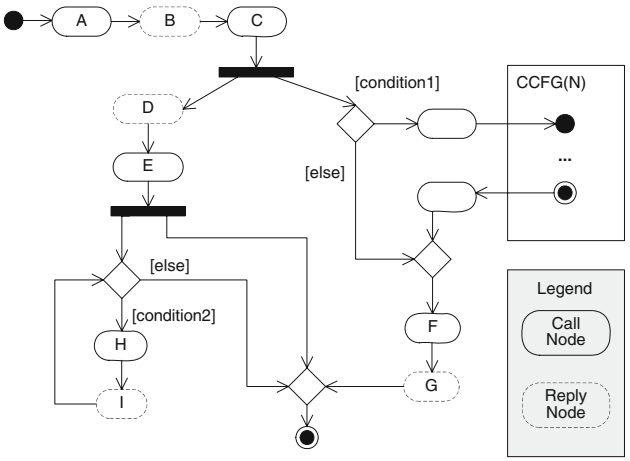


Fig. 3 CCFG of the SD in Fig. 1

simplifications, approximations, and assumptions (e.g., one can specify pessimistic, optimistic, and expected loads using the profile), as discussed by Gomaa [41], but allow early design assessments and decision making. It is a tradeoff. Our approach is meant to be practical, scalable, and cost-effective and is hence not based on complex, detailed modeling and complex model transformations. Only case studies will tell whether assumptions are acceptable, e.g., our controlled experiment in Sect. 8, and this is the objective of the study reported in this work. For instance, our recent work on model-based stress testing [42], that relies on the resource usage and load prediction techniques reported in the current paper, showed that, even if the timing information of message are estimated, one can define stress test cases that do entail maximum network traffic and reveal faults.

Last, it is worth noting that the level of precision of the time information will of course drive the level of precision of the RUA results.

2.2 Control flow analysis of UML 2.0 SDs

The UML 2.0 SD notation [21] provides various program-like constructs such as conditions, loops, and procedure calls, as well as asynchronous messages and parallel combined fragments to model parallelism, thereby allowing a designer to model a large variety of flows of executions in one single view.

We presented a technique to analyze these flows of executions [23]. Based on the well-defined UML 2.0 activity diagrams, we proposed an extended activity diagram metamodel, referred to as Concurrent Control Flow Graph (CCFG), to support control flow analysis of UML 2.0 SDs. Each SD is transformed into a CCFG and the mapping between a SD and the corresponding CCFG was formally specified as a set of Object Constraint Language (OCL) [43] rules. Nodes in a CCFG correspond to messages in the SD, and edges in a CCFG denote possible sequential and concurrent flows of executions between messages in the SD. Concurrency in a SD (either asynchronous messages or *par* combined fragments) map to fork and join nodes in the CCFG. Additionally, when a SD refers to another SD, there exist edges connecting their corresponding CCFGs to form an *Inter-SD*CCFG, which is a concept similar to the inter-procedural CFG [44]. We defined 14 OCL mapping rules, listed in Table 1, that relate elements of an instance of the SD metamodel to elements of an instance of the CCFG metamodel. As an example, Fig. 2 shows the OCL specification of rule #2 in Table 1. This rule specifies that the first message of an SD is mapped to the edge outgoing from the initial node of the corresponding CCFG. More specifically, each interaction fragment in a SD (line 1 in Fig. 2) has an initial node (lines 2–3), and this initial node's outgoing flow (i.e., edge in the CCFG) maps to the first message of the interaction fragment (lines 4–7). Note that to reduce the complexity of our OCL rules, we defined several utility functions inside a utility class *Util*. In Fig. 2, *getCCFG*() returns the CCFG mapped to an interaction fragment, and *getFirstMessage*() returns the first message of a given interaction fragment. Additional details, including a description of the CCFG metamodel and all the OCL consistency rules, are provided in a technical report [45].

As an illustrative (abstract) example, the SD of Fig. 1 is mapped to the CCFG of Fig. 3 by applying our transformation

Table 1 Mapping rules from SDs to CCFGs [23]

Rule #	SD feature	CCFG feature
1	Interaction	Activity
2	First message end	Flow between initial node and first control node
3	SynchCall/SynchSignal	Call node
4	AsynchCall or AsynchSignal	(Call node + Fork node) or reply node
5	Message send event and receive event	Control flow
6	Lifeline	Object partition
7	*par* Combined fragment	Fork node, join node and activities for the interaction fragments
8	*loop* Combined fragment	Decision node and activities for the interaction fragments
9	*alt/opt* Combined fragment	Decision node and activities for the interaction fragments
10	*break* Combined fragment	Activity edge and activities for the interaction fragments
11	Last message ends	Flow between end control nodes and final node
12	Interaction occurrence	Control flow across CCFGs
13	Polymorphic message	Decision node
14	Nested interaction fragments	Nested CCFGs

technique[1]: Activity node B in Fig. 3 corresponds to message B in Fig. 1; The flow from activity node A to activity node B in Fig. 3 corresponds to message Abeing ahead of message B in terms of control flow in Fig. 1; The decision node in Fig. 3 with two outgoing edges labeled with [*condition1*] and [*else*] corresponds to the alternative interaction fragment labeled with [*condition1*], a generic condition, in Fig. 1; The fork nodes in Fig. 3 are due to asynchronous messages C and E in Fig. 1; The CCFG of triggered SD N (in the alternative combined fragment in Fig. 1) is not detailed in Fig. 3 but edges to its start node and from its end node appear, illustrating the notion of inter-SD CCFG.

We showed in [23] that our OCL rules and CCFG metamodel are complete, i.e., the CCFG metamodel has all the classes and associations needed, and the rules are adequate for the purpose of control flow analysis.

Based on the CCFG, we defined the notion of Concurrent Control Flow Paths (CCFPs), which are a generalization of the conventional Control Flow Path concept to account for concurrency [23]. Each CCFP is defined as a concurrent path, starting from the initial node of the CCFG to its final node. Concurrent paths include *all* the branches going out from a fork node in a CCFG. A CCFP is derived by traversing the CCFG from the initial node to the final node and concatenating all the nodes in the path, in order. Similarly to traditional control flow analysis, special considerations regarding decision nodes should be made in terms of conditions and loops, i.e., two different CCFPs for true/false edges of a conditional node should be derived. For instance, the loop made of nodes H and L in Fig. 3 can lead to an infinite number of paths.

[1] Note that the UML-SPT notation in Fig. 1 is not required for control flow analysis. UML-SPT is used in Fig. 1 as this SD will be our running example in Sect. 5.

$$\rho_1 = ABC\left(DE\left(\begin{matrix}\\ FG\end{matrix}\right)\right) \qquad \rho_2 = ABC\left(DE\left(\begin{matrix}HI\\ FG\end{matrix}\right)\right)$$

$$\rho_3 = ABC\left(DE\left(\begin{matrix}(HI)^2\\ \rho_{N,1}FG\end{matrix}\right)\right) \qquad \rho_4 = ABC\left(DE\left(\begin{matrix}(HI)^3\\ \rho_{N,2}FG\end{matrix}\right)\right)$$

Fig. 4 Some of the CCFPs of the CCFG in Fig. 3

Applying a strategy often used when deriving control flow paths from source code, one can try to bypass the loop (if possible), take it only once, a representative or average number, and a maximum number of times (other *loop bound analysis* can be used to identify minimum and maximum numbers of execution [46]). Furthermore, when SDs call each other (as it is the case in Fig. 1), the CCFPs of the caller SD will include the CCFPs of the called SD.

Applying the above procedure generates a number of CCFPs, four of which are shown in Fig. 4: each CCFP is identified by symbol ρ, which will denote CCFPs in the remainder of this article: ρ_1 (the loop of SD M is bypassed), ρ_2 (the loop is taken once), ρ_3 (the loop is taken a representative number of times, in this case twice), and ρ_4 (the loop is taken a maximum number of times, in this case three times). $\rho_{N,1}$ and $\rho_{N,2}$ are two of the CCFPs of the SD N which are included in ρ_3 and ρ_4.ρ_1 and ρ_2 are derived by taking the false (else) branch of *condition1*, while ρ_3 and ρ_4 are derived by taking its true branch.

These CCFPs are represented textually based on a grammar specified in [23]. Following our notation, ρ_3 specifies that the CCFP starts with nodes A, B, and C executed in sequence, followed by two concurrent executions (fork nodes in Fig. 3): the execution of F followed by G on one hand, the execution of D and E followed by two executions of H and I in sequence (loop), on the other hand.

When messages of a SD are annotated with timing information (using the UML-SPT profile), we refer to the corresponding 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 Fig. 4 are also TCCFPs (although our notation for CCFPs does not indicate timing information, the mapping between the SD and the CCFG, and then the identification of CCFPs allows us to retrieve timing information and identify whether a CCFP is a TCCFP).

3 Related work

Among MBPA techniques reported in literature (e.g., [2,4, 15–20]), a few make use of UML models, specifically activity and sequence diagrams, to analyze resource usage and load (e.g., [16]). A comprehensive survey of recent research in the field of model-based performance prediction in software development is reported in [17].

UML-MAST (Modeling and Analysis Suite for Real-Time Applications) [16] is a modeling and analysis suite. It was developed concurrently to UML-SPT with similar goals but it resulted in slightly different extensions of UML (see [16] for details). For instance, UML-MAST suggests the use of specific stereotypes (e.g., *RT_Situation*) to model workloads and timing requirements. UML-MAST diagrams are annotated with resource usage information such as minimum and maximum transmission times of data over network and timing information such as Best- and Worst-Case Execution Times. This data can then be used for analysis purposes, e.g., blocking times of resources (due to mutual exclusion), and worst response times. Besides the fact that UML-MAST suggests extensions to UML that have not been standardized by the OMG—our choice of UML-SPT is indeed primarily driven by its standardization, the fundamental difference between UML-MAST and our work is that MAST supports analysis techniques that are different though complementary to ours: they focus on schedulability analysis and optimal priority assignment (UML-MAST) whereas we investigate resource usage and load analysis.

Other techniques reported in literature rely on either behavioral models [4,15] or structural (architectural) models [2], and, similarly to [16], focus on different though complementary analysis techniques: validation and performance evaluation [2,15], and schedulability analysis [4].

In Bernardi et al. [15], the authors 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", i.e., scenarios of execution. 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, 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 control flow analysis 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.

The work in Feiler et al. [2] presents a model-based architectural approach for improving predictability of performance in embedded systems. 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 [47] is used as the modeling language. 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, and thus yields more accurate predictability. Since this work is based on architectural models (which are structural, rather than behavioral), its ability to predict behavior cannot be evaluated.

Yau and Zhou [4] presented an approach to incorporate schedulability analysis into existing frameworks for model-based software development. The goal is to improve the predictability of a system 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 system 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.

For the purpose of performance analysis, there exist several works (e.g., [18–20]) which transform UML models into different types of performance models, e.g., Layered Queuing Networks (LQN). The technique reported in [18] generates (scenario) *Precedence Graphs* (*PG*) from UML SDs. Then, a *global* precedence graph merges different scenario PGs and represents the behavior of the whole system. An Extended Flow Graph is then generated based on the global PG, in which workload data (from operational profiles) are incorporated. A Layered Queuing Network (LQN) is built afterwards, that is used to perform different types of performance analysis, e.g., response time analysis. A similar approach is presented in [19]. It differs from [18] in that it uses the information specified in use case diagrams (in addition to SDs) to derive a system's operational profile, and it uses deployment diagrams to derive hardware and network attributes (e.g., network bandwidth). The fundamental differences between [18,19] and our work are two-fold: (1) they consider earlier versions of UML (i.e., 1.x) while our work supports UML 2 (Sect. 2.2), and (2) they focus on other types of performance analysis than resource usage and load analysis and do not rely on any analysis of message parameters and their sizes.

The book by Smith and Williams [20] is a comprehensive body of knowledge on Software Performance Engineering (SPE) in the context of UML-driven development which more or less includes the ideas and techniques discussed in the two papers [18] and [19]. Published in 2002 (around the time UML 2 was being finalized), it discusses that complex UML elements (e.g., combined fragments in SDs) should be taken into account in precise performance analysis based on UML models. However, it does not provide the how-to on the issue, i.e., how to perform control flow analysis when SDs have asynchronous messages and concurrent fragments. Furthermore, similar to other works, it does not discuss resource usage and load analysis based on message parameters and their corresponding data sizes.

In summary, our approach's novelty is that parameters and return values of messages in SDs and their corresponding data sizes are accounted for during resource usage and load analysis, and that we fully support the UML 2 SD notation. The main technical contribution of our work is the way SD messages and their parameters are analyzed, using the notions of *Resource Usage Measure, Resource Usage Query*, and *Resource Usage Query* (Sect. 5). No other work takes such a quantitative approach in resource usage prediction.

4 Model-based predictability analysis: an overview

The activity diagram of Fig. 5 provides an overview of our MBPA approach. The diagram conforms to the *general model-processing framework*, proposed by the UML-SPT profile [22], where our technique acts as a *model processor* that takes a UML model as input, plus some additional resource-dependent inputs (for resource usage analysis and load forecasting), and generates the analysis results.

The UML behavioral model used as input is a set of SDs augmented with timing information using UML-SPT: the SDs are used to identify control flow paths and therefore possible scenarios involving resource usage. The structure model (class diagrams) is used to identify generalization relationships among classes to be able to appropriately handle polymorphic behaviors of objects in SD lifelines when analyzing control flow [23]. Furthermore, as discussed in Sect. 5, resource usage analysis will use the internal structure of classes in the system. *Network Deployment Diagrams* (*NDD*) in Fig. 5 are models to describe the distributed architecture (Sect. 5.1.2) of a system, a piece of information required for

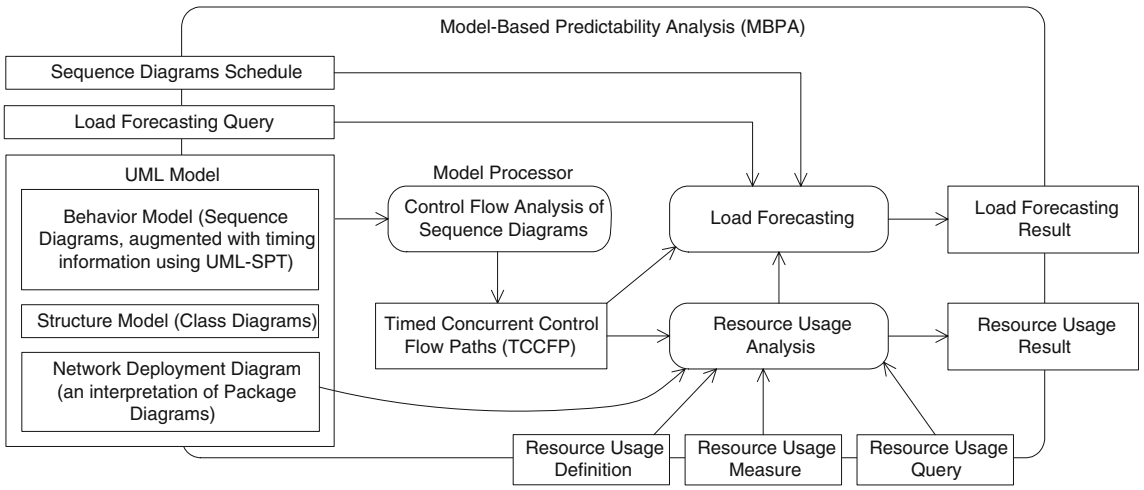


Fig. 5 Overview activity diagram of the model-based predictability analysis (MBPA) approach

network traffic analysis and prediction, which is the type of RUA we focus on in this paper.

The technique then analyzes the control flow in the input model (Sect. 2.2) and uses the resulting TCCFPs for resource usage analysis and load forecasting. The *Resource Usage Definition* (*RUD*), *Resource Usage Measure* (*RUM*) and *Resource Usage Query* (*RUQ*) are input parameters for the Resource Usage Analysis activity, and are used to tailor the analysis to different types of resources: e.g., CPU, memory, network traffic. The above concepts are discussed in detail in Sect. 5.

Thanks to the concepts of RUD and RUM, our MBPA approach is flexible as it can be easily tailored and applied to a variety of resource types (e.g., network, CPU, memory, disk, and database). In the current article, we discuss in detail the use of this approach to one resource, specifically network traffic, and mention two applications to other resources, specifically CPU and memory usage.

5 Resource usage analysis

In this article, we primarily tailor the approach discussed previously to one resource type: network traffic. Section 5.1 provides a set of fundamental definitions and concepts which will be used for network traffic usage analysis. Specific RUD, RUM, and RUQ for network traffic are presented in Sect. 5.2, Sect. 5.3, and Sect. 5.4, respectively. We then define in Sect. 5.5 a set of traffic usage measures. To demonstrate the applicability of the approach to other resource types, Sect. 5.6 briefly presents how to tailor the approach (i.e., discusses possible RUD and RUM) to the CPU and memory resource types.

5.1 Definitions

5.1.1 Formalizing sequence diagram messages

As discussed in Sect. 4, our technique needs to manipulate SD messages, and we therefore need a formal representation for SD messages. Similar to the tabular notation for UML 2.0 SDs in Appendix D.1 of the UML 2.0 standard [21], each SD message, in the design model of a distributed system, can be denoted as a tuple:

message = (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 *sender = (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 has the same form as *sender*.
- *method Or SignalName* 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 [21]. 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 $\langle (p_1, C_1, in/out), \ldots, (p_n, C_n, in/out)\rangle$, where p_i is the $i-$th parameter name of class type C_i and *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 $\langle (o_1, C_1, in), (o_2, C_2, in)\rangle$. 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 $\langle (var_1 = val_1, C_1), \ldots, (var_n = val_n, C_n)\rangle$, where val_i is the return value for variable var_i with type C_i.
- *startTime* is the start time of the message (modeled by UML-SPT's *RTstart* tagged value).
- *endTime* is the end time of the message (modeled by UML-SPT's *RTend* tagged value).
- *msgType* is a field to distinguish between signal, call and reply messages. Although the *msgSort* 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 [23].

As an example of this formalism, message A in the SD in Fig. 1 is represented as $((o_1, C_1, n_1), (o_2, C_2, n_2), A, synchCall, null, null, \text{`0}\,ms\text{'}, \text{`2}\,ms\text{'}, \text{`}Call\text{'})$. Using this formalism, different fields of a message are accessed using the record notation. For example, given a message m, *m.sender.object* refers to the sender object of message m.

5.1.2 Network deployment diagram

A Network Deployment Diagram (NDD) models the distributed architecture of a system and is specifically needed for the RUA in this article (network traffic resource type). A NDD complements the information available in SDs (and other UML diagrams) about the deployment of objects on nodes by describing the topology of a distributed system. NDDs are

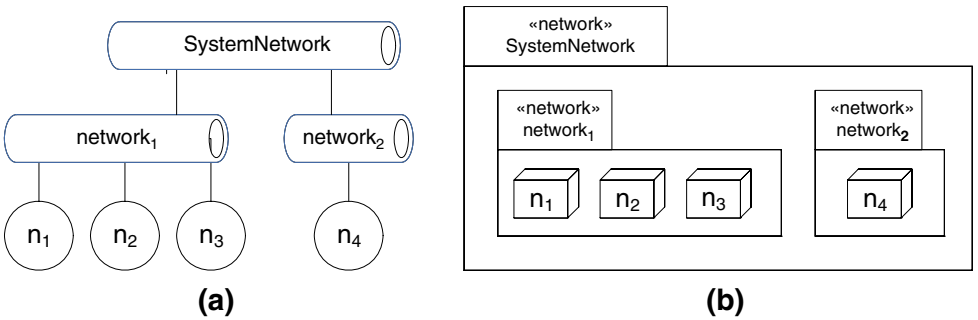


Fig. 6 **a** An example of a hierarchical distributed system. **b** The corresponding network deployment diagram (NDD)

an extension to UML 2.0 package structures [21], 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 Fig. 6a and b, respectively.

As discussed later, analyzing the usage of resource network requires that we identify paths in the network that are involved when messages are sent. To identify the network path between any two given nodes, we define the network path function *getNetworkPath*(n_s, n_r) where n_s and n_r are the sender and the receiver nodes of a message, respectively. For example, the derivation of the network path between n_1(the sender) and n_4 (the receiver) is depicted in Fig. 6b and is formally represented as:

$$getNetworkPath(n_1, n_4) = \langle network_1, SystemNetwork, network_2 \rangle$$

In this paper, we make a simplifying assumption: we assume there is only one path between any two given nodes in the network and therefore the *getNetworkPath* function returns only one path. As a result, the proposed approach is, as reported here, only applicable to systems in which there is only one network path between any two given nodes (for example, our case study system in Sect. 8). As this is out of the scope of the current paper, the reader is referred to [48] for a detailed discussion about how to relax this assumption. In general, there are several possible paths between two nodes in the network, and the data sent from a node to another is actually divided into several parts and transmitted through those paths. The dispatching of parts is handled by networking components of the system (e.g., routers).

Assuming the network paths' dispatching policy does not change during the transmission of a message (adaptive dispatching policies are usually used to balance load in each time instance), the transmission shares of each of the involved networks stay the same during the entire transmission. Further assuming that the share only depends on the number of paths between the two nodes (in practice, more traffic is usually sent through networks with higher bandwidths), we define function *netTransmissionShare* to compute the share of a network between two nodes as the ratio of the number of paths between two nodes in which the network is a member of, to the total number of paths between two nodes. Using additional information on the actual network infrastructure on the distributed system, these two assumptions (the share depends on the number of paths and does not change over time) can be relaxed and a more accurate *netTransmissionShare* function can be defined. The *netTransmissionShare* function is used in the network traffic usage formulas in the rest of this article to calculate the amount of traffic on a specific network. This will enable our test methodology to estimate the anticipated traffic on a network.

5.2 Resource usage definition

Recall that a RUD is a set of criteria to select relevant elements in a behavioral model for a specific resource. Given that we are interested in network traffic, we must identify from SDs how the network is used, i.e., messages that involve the network. One possible RUD for network traffic usage is therefore to transform TCCFPs into Distributed TCCFPs (DTCCFPs) by removing local messages (sent between two objects on a node) and therefore keeping only the distributed ones (TCCFPs and DTCCFPs are types of CCFPs, which are ordered sequences of concurrent message). A formal definition of this RUD is given in Eq. 1.

$$RUD_{network} : TCCFP \rightarrow DTCCFP$$
$$\forall \rho \in TCCFP : RUD_{network}(\rho) = \rho - \underbrace{\{msg \mid msg \in \rho \wedge msg.sender.node = msg.receiver.node\}}_{\text{Local messages in } \rho}$$

Equation 1 RUD of the network traffic usage analysis technique.

The start and end times of the messages in a TCCFP or a DTCCFP are extracted from the timing information specified (using the UML-SPT profile) in the SD. As an example, to derive the DTCCFPs of the TCCFPs in Fig. 4, we first determine if each message in the TCCFPs is local or distributed. According to the corresponding SD (Fig. 1), all the messages except *A* and *B* are distributed. Hence, using the RUD in Eq. 1, the DTCCFPs corresponding to the TCCFPs in Fig. 4 are shown in Fig. 7.

$$DTCCFP(\rho_1) = C\left(\frac{DE(\ )}{FG}\right) \quad , DTCCFP(\rho_2) = C\left(\frac{DE(HI)}{FG}\right)$$
$$DTCCFP(\rho_3) = C\left(\frac{DE((HI)^2)}{FG}\right) \quad , DTCCFP(\rho_4) = C\left(\frac{DE((HI)^3)}{FG}\right)$$

Fig. 7 DTCCFPs of the TCCFPs in Fig. 4

5.3 Resource usage measure

Recall that a RUM is a function to measure the usage of a resource by the model elements selected by a RUD. In the case of resource network, we define a RUM to measure (in terms of resource usage) the distributed messages identified in DTCCFPs, i.e., we measure the data traffic entailed by messages on a network: $\text{RUM}_{network}(\text{msg})$ (Eq. 2).

$$\forall msg \in Message : RUM_{network}(msg) = \begin{cases} CallDT(msg) & ;\text{if } msg.msgType = 'Call' \\ ReplyDT(msg) & ;\text{if } msg.msgType = 'Reply' \\ SignalDT(msg) & ;\text{if } msg.msgType = 'Signal' \end{cases}$$
$$CallDT(msg) = \sum_{C_i|(-,C_i,-)\in msg.parameterList} dataSize(C_i)$$
$$ReplyDT(msg) = \sum_{C_i|(-,C_i)\in msg.returnList} dataSize(C_i)$$
$$SignalDT(msg) = dataSize(msg.methodOrSignalName)$$
$$\forall C \in classDiagram : dataSize(C) = \sum_{a_i \in C.attributes} dataSize(a_i)$$

Equation 2 RUM to analyze network traffic usage.

The most data centric parts of a call, a reply, and a signal message are *parameterList*, *returnList* (Sect. 5.1.1), and the attributes of the corresponding signal class. *CallDT*, *ReplyDT* and *SignalDT* denote the amount of Data Traffic for a call, reply, or signal message. The data traffic for a call message (*CallDT*), is the summation of data sizes of all the attributes of each parameter class. For a reply message, data traffic is the summation of data sizes of all the attributes of each class in the return list (*ReplyDT*). Data traffic for a signal message (*SignalDT*), is the data sizes of all the 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 the 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 *parameterList* and *returnList* can be an estimate for the data sizes of call and reply messages. Admittedly, other measures of network traffic can be considered. For example, a more accurate estimate would also account for the 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. This, however, requires a detailed, platform-specific analysis of packet and frame

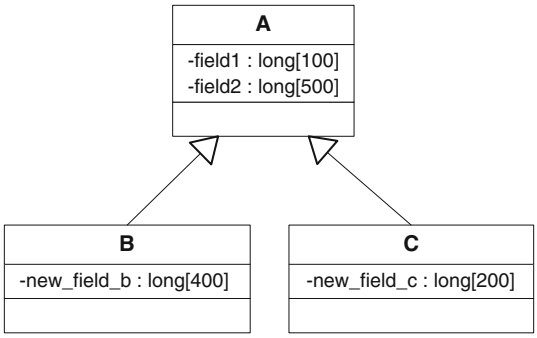


Fig. 8 A class diagram showing three classes with data fields

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 (Sect. 8).

A dash (-) in Eq. 2 indicates that a field can take any arbitrary value. Note the format of *parameterList* and *returnList*, as mentioned in Sect. 5.1. *msg.parameterList* (*msg.returnList*) is the sequence of parameters (returns) for a call (reply) message. The function $dataSize(C_i)$ returns the data size of the class C_i. *classDiagram* is the set of classes (it can be extracted from the system's class diagram). *C.attributes* denotes the set of attributes of class C, accounting for inherited attributes (again, it can be extracted from the system's class diagram). $size(a_i)$ 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., *int*, *long*, *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 *short*, *int* and *long* are two, four and eight bytes in Java, respectively. In case an attribute a_i is itself an object, the size of that attribute, size (a_i), will be the size of its class and can be calculated recursively using Eq. 2. As an example, suppose a call message msg_1 with $parameterList = \langle (o_1, A), (o_2, B) \rangle$, where classes A and B are defined in the class diagram of Fig. 8. Using the class specifications of A and B, we can estimate the size of the message msg_1 as: $size(msg_1) = size(A) + size(B) = (8^*(100 + 500)) + (8^*(100 + 500) + 8^*400) = 12.8KB$

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 means that two parameters of class type A (defined in Fig. 8) are passed to method m and an object of type A is returned. Since B and C are

sub-classes of A (Fig. 8), an object of type B and C can also be an argument of method m, which will cause the message to have different data sizes since B and C both define an extra attribute (compared to A). At least three 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 ($dataSize(m)$ only counts the two parameters, thus the multiplication by factor 2, since the return value is not part of the size of the call message, but part of the return message):

$$\begin{aligned} dataSize(m) &= 2^{*}max(dataSize(A), \\ &\quad dataSize(B), dataSize(C)) \\ &= 2^{*}(8^{*}(100+500), 8^{*} \\ &\quad \times(100+500+400), \\ &\quad 8^{*}(100+500+200)) = 16KB \end{aligned}$$

2. The probabilities of a superclass and its subclasses to be the run-time type of an argument (given in an operational profile for instance) can be 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:

$$\begin{aligned} dataSize(m) &= 2^{*}[p(A)^{*}dataSize(A) + p(B) \\ &\quad {}^{*}\, dataSize(B) + p(C)^{*}dataSize(C)] \\ &= 2^{*}[0.6^{*}8^{*}(100+500) \\ &\quad + 0.3^{*}8^{*}(100+500+400) \\ &\quad + 0.1^{*}8^{*}(100+500+200))] \\ &= 11.84KB \end{aligned}$$

3. The data size of the superclass itself can be used, i.e., one uses a minimum value, not accounting for the child classes' attributes. This approach is accurate when the programming language or the programming conventions selected during implementation enforces that when a parameter type is defined as being A, no subclass of A can be used. Using this approach, the data size of the message m (above) can then be calculated as::$dataSize(m)=2^{*}$ $dataSize(A)=2^{*}(8^{*}(100+500))= 9.6\ KB$

5.3.2 Indeterminism in messages sizes

Some data sizes never change given a programming language (e.g., an *int* in C++ is four bytes) whereas some data sizes may change from one message to another, from one execution to another (e.g., a string). In the latter case, Eq. 2 can not be applied to estimate the data size of a message. Other data structures such as linked lists, hash tables, and trees also may have variable sizes and introduce indeterminism during data sizes estimation.

One simple approach to estimate data size in this latter case is to monitor a sufficient number of runs of the system, measure data sizes and build statistical distributions of size values. This, however, requires that the system be implemented, whereas we assume that our approach is used during design. During design, similarly to the identification of time information in SDs (Sect. 2.1), we have to resort to estimates based for instance on benchmarks [41].

5.4 Resource usage query

Recall that a RUQ is a querying mechanism for RUA used to focus on some particular aspects of the resource usage in a system. For the network traffic usage analysis, we define four query attributes:

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

A RUQ can be a combination of the above four query attributes, such as: *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 scenario, a message is initiated by a sender node, travels along a network path (that consists of one or more networks—Sect. 5.1.2) from the sender to a receiver node, and arrives at the destination node where it is supposed to be handled appropriately. It is therefore important to be able to identify the network path between two nodes. Furthermore, since messages can have different destination (or source) objects, possibly hosted by one node, a fined grained analysis can focus on the traffic involving one particular object (instead of a node or network).

5.4.2 Direction (for nodes only): in, out, or bidirectional

In case traffic location is node or object, we can think of three measurements in terms of traffic *direction*: *in, out* or *bidirectional*; because a node or an object is an end point of traffic in the system. Since a network in the system only relays traffic, i.e., it transmits 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 types: the amount of data (i.e., size) transmitted between two distributed objects, and the frequency (or 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 may frequently send queries to n_A, and get replies, but each message and reply may have small data sizes. Therefore, both the amount of data and the frequency of messages are relevant to us. These two notions have already been used in a different context, specifically stress testing distributed systems, where it has been shown that deriving stress test cases using these two traffic types may reveal different types of faults [49]

5.4.4 Duration: instant or interval

One can analyze network traffic at a given time instant or over a period of time (or time interval). In the latter case, one may for example be interested in an average amount of traffic over the network. 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 system will never overflow. The time unit that we consider here is the time measurement precision (e.g., milliseconds) used by the designer in SDs (Sect. 2.1).

5.5 Resource usage analysis functions

We define a set of traffic usage functions for DTCCFPs which are classified based on the query attributes defined in Sect. 5.4. The naming convention of the functions is discussed in Sect. 5.5.1. For brevity, only formal definitions of resource usage functions for network traffic location are presented in Sect. 5.5.2. The functions for node and object usage locations are derived in a similar fashion and are presented in [48]. 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 Fig. 9. 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.

Five layers are shaded in the tree. The top four layers correspond to the four query attributes discussed in Sect. 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 Fig. 9 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 Fig. 9, i.e., function *NetInsDT*. Its input parameter is (*ρ, net, t*). This function returns the instant (*Ins*) data traffic (*DT*) value of a given DTCCFP (ρ) for a given network (*net*) at a given time (t). Input parameters including *int* in the bottom layer of Fig. 9 correspond to the functions with interval duration. The start and end times of an interval, i.e., *int*=(*start, end*), must 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. Functions with network traffic location are described in Sect. 5.5.2. Similar functions can be defined for node and object traffic location.

5.5.2 Traffic location: network

1. *NetInsDT*(*ρ, net, t*) returns the instant data traffic for DTCCFP ρ on network *net* at time t.

$$NetInsDT(\rho, net, t) = \begin{cases} \sum_{msg_i \in Message} netTransmissionShare(net, msg_i.sender.node, msg_i.receiver.node). RUM_{net}(msg_i)/dur(msg_i) \\ 0 \end{cases}$$

; $|Message| > 0$, where

$$Message = \left\{ msg_i | msg_i \in \rho \wedge msg_i.start \leq t \leq msg_i.end \wedge net \in getNetworkPath(msg_i.sender.node, msg_i.receiver.node) \right\}$$

;otherwise

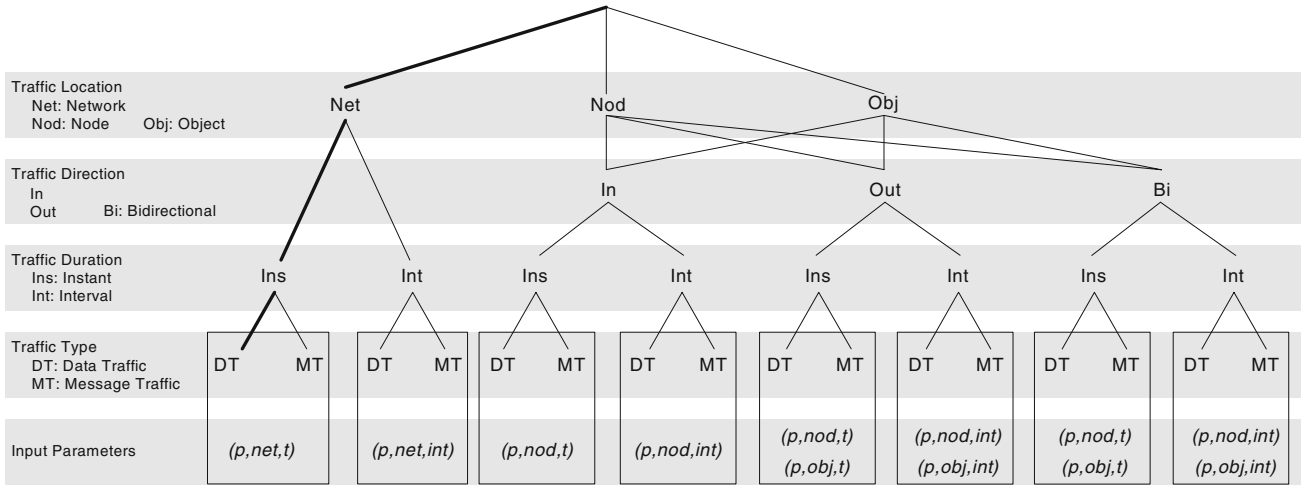


Fig. 9 Naming convention and input parameters of the traffic usage analysis functions

where *dur*(*msg*)returns the time duration of a message: *dur*(*msg*)= *msg.endTime - msg.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. The *getNetworkPath* (*sender node, receiver node*) function (Sect. 5.1.2) returns the path of nodes connecting a sender and a receiver node.

2. *NetInsMT*(*ρ, net, t*) returns the instant message traffic for DTCCFP ρ on network *net* at time t.

$$NetInsMT(\rho, net, t) = |Message|, \text{ where } Message = \left\{ \begin{array}{l} msg_i | msg_i \in \rho \wedge \\ msg_i.start \leq t \leq msg_i.end \wedge \\ net \in getNetworkPath(msg_i.sender.node, \\ \quad msg_i.receiver.node) \end{array} \right\}$$

3. *NetIntDT(ρ, net, int)* returns the data traffic for DTCCFP ρ on network *net* over time interval *int*. *NetIntDT* can be calculated using *NetInsDT*.

$$NetIntDT(\rho, net, int) = \sum_{t=int.start, int.start+1, \ldots, int.end} NetInsDT(\rho, net, t)$$

4. *NetIntMT(ρ, net, int)* returns the message traffic for DTCCFP ρ on network *net* over time interval *int*.

$$NetIntMT(\rho, net, int) = \sum_{t=int.start, int.start+1, \ldots, int.end} NetInsMT(\rho, net, t)$$

5.5.3 Examples

As an example, suppose DTCCFP $\rho = cm_1 cm_2 rm_1 rm_2$ where cm_i and rm_i are call and reply messages, respectively, and are defined in the messages' format (Sect. 5.1.1) as follows:

$$cm_1 = ((o_1, c_1, n_1), (o_2, c_2, n_2), t(), synchCall, \langle (p1 : -), (p2 : -) \rangle, null, 1, 2, \text{'}Call\text{'})$$

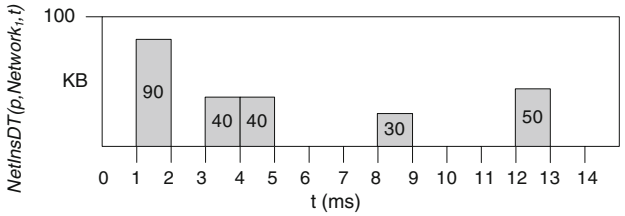
$$cm_2 = ((o_2, c_2, n_2), (o_3, c_3, n_3), u(), synchCall, \langle (p3 : -), (p4 : -) \rangle, null, 3, 5, \text{'}Call\text{'})$$

$$rm_1 = ((o_3, c_3, n_3), (o_2, c_2, n_2), null, null, null, \langle (x = u(), -) \rangle, 8, 9, \text{'}Reply\text{'})$$

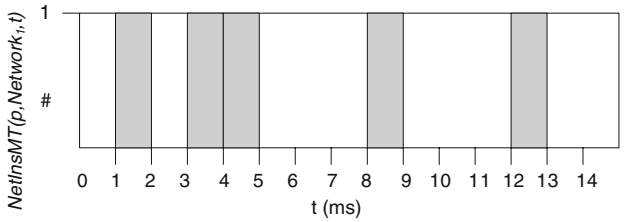
$$rm_2 = ((o_2, c_2, n_2), (o_1, c_1, n_1), null, null, null, \langle (y = t(), -) \rangle, 12, 13, \text{'}Reply\text{'})$$

Recall that a dash (-) indicates that a field can take any arbitrary value (a "don't care" field). Let us assume the system's NDD is the one in Fig. 10. Further assume that the sizes of the four messages of DTCCFP ρ are calculated using the RUM in Eq. 2 yielding the following values: cm_1(90 KB), cm_2(80 KB), rm_1(30 KB), and rm_2 (50 KB). The following traffic functions can then be calculated.

- *NetInsDT(ρ,Network₁,t):*



- *NetInsMT(ρ,Network₁,t):*



- *NetIntDT(ρ,Network₁,(2,9)):*

$$\begin{aligned} NetIntDT(\rho, Network_1, (2,9)) &= \sum_{t=2}^{9} NetInsDT(\rho, Network_1, t) \\ &= NetInsDT(\rho, Network_1, 2) + \cdots + NetInsDT(\rho, Network_1, 9) \\ &= 0 + 40 + 40 + \cdots + 0 = 110KB \end{aligned}$$

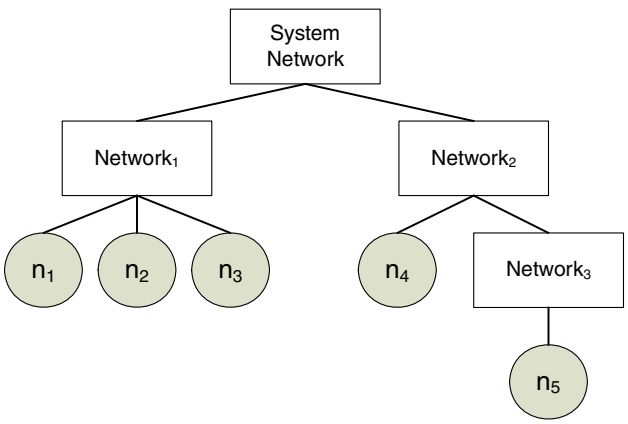


Fig. 10 A simple system topology

It is true that, certain networks, e.g., Controller Area Networks (CAN), place limitations on transmitted data size (due to reliability and predictability constraints). However, let us suppose the case when a designer has specified a message in a UML model whose real data size is going to be more than the maximum size of packets in the underlying dedicated network (e.g., 8 bytes). It is then safe to assume that the large message will be divided into several smaller packets and then transmitted when the system is developed and deployed. We believe the current analysis follows the above scheme without discussing the details of specific networks. Our framework can thus be useful when using networks of the above types.

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 MBPA approach is general as it parameterizes general concepts (RUD and 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 Sect. 5.5 for network usage) for these resource types are not discussed here and should be investigated later based on the presented RUDs and RUMs.

5.6.1 CPU

Our approach for calculating CPU usage using SD messages is as follows. Among call, reply and signal messages, only call and signal messages consume CPU power. Admittedly, a return message also consumes some CPU power (e.g., copying the return results to a stack and returning back the control to the caller). However, as a simplification, we consider the CPU usage entailed by reply messages to be negligible

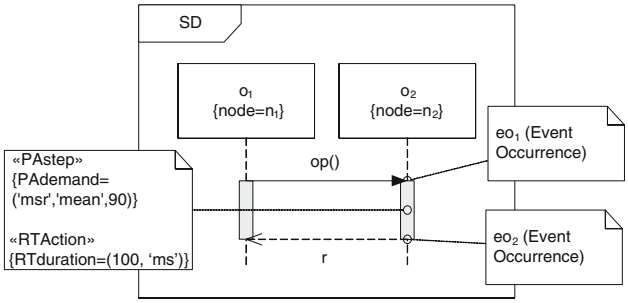


Fig. 11 Modeling CPU usage demand example

compared to the execution of call and signal messages. The practical impacts 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 system 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 [15]. The prediction of processing complexity of an operation is a challenging task in the early design phase. Existing works (e.g., [41,50,51]) have proposed solution to this problem, some of which can be used during design (e.g., [22]). Studying these solutions and finding an adequate one is however outside the scope of this paper.

The UML–SPT [15] discusses ways to model CPU usage in behavioral models. For example, message *op()* in Fig. 11 is annotated with the *PAstep* stereotype and the *PAdemand* tagged-value (PA stands for Performance Analysis).

$$RUD_{CPU} : TCCFP \rightarrow TCCFP$$
$$\forall \rho \in TCCFP : RUD_{CPU}(\rho) = \rho - \{msg \mid msg \in \rho \wedge msg.msgType = 'reply'\}$$

$$\forall msg \in Message : RUM_{CPU}(msg) = CPUUsage(msg)$$

Equation 3 RUD and RUM for CPU resource.

As an example, the *PAdemand* annotation of the execution occurrence for message *op*() in Fig. 11 means that the execution of this message utilizes on average (*mean*) 90% of the CPU (on node n_2) and that this value is *measured* (*msr*).

Based on our approximation for CPU usage by SD messages, we present the RUD and RUM for CPU resource analysis in Eq. 3, where function *CPUUsage* (*message*)returns the processing complexity value associated with the execution occurrence of a message (provided by the *PAdemand* annotation in the UML models).

The RUM_{CPU} of a message equals 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 Fig. 11, 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 eo_1 and finishes at eo_2, the message consumes CPU power only in the time period between these two event occurrences.

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 (Sect. 5.4.1), the CPU usage location denotes the particular CPU on which a message is processed. For example, as shown in Fig. 11, 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 system. 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

We estimate memory usage by SD messages 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 Fig. 12. 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, approximations similar to the ones used in [22]. It should be acknowledged that this is in general a complex task which would need lots of experience and skills from designers. Such information should then be provided by designers in an appropriate way, for example by using specific tagged-values.

[2] "EventOccurrences represents moments in time to which Actions are associated. An EventOccurrence is the basic semantic unit of Interactions. EventOccurrences are ordered along a Lifeline"[21].

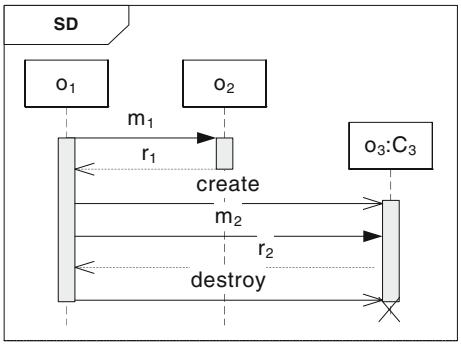


Fig. 12 Memory usage analysis example

Based on our approximation for memory usage by SD messages, we present the RUD and RUM for memory usage in Eq. 4, where function *dataSize* (*class*) returns the data size of a class (Sect. 5.3). 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. As a result, the granularity of the RUA based on the RUD and RUM in Eq. 4 is at the message level. 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. Similar to the *locality* aspect of CPU usage, memory usage analysis should also take *locality* into account in the context of distributed systems.

5.7 Scalability of the approach

We discuss in this section the scalability of our framework w.r.t. the model size of a system. As we discussed in Sect. 2.2, the DTCCFPs of the SDs of a system are generated by our UML 2 sequence diagram control flow analysis. DTCCFPs are then used by the resource usage analysis. Thus, the larger the model size, the more time it will take to perform control flow analysis, derive DTCCFPs, and perform the analysis. The number of SDs (referred to as s), the (average) number of DTCCFPs in SDs (referred to as p), and the number of messages in each DTCCFP (referred to as m) are measures of model size that have an impact on scalability. Based on these measures, the average-case time complexity of our control flow analysis is in O($s.m.p$): during the analysis, we have to traverse all the SDs, all the DTCCFPs of each SD, and all the messages of each DTCCFP. Considering that the number of SDs (s) is usually not a very large number, and that the average number of DTCCFPs per SD (p) is usually small, we can conclude that our control flow analysis is quite scalable.

The RUA is performed using the functions in Sect. 5.5 which basically query the messages of a given DTCCFP and return the value of a formula calculated based on those messages. Thus, the average-case time complexity of our resource usage analysis is in O(m), and is therefore scalable.

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 system. 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 specific execution of the system. Such system execution involves the execution of several scenarios that we intend to derive from SDs. Since the notion of time is paramount in the systems we analyze, those scenario executions have to be scheduled. In other words, we need as input (recall Fig. 5) a Sequence Diagram Schedule (SDS) that is a list of SD scenarios (under the form of TCCFPs) and their start time, i.e., time instants at which to start the execution of the TCCFPs. The designer has therefore to choose time instants for triggering SDs (i.e., the TCCFPs) to be triggered. Helping the designer choosing a SDS is out of the scope of this paper. However, a testing technique such as the one defined in [48] could help. Our objective is therefore to estimate the total load entailed on an entity by triggering the execution of a set of SDs at specific time instants. The motivations for doing this at the design stage are:

1. Analyzing the load on each entity to check whether it conforms to specifications at a stage of development where design decisions can still be easily changed.
2. Finding the entities with highest loads and applying Software Performance Engineering [13] practices to balance load, if needed.
3. Stress testing: Using load forecasting data, stress test cases can be devised to maximize load and evaluate the robustness of the system [52].

Equation 4 RUD and RUM for memory resource.

$$RUD_{Memory} : TCCFP \rightarrow TCCFP$$
$$\forall \rho \in TCCFP : RUD_{Memory}(\rho) = \rho - \{msg \mid msg \in \rho \wedge msg.methodOrSignalName \notin \{create, destroy\}\}$$
$$\forall msg \in Message : RUM_{Memory}(msg) = \begin{cases} +\,dataSize(msg.receiver.class) & ;\text{if } msg.methodOrSignalName = create \\ -\,dataSize(msg.sender.class) & ;\text{if } msg.methodOrSignalName = destroy \end{cases}$$

In our context, MBLF is closely related to RUA because RUA results are used to perform MBLF (recall Fig. 5). While Sect. 5 discussed how to measure resource usage for one TCCFP (we selected network usage and therefore considered DTCCFPs), MBLF determines how much total load is imposed on an entity by triggering several TCCFPs (whose triggering times have been provided). MBLF can be performed on different resource types, and, similarly to the RUA in Sect. 5, we select network traffic in this section as an example resource type, and therefore consider several DTCCFPs.

The concept of Load Forecasting Query (LFQ), to filter the MBLF results for an entity, is described in Sect. 6.1. We provide a high-level overview of our load forecasting approach in Sect. 6.2. We then define a class of load forecasting functions in Sect. 6.3 which are classified based on four load attributes (Sect. 6.1). The functions are similar to the traffic usage analysis functions (Sect. 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 instant (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* (*i.e., by all SDs* in a SDS) *number of requests towards object o at time instant t?* Note that there is a difference between a LFQ and a RUQ (Sect. 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 TCCFP, while MBLF performs an analysis for a set of TCCFPs.

6.2 Load forecasting approach

As already defined, a Sequence Diagrams Schedule (SDS) is a set of specific TCCFPs from SDs (one TCCFP per SD) and their start time. A formal definition of a SDS is as follows. Assuming that a system has nTCCFPs (ρ_1, …, ρ_n), a SDS is a schedule of a selected set of TCCFPs in the form of: $\langle(\rho_1,\alpha_1\rho_1)$, $(\rho_1,\alpha_2\rho_1)$,…, $(\rho_1,\alpha_i\rho_1)$, $(\rho_2,\alpha_1\rho_2)$, …,$(\rho_m,\alpha_1\rho_m)$,$(\rho_m,\alpha_2\rho_m)$,…,$(\rho_m,\alpha_j\rho_m)\rangle$, where the value of m is independent of the value of n and each entry of the sequence is a tuple $(\rho, \alpha\rho)$ such that $\alpha\rho$ is the start time of TCCFP ρ, i.e., the time to trigger ρ. Note that zero, one or more different schedules of a TCCFP can appear in a SDS. For example, in a system which has three TCCFPs (ρ_1, ρ_2, ρ_3), sds=$\langle(\rho_1,2ms)$, $(\rho_1,5ms)$, $(\rho_3,1ms)\rangle$ includes two schedule of ρ_1 (at $2ms$ and 5 ms), one schedule of ρ_3 (at $1ms$) and no schedule of ρ_2.

A high-level overview of our load forecasting approach is illustrated using the example in Fig. 13, which shows how the example LFQ in Sect. 6.1 can be answered. We assume that the system has three SDs, one DTCCFP has been selected for each SD (namely ρ_1, ρ_2 and ρ_3) to form a SDS. In order to forecast load, the RUA in Sect. 5 is performed first (Fig. 13a). This results in a curve for each DTCCFP that shows the resource usage of that DTCCFP over time (recall the naming convention of Fig. 9). For example, ObjInInsMT(ρ_1,o,t) shows the message traffic entailed by DTCCFP ρ_1 at each instant t towards object o. Using the start times from the SDS, MBLF schedules RUA results of DTCCFPs (Fig. 13b) and then calculates the amount of load at a given time instant by adding up the usage values of all the selected DTCCFPs (shown by a dashed line in Fig. 13c).

6.3 Load forecasting functions

The naming convention of the functions is given in Sect. 6.3.1. For brevity, only formal definitions of load functions for network load location are presented in Sect. 6.3.2. The functions for node and object load locations are derived in a similar fashion and are presented in [48].

6.3.1 Naming convention

A tree structure denoting the convention for naming load forecasting functions is shown in Fig. 14. 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 Fig. 14 specifies the input parameters of each load forecasting function. For example, consider the path specified by the bold line in Fig. 14, i.e., function *NetInsDL*. According to the bottom layer, the input parameter of this function is (*sds, net, t*). This function returns the instant (*Ins*) data load (*DL*) value of a given SDS (*sds*) for a given network (*net*) at a given time (*t*). Input parameters with *int* in the bottom layer of Fig. 14 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.

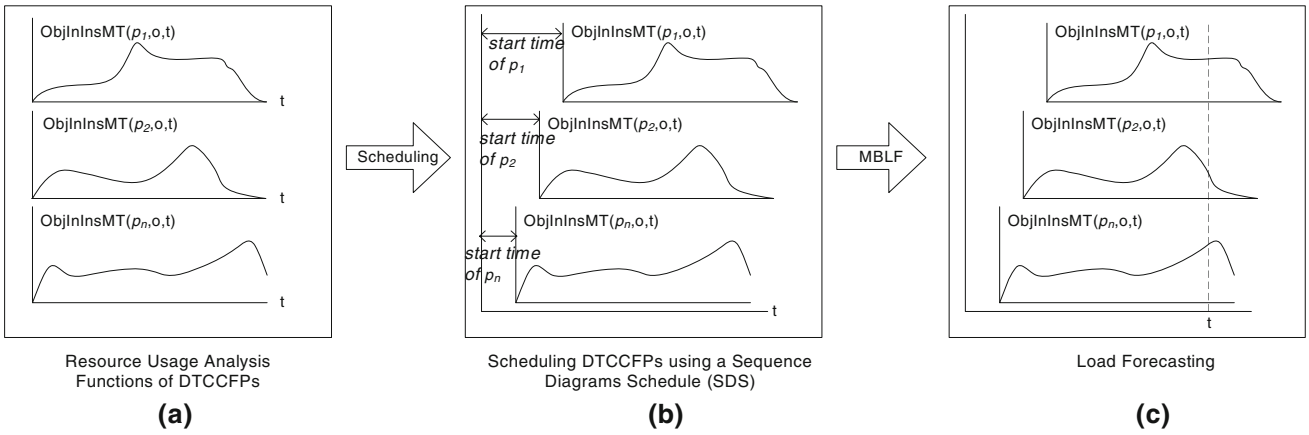


Fig. 13 Load forecasting approach

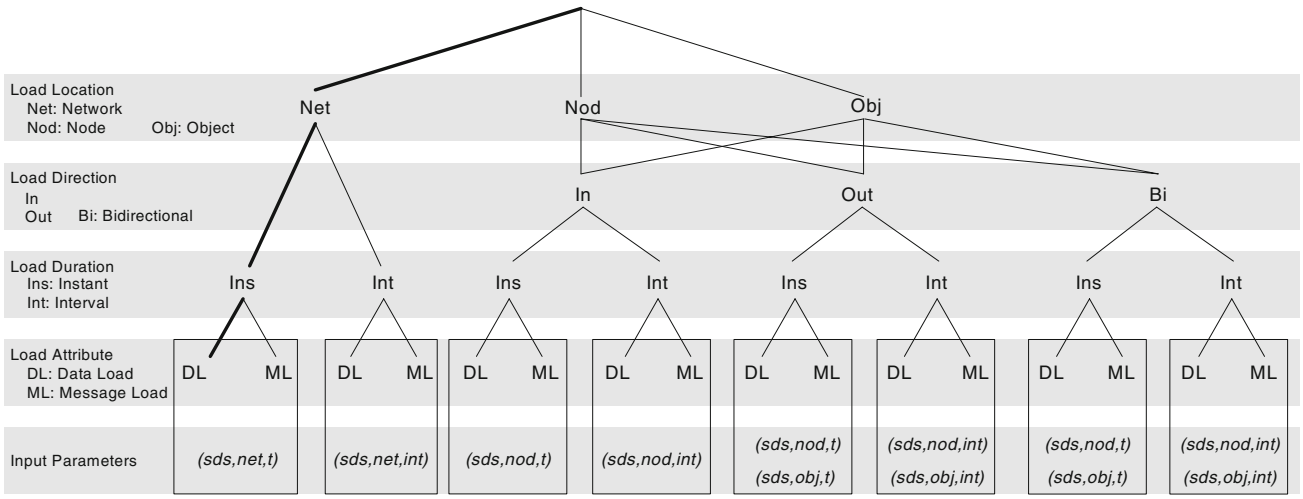


Fig. 14 Naming convention and input parameters of load forecasting functions

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 [48].

1. *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).

$$NetInsDL(sds, net, t) = \sum_{\forall(\rho,\alpha\rho)\in sds} NetInsDT(\rho, net, t-\alpha\rho)$$

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

$$NetInsML(sds, net, t) = \sum_{\forall(\rho,\alpha\rho)\in sds} NetInsMT(\rho, net, t-\alpha\rho)$$

3. *NetIntDL*(*sds, net, int*) returns the interval data load for SDS *sds* on network *net*over time interval *int*. *NetIntDL* is calculated using *NetInsDL*.

$$NetIntDL(sds, net, int) = \sum_{t=int.start, int.start+1, \cdots, int.end} NetInsDL\ (sds, net, t)$$

4. *NetIntML*(*sds, net, int*) returns the interval message load for SDS *sds* on network *net*over time interval *int*.

NetIntML is calculated using *NetInsML*.

$$NetIntML(sds, net, int) = \sum_{t=int.start, int.start+1, \cdots, int.end} NetInsML\ (sds, net, t)$$

7 Reusing MBPA principles for additional applications

We describe in this section how the prediction of resource usage and load enables designers to perform two additional, important activities: Detecting resource overuse (Sect. 7.1); Predicting resource utilization (Sect. 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 reaches or exceeds C_R ($U_R \geq C_R$). 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 Sects. 5 and 6, Eq. 5 can be used to detect network traffic overuse based on load analysis information. The function $DetectNetworkTrafficOveruse(sds, net, C_{net})$ 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 discrete time instance (t in Eq. 5).

Note that we use the *data* load forecasting function *NetIns DL* in Eq. 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., queuing theory [53]).

7.2 Predicting resource utilization

Informally, the resource utilization*ratio* (or simply resource utilization) is defined as the amount of usage of a resource divided by its capacity/speed/etc. (depending on resource type) [54], e.g., if 500 MB of a 2 GB RAM module is allocated in a time instant, we can say that the resource utilization of this module is 25% at that particular time instant.

It is possible to predict the utilization of a resource from our proposed analysis by simply dividing the RUA metric by the capacity of the resource, e.g., the first two formulas in Eq. 6 can be used to predict the utilization (ratio) of a network and a memory module with the capacity of C_{net} and C_{mem}, respectively, at a time instant t by executing a given Sequence Diagram Schedule (SDS). The $MemInsDT(\rho, mem, t)$ memory usage analysis function can be easily calculated by using $RUM_{Memory}(msg)$ from Eq. 4 similar to what was done in Sect. 5.5 for the network resource type. Calculating the utilization of a resource over a time period is also possible using our proposed analysis: the 3rd formula of Eq. 6 can estimate the utilization ratio of a network over a time period using instant utilization values. System-wide resource utilization prediction is also possible by considering all possible SDSs in a system (4th formula in Eq. 6).

8 Case study

We apply our model-based resource usage analysis (MBRUA) approach to a distributed system to demonstrate its feasibility and to illustrate the variety of MBRUA activities that can be performed. The system and its UML model are described in Sect. 8.1. We then report on two applications of our MBRUA approach: (1) Predicting resource usage (Sect. 8.2), and (2) Detecting resource overuse (Sect. 8.3). Final discussions and limitations of our framework are summarized in Sect. 8.4.

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 [55]). The system is referred to as *SCAPS* (*a SCAda-based Power System*) [52]. 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 (TC) 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 complete UML

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

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

Equation 6 Predicting Resource Utilization.

$$NetworkUtilizationIns(sds,net,t) = \frac{\sum_{\forall \rho \in sds} NetInsDT(\rho,net,t)}{C_{net}}$$

$$MemoryUtilizationIns(sds,mem,t) = \frac{\sum_{\forall \rho \in sds} MemInsDT(\rho,mem,t)}{C_{mem}}$$

$$NetworkUtilizationInt(sds,net,\text{int}) = \sum_{t=\text{int}.start,\text{int}.start+1,\cdots,\text{int}.end} NetworkUtilizationIns(sds,net,t)$$

$$SystemWideNetworkUtilizationIns(net,t) = \sum_{\forall sds} NetworkUtilization(sds,net,t)$$

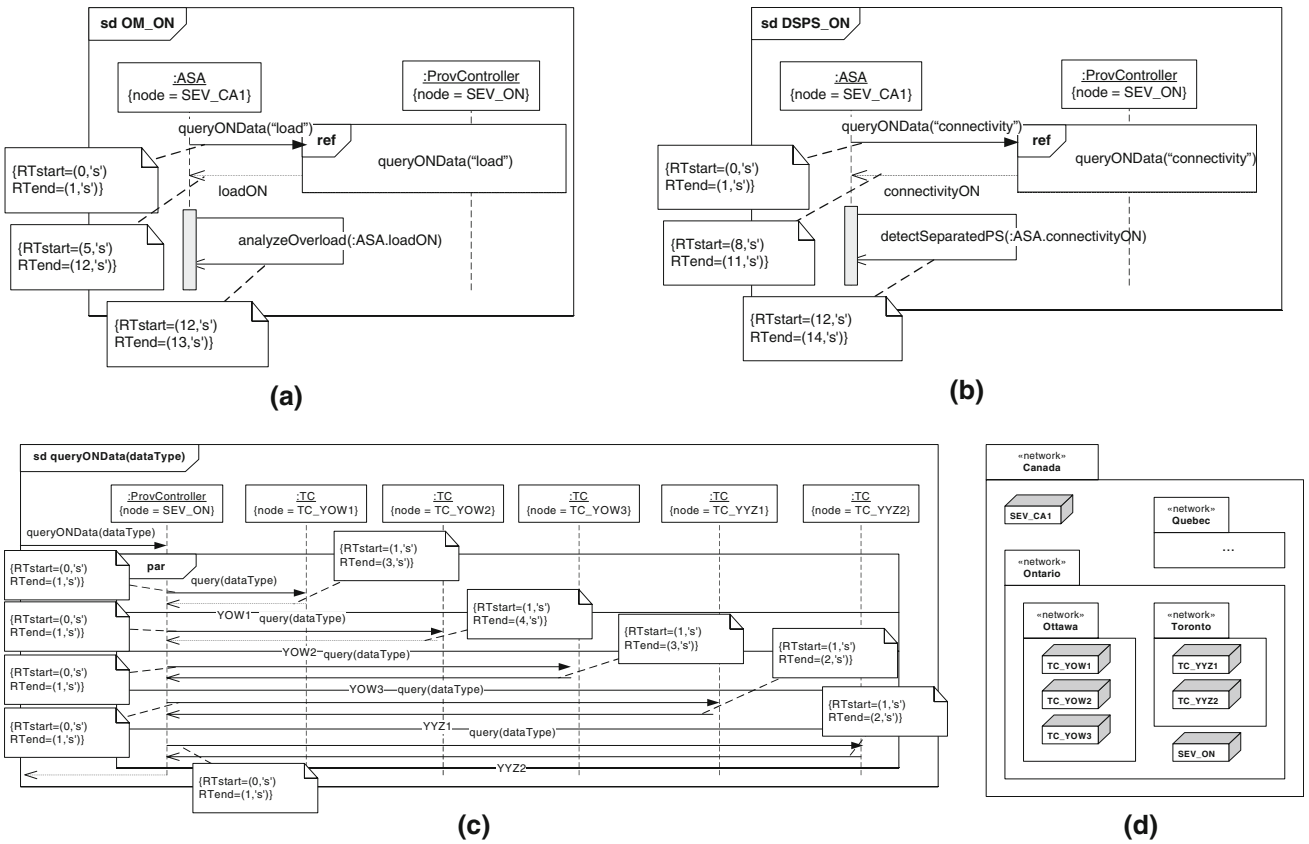


Fig. 15 Parts of the SCAPS design UML model

design model of SCAPS is presented in [56], and we only present in Fig. 15 the parts of the model used in the two analyses of Sects. 8.2 and 8.3.

The SD in Fig. 15a, *OM_ON*, corresponds to the overload monitoring (OM) control of the province of Ontario (ON) in SCAPS. The SD in Fig. 15b, DSPS_ON, corresponds to a use case which is responsible for Detection of Separated Power System (DSPS) for the province of Ontario (ON). This SD fetches grid connectivity data from the ON provincial controller and checks whether there is any separated power system by using *detectSeparatedPS()*. By using the interaction occurrence construct of UML 2.0 SDs, the SD *OM_ON* references the SD in Fig. 15c 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 object in those SDs.

Figure 15d is the NDD, as defined in Sect. 5.1.2, of the system. We do not show the SCAPS class diagram here due to its large size. Parameter *dataType* in Fig. 15c, 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 Eq. 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., [55]). Furthermore as we discuss in detail in [56], the real-time constraints in Fig. 15 are realistic estimates of message duration times used in SCADA power systems. For example, the *analyzeOverload* message should be completed in less than a second or a RT fault (resulting in a catastrophic result) will occur in the system.

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 Fig. 15 and considering network as an example resource type. The predicted network usage 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*tool [57] 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 Fig. 16. To ease discussion, CCFG nodes have been labeled A_x. 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 ρ_{OM_ON} in Fig. 17. Using RUD (Sect. 5.2), the RUA process converts ρ_{OM_ON} to a DTCCFP, as shown in Fig. 17. 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 $NetInsDT(\rho_{OM\_ON}$, *"Canada", t*) is computed for different values of t in the time interval [0 ms, 13 ms (duration of ρ_{OM_ON})], as depicted in Fig. 18. Referring to the messages in the SD *OM_ON*and the SCAPS deployment structure (Fig. 15d), only the reply message from an object of type *ProvController* to an object of type *ASA*, i.e., message A_{12} in Fig. 16, is sent over the national network (*Canada*) and is considered in the RUA. The data size of this return message is *20MB*: $5(numberofTCsin\ Ontario) \times 4\,MB\ (data\ size\ of\ the\ LoadStatus\ class)$. Since the duration of this message is 7 s (12-5 as specified in Fig. 15a), 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 usage with the measured network usage entailed by executing CCFP ρ_{OM_ON}. The runtime resource usage values were measured by executing CCFP ρ_{OM_ON} and recording the amount of network traffic using the monitoring tool [57] in each time instant. Note that we have the same time precision (1 s) in the monitoring tool, the time annotations in the SDs, and our formulas. We used the approximation [58] 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_CA1* (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

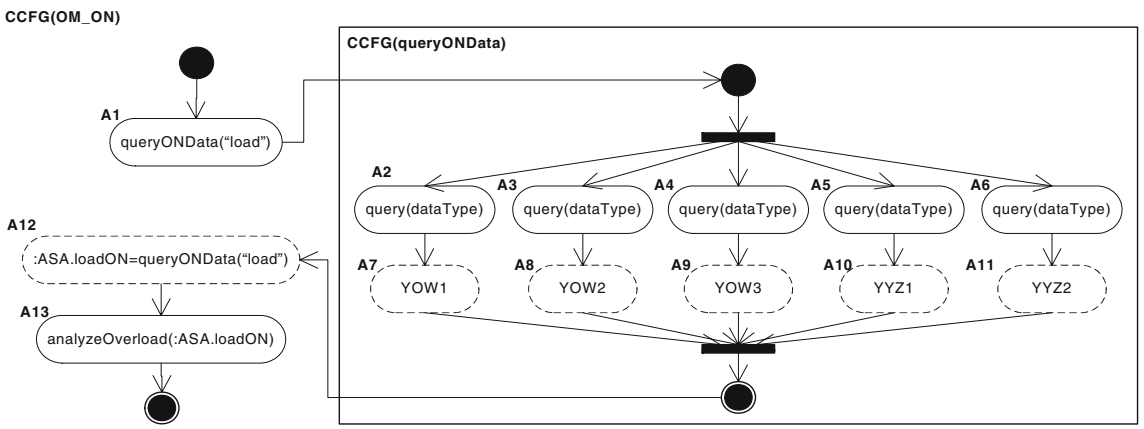


Fig. 16 CCFG of SD *OM_ON*

$$\rho_{OM\_ON} = A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}A_{13} \Rightarrow DTCCFP(\rho_{OM\_ON}) = A_1 \begin{pmatrix} A_2A_7 \\ A_3A_8 \\ A_4A_9 \\ A_5A_{10} \\ A_6A_{11} \end{pmatrix} A_{12}$$

Fig. 17 The only TCCFP and DTCCFP of the CCFG in 16

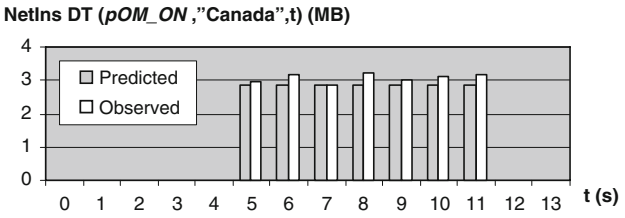


Fig. 18 An example of predicted and observed resource usage analysis

observability. The later two properties are important for the case study system since, in our context, they facilitate traffic measurement in our experiments. The system was designed in a way that TCs operation could be easily controlled (high controllability) and their behavior and resource usage pattern could be easily observed (high observability). The actual network deployment of our case study is illustrated in Fig. 19: three intranets in our institution (servers, SCE department, and squall.sce.carleton.ca) were chosen to simulate the network architecture of SCAPS.

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 Sect. 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 Figs. 15d and 19, 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.

The Operating System (OS) and hardware configurations of the machines used are detailed in Table 2. All the network connections had a bandwidth of 100 Mbps. The monitoring tool [57] 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 monitoring tool 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 (Fig. 19) were not used in the analysis.

Note that although the monitoring tool allows us to let us extract the traffic values resulting from a network application among several running applications on a machine, 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 systems, and to minimize any network-related effect by other applications on our case study.

8.2.3 Comparison results

The average values of observed data traffic in each time instant over 10 runs are depicted in Fig. 18. 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 Sect. 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 Sect. 5.3. To predict worst case scenarios for network traffic usage (upper bound values), one might estimate the upper bound value of the overhead data added by the lower layers of the OSI for a given system and add it to the RUM values estimated using our approach.

As we can see in Fig. 18, the difference is small in our case study, providing evidence that our estimates of data traffic

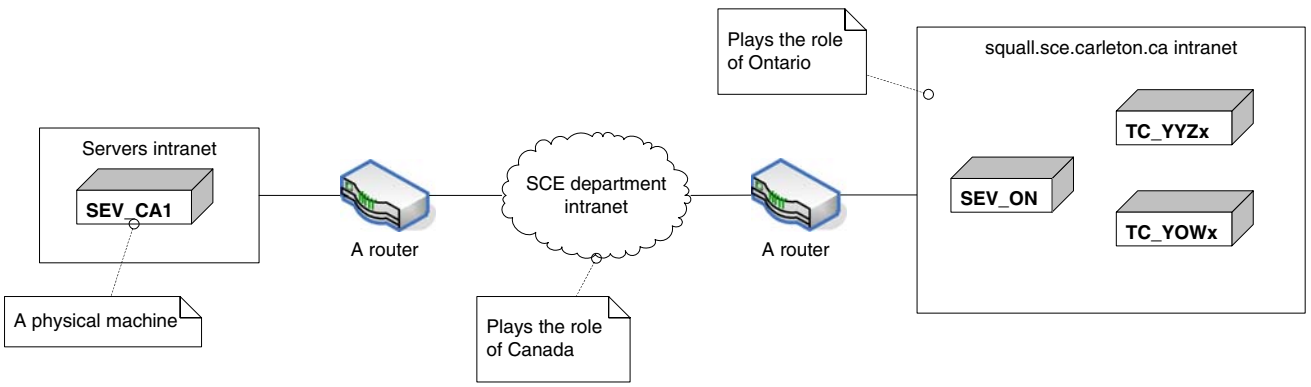


Fig. 19 The actual network deployment of our case study

Table 2 OS and hardware configurations of the machines used in our case study

Machine hosting	OS	CPU	RAM (GB)	Network card
SEV_CA1	Windows 2000	2.8 GHz Intel Xeon	2	Intel PRO/1000 XT
SEV_ON	Windows XP	2.8 GHz Intel Pentium 4	2	3COM Fast Ethernet Controller
TC_YOWx	Windows 2000	863 MHz Intel Pentium III	1	3COM Fast Ethernet Controller
TC_YYZx	Windows 2000	863 MHz Intel Pentium III	1	3COM Fast Ethernet Controller

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.

Recall, however, that our approach takes in input a number of estimates, e.g., regarding properties of messages in SDs such as *RTstart*, *RTend* and data sizes. In our case study, because the implementation was available, those properties could be precisely measured and this lead to an accurate prediction of data traffic. We should, however, consider the observed accuracy to be an upper bound as our approach is meant to be applicable during the design stage.

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 consisting of TCCFPs from two SDs: *OM_ON*, Fig. 15a, and *DSPS_ON* (Detection of Separated Power System), Fig. 15b.

Since both SDs *OM_ON* and *DSPS_ON* are initiated by ASA and both query specific provincial power grid data (load and connectivity, respectively), the entailed traffic goes through the national network (*Canada*) (Fig. 15d). As a given SDS, we consider $sds = \langle(\rho_{OM\_ON}, 0ms), (\rho_{DSPS\_ON}, 0ms)\rangle$, 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 Sect. 7.1, we have to evaluate function *DetectNetworkTrafficOveruse* (sds, "$Canada$", $C_{\text{“Canada”}}$), where $C_{\text{“}Canada\text{”}} = 100$ Mbps (mega bits per second)=12.5 MBps (mega bytes per second).

As shown in Eq. 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 Sect. 8.2, which corresponded to the only CCFP (ρ_{OM_ON}) of SD *OM_ON*. By using a similar procedure, we calculated the *NetInsDT*(ρ_{DSPS_ON},*"Canada", t)* values as shown in Fig. 20b. In order to calculate the predicted resource usage values of ρ_{DSPS_ON}, in Fig. 20b, we assumed that the size of the Ontario grid connectivity data (*connectivityON* in Fig. 15b is 50 MB (a realistic estimation based on the SCADA literature [55]). Therefore, the resource usage value per time unit will be $50/3 \approx 16.6$ MB. *3 ms* here is the duration of the return message from the *queryONData* interaction occurrence in Fig. 15b.

Figure 20a corresponds to *NetInsDT*((ρ_{OM_ON}, *"Canada", t)* and uses the predicted values from Fig. 18 (i.e., 2.85 MB/s). Using the definitions in Sect. 6 and based on the definition of *sds* (above), the two *NetInsDT* functions in Fig. 20a and b yield the *NetInsDL* function values shown in Fig. 20c, 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 Eq. 5 with the values in this section (network name and its capacity), we can write:

$$
\begin{aligned}
&DetectNetworkTrafficOveruse(sds, \text{“}Canada\text{”}, \\
&\quad 12\,MB) \\
&= \begin{cases} true & ; \text{if } \exists t | NetInsDL(sds, \text{“}Canada\text{”}, t) \\ & \quad > 12.5\,MB \\ false & ; \text{otherwise} \end{cases} \\
&= \text{true because for } t \in \{8, 9, 10\} : NetInsDL(sds, \\
&\quad \text{“}Canada\text{”}, t) > 12.5\,MB
\end{aligned}
$$

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 $C_{\text{“}Canada\text{”}} = 12.5$ MB (the capacity of network *Canada*). Such an analysis is shown graphically in Fig. 20c, 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 Sect. 8.2. A comparison between the predicted values (from Fig. 20c) and the observed network traffic load is reported in Fig. 20d. Note the discrepancy (during time interval [8,12]) between

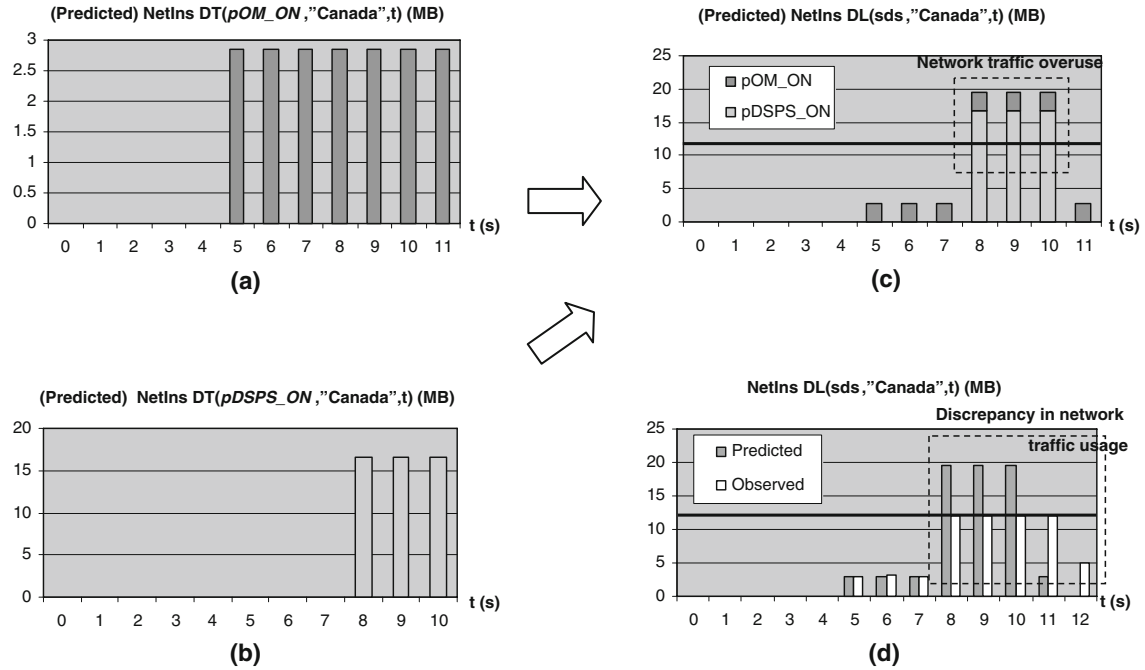


Fig. 20 **a–c** Calculating *NetInsDL(sds, "Canada", t)*. **d** Comparison between predicted and observed network traffic load when an overuse occurs

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 Fig. 20d, 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 s, respectively). In other words, we have found a network traffic overuse situation which has led to a constraint violation 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.

8.4 Discussions and limitations

Let us recall from Sect. 2.1 the challenge of estimating timing information in SD messages. Our case study was performed over a local-area network and in controlled conditions (Sect. 8.2) to alleviate this problem. However, conducting the experiment over a wide-area network, where the network delays are much more unpredictable and variable, would probably make the results less predictable.

We would also like to note that, as discussed in Sect. 2.1, even if message timing information is not predictable to a reasonable level of precision, our resource usage and load prediction framework can still be useful. For example, according to another experiment we conducted regarding a UML-driven stress test technique [42] in the presence of time uncertainty of SD messages, the framework presented in this paper was useful to increase the chances of detecting RT faults.

As we discuss throughout this paper, it is to be expected that approximations, simplifications, and assumptions would be required (e.g., timing information of messages) when performing prediction that early in the design process. Our goal in this work here is to provide designers with a tool to make predictions based on what is known at design time. There are certainly many things that are not known yet regarding the implementation and execution that will affect resource usage and load. But having a tool allowing designers/architects to make early predictions based what they know is a way to support early design decisions and identify potential bottlenecks that will have to be carefully watched in the remainder of development.

9 Conclusions

This paper presents a quantitative framework for the early, model-based prediction of resource usage and load in DRTS 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 DRTS by offering

a systematic approach, based on plausible and standard early design models, to predict system behavior (usage and load of network traffic as an example resource type in this work) in each time instant during its execution. The results of a case study on an actual DRTS have shown that the approach is promising as it yields reasonably accurate results (estimates are on average 6% below actual values). However, it should also be noted that due to discrepancies between real and predicted timing behavior of a system, in practice one might produce less accurate estimations of resource usage and load in real situations, thus making any prediction framework like the one we propose less accurate.

Based on our model-based resource usage analysis (MBRUA) principles, we furthermore develop automated techniques to perform two important activities in the context of DRTSs: (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 DRTSs and assess its effectiveness in improving the predictability of DRTSs; (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.

To perform more realistic RUA, we also plan to use Probability Density Functions (PDF), e.g., the beginning and the end of a time interval, as start and end time measures of messages. It is expected that this would lead to increased complexity in our formulas without changing the fundamental ideas we convey in this paper.

Acknowledgments This work was in part supported by Siemens Corporate Research, Princeton, NJ and a Canada research chair. Vahid Garousi was further supported by the Discovery Grant no. 341511-07 from the Natural Sciences and Engineering Research Council of Canada (NSERC), and also by the Alberta Ingenuity New Faculty Award no. 200600673.

References

1. Tsai, J.J.P., Bi, Y., Yang, S.J.H., Smith, R.A.W.: Distributed Real-time Systems: Monitoring, Visualization, Debugging, and Analysis. Wiley, London (1996)
2. Feiler, P., Lewis, B., Vestal, S.: Improving Predictability in Embedded Real-time Systems. Technical Report CMU/SEI-2000-SR-011, Carnegie Mellon Software Engineering Institute (2000)
3. Buttazzo, G., Lipari, G., Abeni, L., Caccamo, M.: Soft Real-Time Systems: Predictability vs. Efficiency. Springer, Heidelberg (2005)
4. Yau, S.S., Zhou, X.: Schedulability in Model-based Software Development for Distributed Real-Time Systems. In: Proc. of Int. Workshop on Object-Oriented Real-Time Dependable Systems, pp. 45–52 (2002)
5. Igarashi, A., Kobayashi, N.: Resource Usage Analysis. In: Proc. of Symposium on Principles of Programming Languages, pp. 331–342 (2002)
6. Marriott, K., Stuckey, P.J., Sulzmann, M.: Resource Usage Verification. In: Proc. of Asian Symposium on Programming Languages and Systems, pp. 212–229, (2003)
7. Cachera, D., Jensen, T., Pichardie, D., Schneider, G.: Certified Memory Usage Analysis. In: Proc. of Formal Methods Conf., pp. 91–106, (2005)
8. Dinda, P., O'Hallaron, D.: An Extensible Toolkit for Resource Prediction In Distributed Systems, Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University (1999)
9. Dinda, P.A., O'Hallaron, D.R.: An Evaluation of Linear Models for Host Load Prediction. In: Proc. of Int. Symp. on High Performance Distributed Computing, pp. 87–96 (1999)
10. Andreolini, M., Casolari, S.: Load Prediction Models in Web-based Systems. In: Proc. of Int. Conf. on Performance Evaluation Methodolgies and Tools, pp. 27–36 (2006)
11. Menascé, D.A., Almeida, V.A.F.: Capacity Planning for Web Services: Metrics, Models, and Methods. Prentice Hall, Englewood Cliffs (2001)
12. Jones, C.: Software change management. IEEE Comput **29**(2), 80–82 (1996)
13. Smith, C.U., Williams, L.G.: Software performance engineering. In: Marciniak, J.J. (ed) Encyclopedia of Software Engineering, 2nd edn. Wiley, London (2002)
14. Object Management Group (OMG) UML 2.1.1 Superstructure Specification (2007)
15. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable petri-net models. In: Proc. of Int. Workshop on Software and Performance, pp. 35–45 (2002)
16. Pasaje, J.L.M., Harbour, M.G., Drake, J.M.: MAST real-time view: a graphic UML tool for modeling object-oriented real-time systems. In: Proc. of Real-Time Systems Symposium, pp. 245–256 (2001)
17. Balsamo, S., DiMarco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development. IEEE Trans. Softw. Eng. **30**(5), 295–310 (2004)
18. Cortellessa, V., D'Ambrogio, A., Iazeolla, G.: Automatic derivation of software performance models from CASE documents. Perform. Eval. **45**(2–3), 81–105 (2001)
19. Mirandola, R., Cortellessa, V.: UML based performance modeling of distributed systems. In: Proc. of the Unified Modeling Language (UML) Conference, pp. 178–193 (2000)
20. Smith, C.U., Williams, L.G.: Performance Solutions. Addison-Wesley, Reading (2002)
21. Object Management Group (OMG) UML 2.0 Superstructure Specification (2005)
22. Object Management Group (OMG) UML Profile for Schedulability, Performance, and Time (v1.1) (2005)
23. Garousi, V., Briand, L., Labiche, Y.: Control flow analysis of UML 2.0 sequence diagrams. In: Proc. of the European Conf. on Model Driven Architecture-Foundations and Applications, LNCS 3748, pp. 160–174 (2005)

24. Paltor, I.P., Lilius, J.: Digital sound recorder: a case study on designing embedded systems using the UML notation, Turku Centre for Computer Science, Finland TUCS Technical Report No. 234 (1999)
25. Douglass, B.: Doing Hard Time, Developing Real-Time Systems with UML Objects, Frameworks, and Patterns. Addison-Wesley, Reading (1999)
26. Herzberg, D.: UML-RT as a candidate for modeling embedded real-time systems in the telecommunication domain. In: Proc. of Int. Conf. on the Unified Modeling Language, pp. 331–338 (1999)
27. Kabous, L., Neber, W.: Modeling hard real time systems with UML: the OOHARTS approach. In: Proc. of Int. Conf. on the Unified Modeling Language, pp. 339–355 (1999)
28. Lanusse, A., Gerard, S., Terrier, F.: Real-rime modeling with UML: the ACCORD approach. In: Proc. of Int. Conf. on the Unified Modeling Language, pp. 319–335 (1998)
29. Object Management Group (OMG) UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), Version 1.0 (Finalization Underway). http://www.omg.org/cgi-bin/doc?ptc/2007-08-04 (2007)
30. Hakansson, J., Mokrushin, L., Pettersson, P., Yi, W.: An analysis tool for UML models with SPT annotations. In: Int. Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems (2004)
31. Woodside, C.M., Petriu, D.C.: Capabilities of the UML profile for schedulability performance and time (SPT). In: Workshop on the Usage of the UML profile for Scheduling, Performance and Time (2004)
32. Petriu, D.C.: Performance analysis based on the UML SPT profile. Tutorial given at Int. Conf. on Quantitative Evaluation of Systems (2004)
33. Douglass, B.P.: Rhapsody 5.0: Breakthroughs in Software and Systems Engineering, I-Logix Corp. whitepaper (2003)
34. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering: Using UML, Patterns, and Java, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
35. Haugen, Ø.: From MSC-2000 to UML 2.0—the future of sequence diagrams. In: Proc. of Int. System Design Languages (SDL) Forum, pp. 38–51 (2001)
36. Ben-Abdallah, H., Leue, S.: Timing constraints in message sequence chart specifications. In: Proc. of Int. Conf. on Formal Description Techniques for Distributed Systems and Communication Protocols, pp. 91–106 (1997)
37. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Formal Methods Syst. Des. **19**(1), 45–80 (2001)
38. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems **40**(2–3), 117–134 (1994)
39. Puschner, P.P., Nossal, R.: Testing the results of static worst-case execution-time analysis. In: Proc. of IEEE Real-Time Systems Symp., pp. 134–143 (1998)
40. Thane, H.: Monitoring, Testing and Debugging of Distributed Real-Time Systems, PhD Thesis, Royal Institute of Technology (2000)
41. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley, Reading (2000)
42. Garousi, V.: Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty. In: Proc. IEEE Int. Conf. on Software Testing, Verification and Validation (to appear) (2008)
43. Object Management Group (OMG) OCL 2.0 Specification (2005)
44. Muchnick, S.: Advanced Compiler Design and Implementation, First ed. Morgan Kaufmann, Los Altos (1997)
45. Garousi, V., Briand, L., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams, Technical Report SCE-05-09, Carleton University, http://squall.sce.carleton.ca/pubs/tech_report/TR_SCE-05-09.pdf (2005)
46. Byhlin, S., Ermedahl, A., Gustafsson, J., Lisper, B.: Applying static WCET analysis to automotive communication software. In: Proc. of Euromicro Conf. on Real-Time Systems, pp. 249–258 (2005)
47. Honeywell Inc., MetaH Product Information, http://www.htc.honeywell.com/metah (1998)
48. Garousi, V., Briand, L., Labiche, Y.: A Quantitative Framework for Predicting Resource Usage and Load in Distributed Real-Time Systems based on UML Models, Technical Report SCE-06-05, Carleton University (2006)
49. Garousi, V.: Traffic-aware Stress Testing of Distributed Systems based on UML Models using Genetic Algorithms, PhD Thesis, Carleton University (2006)
50. Avritzer, A., Ros, J.P., Weyuker, E.J.: Estimating the CPU utilization of a rule-based system. ACM SIGSOFT Softw. Eng. Notes **29**(1), 1–12 (2004)
51. Wang, Y.F., Hsu, M.H., Chuang, Y.L.: Predicting CPU Utilization by Fuzzy Stochastic Prediction. Comput. Inform. **20**(1), 67–76 (2001)
52. Garousi, V., Briand, L., Labiche, Y.: Traffic-aware stress testing of distributed systems based on UML models. In: Proceedings of International Conference on Software Engineering, pp. 391–400 (2006)
53. Ganesh, A., O'Connell, N., Wischik, D.: Big Queues. Springer , Heidelberg (2004)
54. Rak, J.: Priority-enabled optimization of resource utilization in fault-tolerant optical transport networks. In: Proc. of Int. Conf. on High Performance Computing and Communications, pp. 863–873 (2006)
55. Daneels, A., Salter, W.: What is SCADA? In: Proc. of Int. Conf. on Accelerator and Large Experimental Physics Control Systems, Trieste, pp. 339–343 (1999)
56. Garousi, V., Briand, L., Labiche, Y.: Traffic-aware Stress Testing of Distributed Systems based on UML Models, Technical Report SCE-05-13, Carleton University (2005)
57. Nico Cuppen Software, Network Traffic Monitor, http://www.nicocuppen.com (2005)
58. Caceres, R., Duffield, N., Feldmann, A.: Measurement and analysis of IP network usage and behaviour. IEEE Commun. Mag. **38**(5), 144–151 (2000)

Author Biographies

Vahid Garousi is an assistant professor of software engineering and an Alberta Ingenuity new faculty at the University of Calgary, leading the software quality engineering research group. He won an Alberta Ingenuity new faculty award in June 2007. Vahid received a PhD in software engineering from Carleton University in 2006. His MSc degree was in electrical and computer engineering from the University of Waterloo in 2003. He earned his software engineering undergraduate degree from Sharif University of Technology (the first rank engineering school in Iran) in 2000. Vahid has been on the program or organization committees of many international, IEEE and ACM conferences. He is a member of the IEEE and the IEEE Computer Society,

and is a registered professional engineer in Canada. His research interests include: model-driven development, testing and quality assurance, and applications of optimization and evolutionary computation to software testing.

Lionel C. Briand is a professor of software engineering at the Simula Research laboratory and University of Oslo, leading the project on software verification and validation. Before that, he was on the faculty of the department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was full professor and held the Canada Research Chair in Software Quality Engineering. He has also been the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland, USA. He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the coeditor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification, and Reliability (Wiley). He was on the board of IEEE Transactions on Software Engineering from 2000 to 2004. His research interests include: model-driven development, testing and quality assurance, and empirical software engineering.

Yvan Labiche is an Associate Professor with the Department of Systems and Computer Engineering at Carleton University, Ottawa, Canada. Yvan received the BSc degree in computer system engineering from the Graduate School of Engineering: CUST (Centre Universitaire des Science et Techniques, Clermont-Ferrand), France. He received the master's degree in fundamental computer science and production systems in 1995 (Université Blaise Pascal, Clermont-Ferrand, France). While completing his PhD degree in software engineering, received in 2000 from LAAS/CNRS in Toulouse, France, he worked with Aerospatiale Matra Airbus (now EADS Airbus) on the definition of testing strategies for safety-critical, on-board software, developed using object-oriented technologies. His research interests include: object-oriented analysis and design, software testing in the context of object-oriented development, and technology evaluation. He is a member of the IEEE.