Transforming Use Case Maps to the Core Scenario Model Representation

Yong Xiang Zeng

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Under the auspices of the Ottawa-Carleton Institute for Computer Science

University of Ottawa
Ottawa, Ontario, Canada
June 2005

Abstract

Scenarios describe system functionalities and, when supplemented with performance annotations, provide a basis for performance analysis. This thesis presents a Core Scenario Model (CSM), which includes common scenario information in the Use Case Maps (UCM) notation and in UML 2.0 activity diagrams and interaction diagrams. The intent is to enable automatic transformations from a given scenario model to CSM, and then from a CSM model to performance models, hence enabling performance analysis early in the development process. CSM captures scenarios, performance, and deployment information, and is flexible enough to support multiple input scenario notations and output performance languages.

The thesis also includes a transformation from UCM models to CSM. Such automatic transformation enables the quick generation of performance models, ready to be analyzed. The results from performance analysis can be traced back to improve design decisions and any changes in design can be updated in performance models quickly through re-generation.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor Daniel Amyot. This thesis would never have been finished without his guidance and support. I especially thank him for his detail reviews and for providing feedback on every chapter. His suggestions have made positive contributions to every chapter of this thesis.

Parts of this thesis benefited from the collaboration work in PUMA and RDA groups. At Carleton University, I want to thank my co-supervisor Dorina Petriu, Murray Woodside, and Dorin Petriu, for their work and suggestions to make this research work possible. At the University of Ottawa, I want to thank Gregor v. Bochmann and Shabaz Maqbool for their contributions to the CSM meta-model. I also want to thank Gunter Mussbacher for his comments on this thesis.

Finally, I want to thank my parents and my parents in law for their understanding and support. I especially would like to thank my wife, Lihong, who always encourages me, admires me, and pushes me forward. When I was off track, she let me do my writing without distracting me with the mundane needs of running the household (I did the same thing when she prepared her CGA exams).

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
AD	Activity Diagram
CD	Class Diagram
CSM	Core Scenario Model
DD	Deployment Diagram
DOM	Document Object Model
EMPA	Extended Markovian Process Algebra
EQN	Extended Queueing Network
GSPN	Generalized Stochastic Petri Net
LQN	Layered Queueing Network
MSC	Message Sequence Chart
PA	Process Algebra
PEPA	Performance Evaluation Process Algebra
QN	Queueing Network
RA	Resource Acquire
RR	Resource Release
SAX	Simple API for XML
SD	Sequence Diagram
SPE	Software Performance Engineering
UCM	Use Case Maps
UCM2LQN	UCM to LQN transformation tool
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSD	XML Schema Definition

1. Introduction

This thesis is part of the PUMA project, which develops a unified approach to building performance models from design models [27]. For that purpose, one of the first steps is to define a scenario definition interface to integrate a wide variety of scenario specifications. This thesis presents a Core Scenario Model (CSM) and its XML syntax, which result from a collaborative effort between PUMA team members. This thesis also presents a tool-supported approach to transform Use Case Maps (UCM) design models to CSM models. This transformation is validated through systematic testing.

1.1 Motivation

Performance is an essential quality attribute of every software system. Software performance is an indicator of how well a software system or component meets its requirements for timeliness [30]. Performance failures may cause the inability of a software product to meet its overall objectives. The cost of fixing those failures sometimes may be huge, especially when the problem is buried in the implementation. Therefore, software performance analysis is very important and necessary early in the software development cycle. Early software performance analysis often involves the construction of predictive performance models from software requirements and specifications, the evaluation of the performance models using different solvers, and the generation of relevant performance results back to software developers. Through software performance analysis, software developers can detect critical performance problems in the early development phases, rather than after those critical performance problems have already been frozen in the software products, in which case it will take much more time and money to correct those problems. Software performance analysis can also help software developers select appropriate solutions when there are several alternative software architectures to choose from. In the past fifteen years, a new discipline called Software Performance Engineering (SPE) has emerged because of the importance and complexity of performance analysis.

Software Performance Engineering is a systematic, quantitative approach to constructing software systems that meet performance objectives. It applies proven techniques to predict the performance of emerging software and responds to problems while they can be fixed with a minimum of time and efforts [30]. SPE requires knowledgeable and

trained people to implement it. Since software developers are not performance analysts, major efforts are still required to integrate software performance analysis into the ordinary software development process [6]. To ease this integration, there are two key factors that need to be considered for software performance analysis.

The first key factor is to *automate* the performance model building process. To do software performance analysis, we need performance models that characterize the quantitative behaviour of software systems. The starting point is the software behaviour model that represents key facets of software execution behaviour. The transformation to a performance model should be automated. Several approaches have proposed different solutions.

In Queueing Network based methodologies, Williams and Smith [30] proposed the first approach based on the Software Performance Engineering (SPE) methodology. They made use of UML class diagrams, deployment diagrams, and sequence diagrams to describe the software architecture and software behaviour. Petriu et al. [27][28] proposed three pattern based methodologies. They also specified the software architecture by using UML class, deployment, sequence, activity, and use case diagrams. Woodside and Petriu [23] showed the automatic transformation from UCM design models to Layered Queueing Network (LQN) performance models. They defined software architecture and software behaviour by means of UCM models, which are enriched with performance annotations, such as work load distribution and service requests.

In Petri Net-based approaches, UML use case diagrams, sequence diagrams, and statechart diagrams are widely used to describe software behaviours. Other methodologies, such as simulation-based approaches and stochastic process-based methodologies also made use of different UML diagrams.

In general, theses approaches have different starting points and different transformation tools. They use different combinations of UML views and/or UCM view to model software systems, and some of them even require intermediate models during the transformation. The availability of tools obviously restricts the number of source modelling languages to choose from. Software developers have to make compromises while modeling software system in order to enable (automated) performance analysis.

Having different transformation tools also becomes a major concern in the automatic transformation process. There are presently 15 different transformation methods described by a recent survey published by IEEE Computer Society [6], and there are likely other unpublished approaches used in the research community. Each method fo-

cuses on its own source input format and its own target performance model. Regardless of the complexity required to understand each method, the immediate effect has created confusion in the selection of appropriate inputs and outputs. The time is appropriate for a unification of existing achievements.

The second key factor is to provide *feedback* automatically. Here we mean by feedback the process of translating performance evaluation results to software design decisions. Software developers should be notified about which components or interactions are responsible for performance problems, and they can also make meaningful design decisions according to the feedback information. Therefore, software architecture and behaviour models can be improved accordingly. The current situation is that few early performance analysis approaches provide some sort of design-level feedback information, according to the survey provided by Balsamo *et al.* [6]. How we can successfully and easily provide automatic feedback support for software developers to improve performance issues will contribute to the applicability of early software performance analysis.

1.2 Goals and Contributions of the Thesis

1.2.1 Goals

To solve the complexity in choosing transformation tools, we propose a Core Scenario Model (CSM) that supports multiple source design models and target performance models to reduce the number of transformation tools. Our first objective is to define the basic elements in CSM. Since most of inputs for early performance analysis come from software architecture and behaviour models, and since the current trend is to express that information in a standard scenario language, our approach consists in extracting the common scenario information from multiple views of UML and UCM, define them as core elements in CSM, and then supplement CSM with performance elements and some other additional attributes which are specific to the individual views.

Our second objective is to automate the transformation from UCM design models to CSM models. Other people can make use of the resulting CSM models and transform them to LQNs, Petri Nets, or some other performance models.

1.2.2 Contributions

The thesis makes the following contributions:

- Identification of common elements among UCM, UML activity diagrams, and sequence diagrams.
- Definition of a Core Scenario Model (in collaboration with others).
- Definition of algorithms for transforming UCM models to CSM models.
- Integration to the UCMNav framework.
- Validation and testing of the transformation algorithm.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 describes the background of scenario models and performance models in detail. It also reviews the XML schema and transformation mechanisms and tools. Chapter 3 compares the common attributes of UCM, UML activity diagram and sequence diagram, lists the requirements for CSM, defines the concepts of CSM, and finally gives the benefits of CSM. Chapter 4 describes the transformation algorithms from UCM design models to CSM representations. Chapter 5 discusses the validation of the transformation tool and chapter 6 presents conclusion and future work.

2. Background

This chapter provides general background information on Use Case Maps (UCM), UML 2.0 activity diagrams, UML 2.0 interaction diagrams, performance modeling languages, and XML schema. Given the recent availability of UML 2.0, many elements of activity and sequence diagrams will be reviewed in detail.

2.1 Use Case Maps and UCMNav

The Use Case Maps (UCM) notation is a simple and very useful notation for requirements specification, system design, and testing. UCM captures system behaviour at a high level of abstraction [7]. It also enables software designers to consider different software architectures at the earliest stages of design [2]. When compared to the Unified Modeling Language (UML), UCM fits in between use cases diagrams and UML interaction diagrams [1].

As an effective effort of an active user community, UCM is gradually known and accepted by more and more academy and industry users. UCM is also being proposed as an ITU-T standard as part of User Requirement Notation (URN) [12].

2.1.1 UCM Meta-model

The UCM meta-model used in UCMNav defines all the vocabularies that could be used in a UCM design model. There are three packages in the UCM meta-model (see Figure 1). Package Top includes all the top level elements; Package Path includes path elements; and Package Performance includes information related to performance (detailed discussions follow).

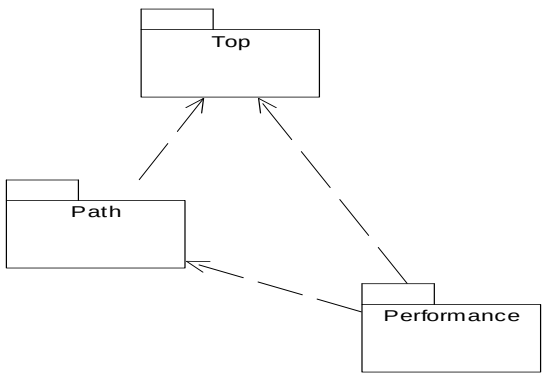


Figure 1 Package diagram for UCM meta-model

2.1.1.1 Package Top

Package Top shows that a UCM design is composed of a collection of root maps, a collection of plug-in maps with their bindings, a collection of path variables, and a collection of scenario groups. Both root maps and plug-in maps define functional requirements as UCM models. Plug-in bindings, which include in-connections and out-connections, connect plug-in maps to their containing stubs. A UCM design also includes a list of responsibility and component definitions, which are referenced in specific models.

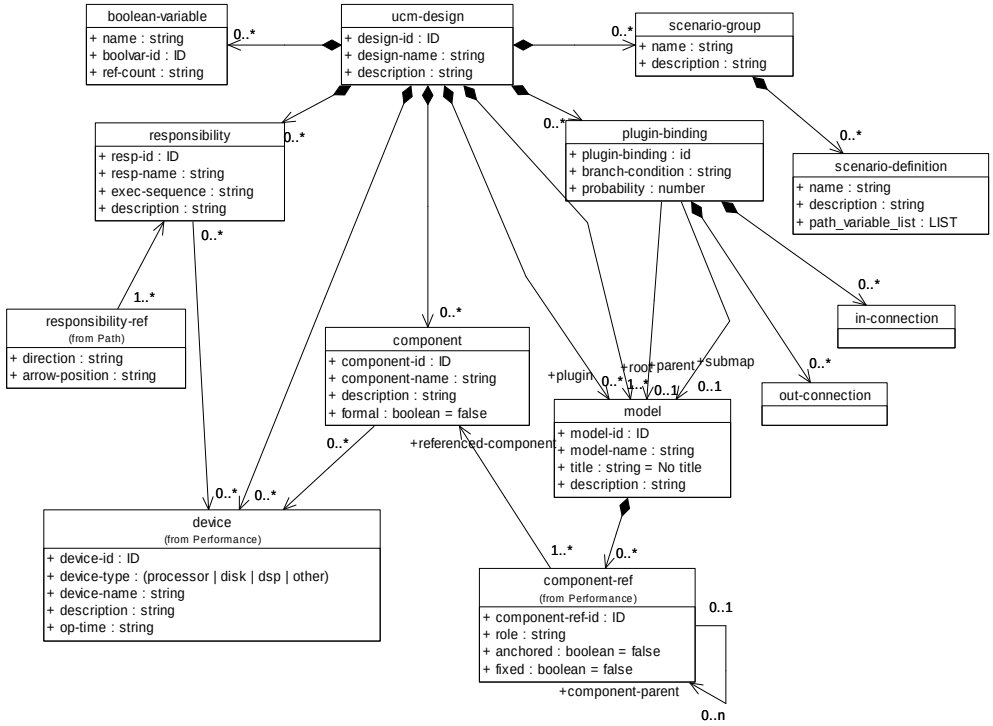


Figure 2 Package Top

2.1.1.2 Package Path

Package Path defines all the elements (called hyperedges) that make up the paths and their connections. Figure 4 lists all the basic path constructs: start and end points; responsibility references; OR-forks and OR-joins; AND-forks and AND-joins; empty points; loops; stubs and so on. Figure 3 shows that a model (i.e., a map) includes a hypergraph, connections between hyperedges, and stubs. Hypergraph defines paths and includes a collection of hyperedges. Hyperedge-connection is a list of connections between hyperedges. Stubs are the container for plug-in maps.

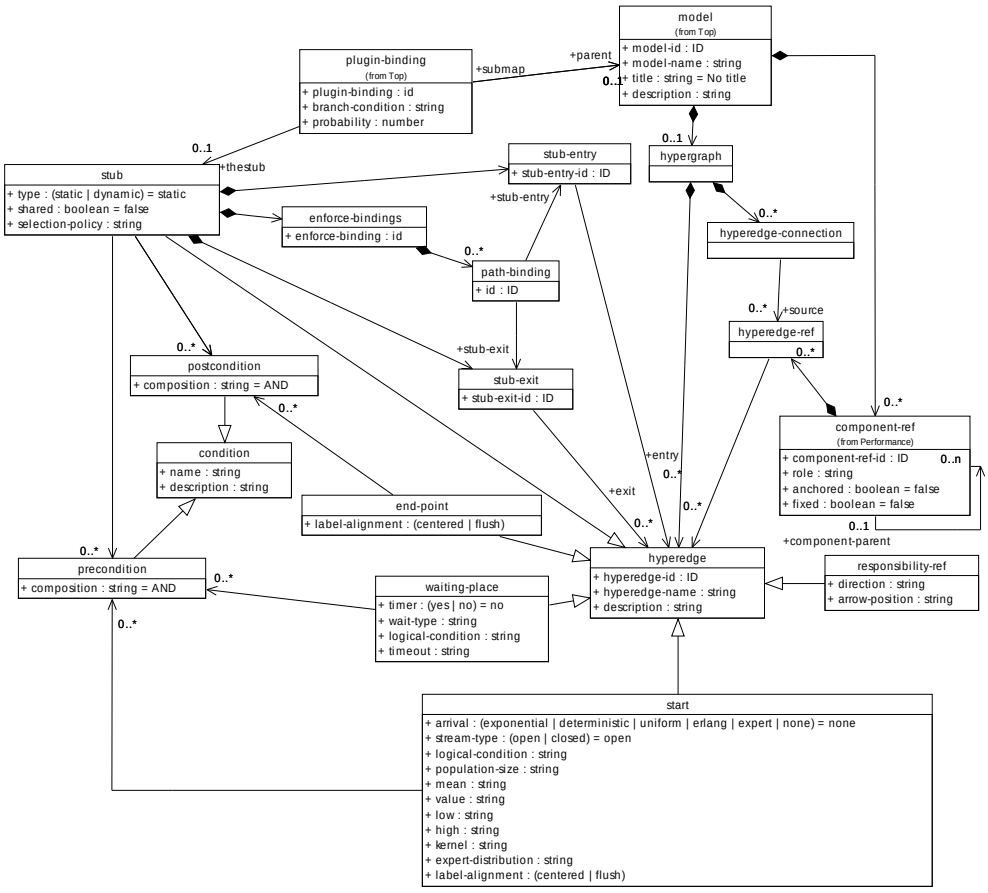


Figure 3 Package Path (1)

2.1.1.3 Package Performance

Package Performance includes specific hyperedges and data structures that can be used to create performance models. Devices represent an execution environment and components show the structure information in a UCM. A response-time-requirement contains two references to timestamp points (starting and ending points) found on a path.

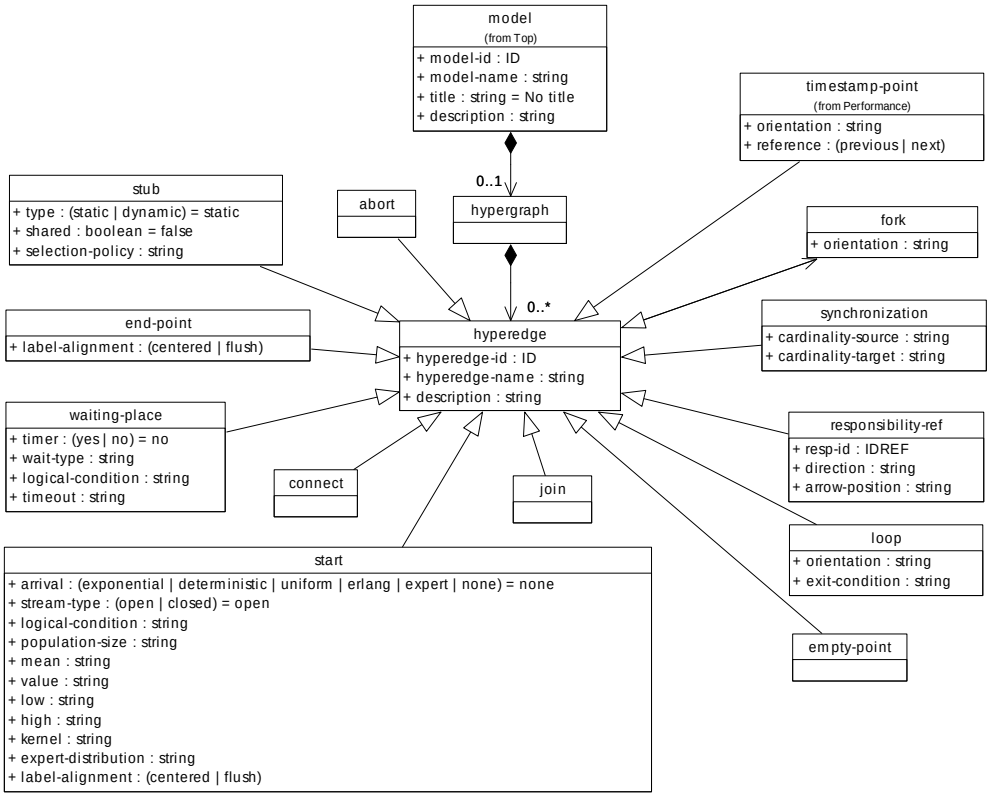


Figure 4 Package Path (2)

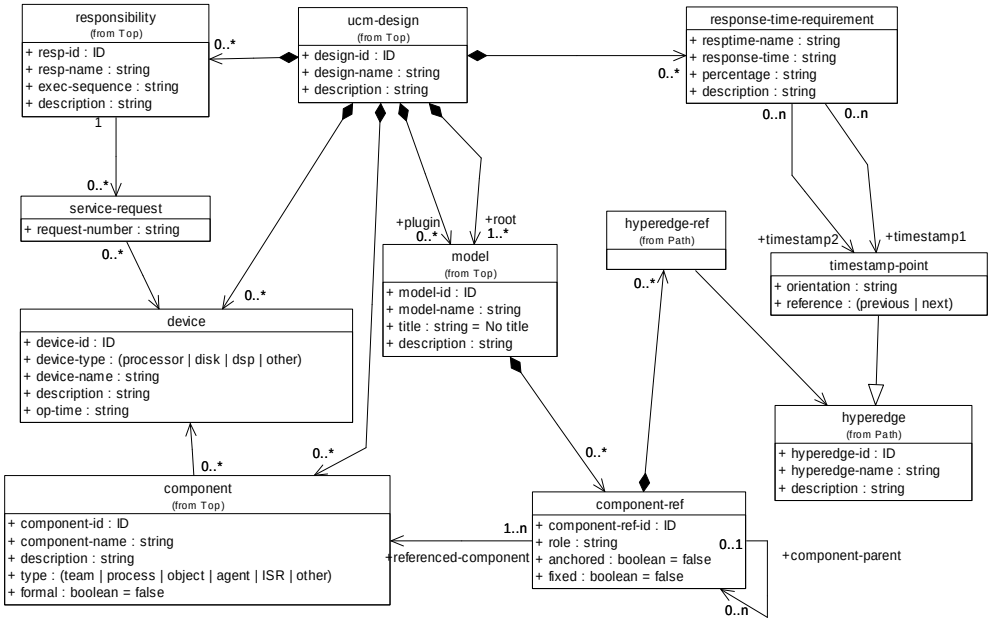


Figure 5 Package Performance

2.1.2 UCM Notation

A UCM is a collection of elements that describe one or more scenarios unfolding through a system [7]. We are going to introduce these basic elements in this section and then compare them with UML concepts in Chapter 3.

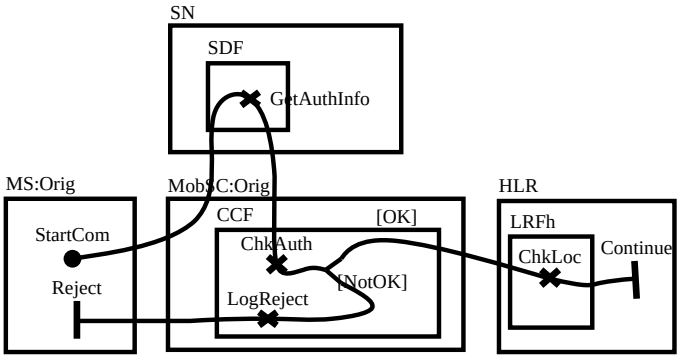


Figure 6 A simple UCM example [2]

The above example (Figure 6) shows how a scenario is represented by a path, which is a line with a start point (a filled circle) and an end point (a bar). The path is traversed by imaginary tokens from start points to end points.

UCM paths can be superimposed on components. Components represent functional or logical entities that are encountered during the execution of a scenario. They can represent both hardware and software resources, as well as actors.

With the addition of responsibilities, UCM paths can show more detailed scenario information. Responsibilities represent functions that need to be accomplished by a software system. Within the graph, responsibilities are denoted with an X-shaped mark on the path, and they can be allocated to specific components.

UCM paths can also be refined by adding another construct called a stub (shown as a diamond). A stub contains separately specified sub-maps called plug-ins. A plug-in has further detail information about a given aspect of a scenario. For a given dynamic stub, there may be several alternative plug-ins.

In UCMs, there may be several paths in parallel. A UCM synchronization construct is used to indicate where parallel path segments fork or join. Two kinds of synchronization are defined in UCM: AND Fork and AND Join. An AND Fork represents one path splitting into two or more parallel paths whereas an AND Join represents two or more parallel paths gathering into one path and synchronizing.

Scenario alternatives are represented by OR Forks and OR Joins. An OR Fork represents a single path splitting into two or more alternative paths. A choice between alternatives is being made and only one of the possible branches may be traversed after the fork. An OR Join represents two or more alternative paths merging into a single path. Before processing further, one of the possible paths leading into OR Join needs to be traversed.

UCM also defines a loop construct to represent looping structures. The loop construct indicates that the body of the loop is traversed a certain number of times (until an exit condition becomes true) before the traversal of the main path resumes.

2.1.3 UCM Navigator

The UCM Navigator (UCMNav) is a UCM editor and repository manager developed by Andrew Miga [21]. It is currently used at Carleton University and the University of Ottawa, as well as some other universities around the world and some industrial users. The UCMNav tool enables users to do the following (among other functionalities):

- Draw and edit UCMs, including multiple scenarios, and store the diagram information in an XML format.
- Add comments and descriptions for the design and/or individual elements.
- Specify system devices and time requirements along paths.
- Generate Message Sequence Charts (MSC) from UCMs.

As more and more functionalities are integrated to UCMNav, UCMNav works more like a framework, rather than just an editor. For instance, Petriu added an UCM2LQN export mechanism to UCMNav in his thesis [23]. UCM2LQN is an automated conversion tool that converts annotated UCM design models into LQN performance models. It works as a link between the UCMNav and two LQN analysis tools: LQNS and ParaSRVN. Echihabi developed a complementary tool called UCMExporter to transform UCM scenario information into TTCN, MSC, and UML sequence diagrams XMI [4]. Jiang recently added an export mechanism to integrate UCM models with other requirements in a requirements management system (Telelogic DOORS) [25]. This thesis presents another UCMNav extension to export UCM models to the CSM representation.

2.2 UML Activity Diagrams

2.2.1 Purpose

UML 2.0 activity diagrams are typically used for modeling the workflow and control flow captured by use cases or scenarios [19]. Activity diagrams emphasize the sequences of actions in scenarios but do not necessarily indicate who perform these actions. In this section, we introduce the basic activity diagram notation and point out the differences in activity diagrams from UML 2.0 and UML 1.x.

2.2.2 Activity Diagram Notation

In the following sections, we use *data* and *object* interchangeably since they are unified in UML under the notation of classifier. The term *token* is a general word for control and data values that flow through actions and edges.

In general, UML2 activity diagrams contain activity nodes and edges. Tokens flow along edges and are operated on by activity nodes or stored temporarily in object nodes. There are three kinds of activity nodes in activity diagrams:

- *Action nodes* operate on control and data values they received, and provide tokens to other nodes after finishing executions.
- *Control nodes* sequence the flow of control and data among activity nodes. These include constructs for modeling contingency (decisions and merges), for proceeding along multiple flows in parallel (forks and joins), and for starting and ending flows.
- *Object nodes* represent objects and data as they flow in and out of nodes, which hold data tokens temporarily while they wait to move downstream.

Action nodes are connected by two kinds of directed edges, i.e., control flow edges and object flow edges. Control flow edges connect activity nodes to indicate that an action cannot start until the previous one is finished. Objects and data cannot pass along a control flow edges. Object flow edges connect object nodes to provide inputs to activity nodes. Only objects and data can pass along an object flow edge.

The rest of this section details the basic notation using small examples.

2.2.2.1 Action Nodes

The executable functionality in the activity diagram is represented by actions and activities. An action represents a single executable step and cannot be further decomposed within an activity. Actions are represented as round-cornered rectangles. Sets of inputs and outputs pins, which are connected by incoming and outgoing activity edges, specify the control flow and data flow tokens from and to other nodes. All inputs to an action are required to be available before execution. When the execution of an action is complete, it may enable the execution of a set of successor nodes by offering tokens in the outgoing pins, where they are accessible to other actions.

An activity represents a user-defined behaviour which is composed of individual actions. The notation for an activity is a combination of the notations of nodes and edges it contains plus a border. Activities can be parameterized. Activity parameters are modeled as a special kind of object node, which is called activity parameter node, to hold temporarily inputs and outputs to activities. Parameter nodes are not pins, because pins are used to connect actions in a flow.

One thing to note is that the execution of a single action, such as a call behaviour action, may induce the execution of many other actions, which are defined in a referenced activity. In such a case an action may not be atomic.

2.2.2.2 Activity Edges

Between two activity nodes there is an activity edge. An activity edge is a directed connection with token flow between activity nodes. There are two kinds of activity edge in activity diagrams: control flow and object flow. A control flow edge can only pass control tokens, which means that objects and data cannot be passed along a control flow edge. Explicitly modeled control flows are new to activity diagrams in UML 2.0. An object flow models the flow of objects to or from object nodes. It may have an action node on at most one end. Object flow replaces the use of transitions in UML1.5 activity diagrams.

The notation for both activity edges is the same. An arrowed line connecting two activities may notate a control flow or an object flow edge. They are distinguishable by usage. Control flow edges connect action nodes directly, and object flow edges connect object nodes or input and output pins of actions.

2.2.2.3 Initial Node

An initial node is the starting point for invoking an activity. The initial node receives a control token when the activity starts and passes it to outgoing edge. Initial nodes are represented by a filled cycle (see Figure 7) and cannot have edges coming into them.

An activity may have more than one initial node, but it might be clearer to use one initial node connected to a fork node to initialize multiple flows. Initial nodes are not required for an activity to start execution.

If an initial node has multiple outgoing edges, only one of these edges will receive the control token since the initial node cannot copy token as a fork can.

2.2.2.4 Final Node

A final node is an abstract control node where a flow in an activity stops. The activity diagram has two kinds of final nodes: flow final and activity final.

A flow final is a final node that terminates a flow. Shown as a circle containing an X, the flow final node has no outgoing edges; therefore, any token or data coming into flow final node is simply destroyed. Since object tokens are just references to objects, destroying an object token does not destroy the object. The flow final node indicates that a process stops at this point and it has no effect on other processes in the activity.

On the other hand, an activity final is a final node that stops all flows in an activity. Shown as a solid circle with a hollow circle, the activity final node indicates that an activity is terminated, i.e., all the flows in the activity are stopped. An activity may have more than one activity final node. The first one reached terminates all flows in the activity.

2.2.2.5 Fork and Join Nodes

Similar to UCMs, activity diagrams use fork and join nodes to indicate parallel activities between multiple flows. The fork node denotes the start of parallelism and the join node denotes its end. The notation for fork and join nodes is simply a line segment (see Figure 7). However, a fork node must have a single edge entering it and two or more edges leaving it, whereas a join node must have one or more edges entering it and only one edge leaving it.

Fork nodes split a flow into multiple concurrent flows. Tokens arriving at a fork are duplicated across outgoing edges. Tokens from the incoming edges are all offered to

the outgoing edges. Since object tokens are only references to objects, duplicates of the object token are the copies of reference to object. It is not required that flows coming out of the fork be synchronized. There is no synchronization restriction on concurrent flows in a UML 2.0 activity, as there are in a UML 1.x activity, which is a kind of state machine.

Join nodes synchronize multiple flows. Tokens must be available on every incoming edge in order to be passed to outgoing edge. If all the incoming tokens are control tokens, then one control token is passed to the outgoing edge; if some of the incoming tokes are control tokens and others are data, then only the data tokens are passed to the outgoing edge. The control tokens are destroyed. Modellers should ensure that joins do not depend on the arrival of tokens that may never arrive. A decision node may be needed to avoid such problems.

2.2.2.6 Merge and Decision Nodes

To indicate alternative activities between multiple flows, activity diagrams define merge and decision nodes. The notation for both nodes is a diamond-shaped symbol. The difference is that a decision node has one incoming edge and multiple outgoing edges, whereas a merge node has multiple incoming edges and one outgoing edge.

A merge node brings together multiple alternate flows. Merge nodes do not synchronize concurrent flows but accept one of the several alternate incoming flows. All tokens arriving at a merge node are offered to the outgoing edge.

A decision node chooses a flow direction among outgoing edges, but exactly which direction is determined by the outgoing edges. Usually, the outgoing edges have guarding conditions that are evaluated to determine if the token can pass along the edge. The order in which the guards are evaluated is not defined. Each token can traverse exactly only one outgoing edge. Tokens are not duplicated. Therefore, the modeller should arrange only one guard can succeed for each token, otherwise there will be race conditions among the outgoing edges.

2.2.2.7 Activity Partition

Activity partitions (formerly known as swimlanes) are groups of actions that have some characteristic in common. Similar to components in UCMs, activity partitions can relate actions to a class that is responsible for executing them. Here we mean by “responsible”

that a class supports the behaviour that is invoked by actions in the partition. The class owns the behaviour, but there is a question of which instance in particular will be selected. There are some of ways to show this. The object flow could be specified to tell which instance should be the target at runtime. An alternative would be to use partitions that represent instances directly. The activity diagram in Figure 7 shows three activity partitions, i.e., Order Department, Accounting Department and Customer. But this is only useful for individual scenarios, not for specifying a behaviour that must operate on many instances.

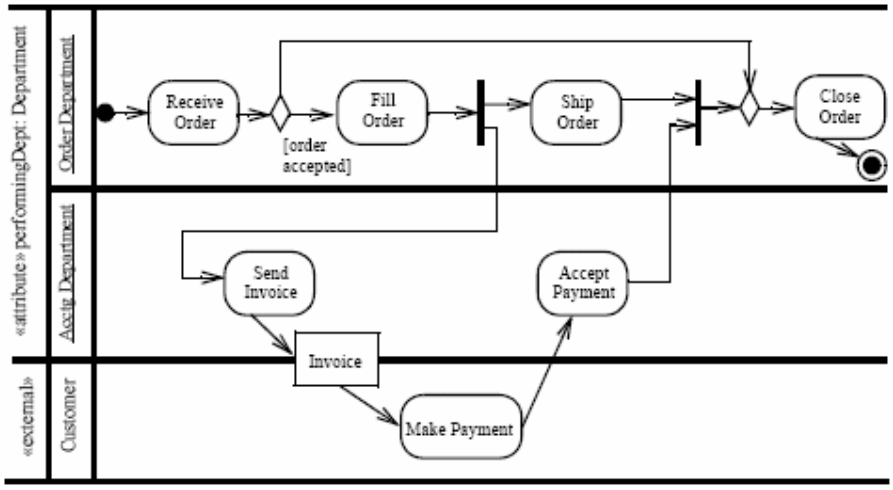


Figure 7 Activity diagram with objects and partitions [19]

2.3 UML Sequence Diagrams

2.3.1 Purpose

The purpose of a sequence diagram is for an individual designer or for a group to get a common understanding of an interaction situation. By showing how various objects interact, sequence diagrams document how the system works for a given scenario.

Another primary use of sequence diagrams is to refine the requirements expressed as use cases to a more formal and detailed level in the design phase.

2.3.2 Basic Notation

Sequence diagrams focus on messages and on the order in which these messages occur. They convey this information along two dimensions: the horizontal dimension shows, from left to right, the object instances that messages are sent to or from; the vertical di-

mension shows, top down, the sequence of messages in time. In the following sections, we introduce the basic concepts and notations in UML 2.0 sequence diagrams [19].

2.3.2.1 Lifeline

A lifeline represents a role or an object instance that participates in the interaction. Usually lifelines represent only one interacting entity. Lifelines are drawn as a box, placed on the top of the diagram, with a dashed line descending from the center of the bottom edge. Inside the box is the lifeline's name.

2.3.2.2 Message

A message defines one specific kind of communication between lifelines. To show an object instance (i.e. lifeline) sending a message to another object, we draw a line from the sender to the receiver. The form of the line's arrowhead reflects properties of the message: an open arrow head represents an asynchronous message, where sending object does not pause to wait for results. A filled arrowhead represents a synchronous message, and this message represents the method/operation that receiver's class implements. There will normally be a return message, represented as a dashed line, from the called object to the calling object whenever there is a return value.

2.3.2.3 Guards

A guard is a Boolean expression to represent the condition that must be met when a message to be sent to an object. The notation for a guide is shown in square brackets, and the format is: [Boolean expression]. Guides are placed above message lines just in font of message names. A guard can only be assigned to a single message.

2.3.2.4 Combined Fragments

UML 1.x sequence diagram has problems to handle the logical sequence of behaviours, such as the choice of behaviour and parallelism. In UML2, sequence diagram defines a notation element called combined fragment (see Figure 8) to solve the problem. A combined fragment groups a set of messages together to show conditional flows. Combined fragments are defined by an interaction operator and corresponding interaction operands. The UML2 specification defines twelve interaction operators, and I will cover only those that are similar as in UCM.

The first is to show alternatives. Alternatives are used to model "if then else" logic and means that at most one of the message sequences will execute. The interaction operator *alt* (See Figure 8) in the combined fragment represents a choice of behaviour. Alternatives may have more than two operands. UML2 sequence diagrams have another similar operator called option (*opt*), which is semantically equivalent to an alternative except that that the option has one non-empty operand and the second operand is empty.

To show the reference to another sequence diagram, UML2 requires the operator called *ref* and a new notation element called interaction occurrence. The addition of interaction occurrence is the most important innovation in UML2 interaction modeling. Interaction occurrences give us the ability to compose primitive sequence diagrams into more complex sequence diagrams and to reuse the simple sequence diagrams.

To represent a parallel merge between the message sequences, we use parallel combined fragment. The interaction operator *par* illustrates that two or more processing sequences are in parallel. After all of the messages are done, the next message after parallel combined fragment can start to execute.

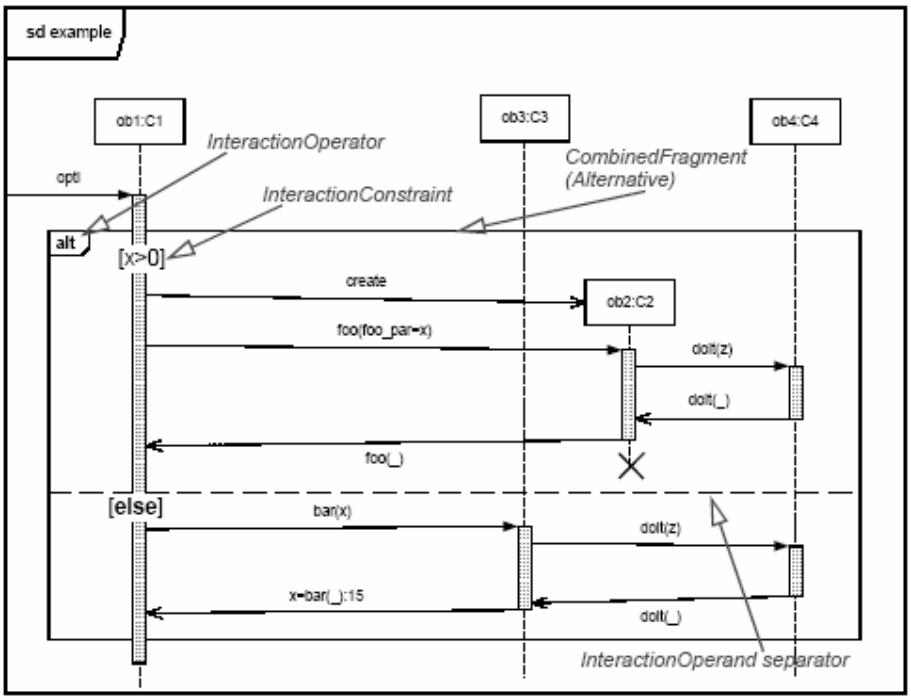


Figure 8 Interaction diagram [19]

The interaction operator *loop* represents that the interaction operand (see Figure 8) will be repeated a number of times. The guard may include a lower and upper numbers of iterations as well as a Boolean expression [19]. A loop will iterate minimum the lower number of times and at most the upper number of times.

2.4 Performance Modeling Languages

Although the main focus of this thesis is not on performance models, the output from the transformation covered by this thesis will be the input used to generate various performance models. In CSM, there are also elements related to performance modeling. Therefore, we cover some general background information about performance modeling language in this section.

2.4.1 QN and LQN Models

Queueing networks (QNs) is a model in which jobs departing from one queue arrive at another queue [13]. Basically, a QN includes several queue-servers connected in a network [30]. Clients request services from several resources (e.g., the CPU, the disk, and so on), and these requests enter the queue of servers. After being served, the requests proceed to the next queue in the network or just exit the system. There are two types of QN models: closed models and open models. A closed QN model has a fixed number of jobs circulating among queues; an open QN model has external arrivals and departures, and we call this job arrival rate as "workload intensity".

A layer queuing network model (LQN) is a kind of queuing network model [15]. QN model can have only a single set of client-server relationships [23], but a LQN model organizes the resources into clear layers. Layers provide ordering, and with a proper layering, deadlock among requests is impossible. For this reason, LQN is very common in practice.

The LQN notation uses the term task, entry, service request, host processor, and demand. We cover these concepts one by one in the following:

A *task* is an interacting entity in LQN. It has operations that other tasks can request. Tasks also have properties of resources, including a queue. The user of the system is a special reference task that does not receive any request.

There are one or more *entries* in one task. Each entry represents the service the task can provide. Entries may have their own service demands from other tasks or from devices.

Service requests are calls for services from one entry to an entry of another task. There are three type of service request: synchronous, asynchronous and forwarding. Synchronous service request needs a reply. The task sending a synchronous message suspends its normal execution until it receives the reply. Asynchronous call does not require

a reply. The task sending an asynchronous call continues its normal execution without waiting. The synchronous call may be forwarded to the third party by the call receiving task. Now it is the third party's responsibility to reply to the original calling task, which is being blocked and waiting for the reply.

A *host processor* models the physical entity that carries out operations. This separates the logic part of an operation from its physical execution. The task or entry shows the logic part, and the host processor illustrates the physical device. The benefit of this separation is the ability to model the operations in more details.

2.4.2 Stochastic Petri Nets

Stochastic Petri Nets (SPNs) are a unifying formal specification framework [6] developed for modeling computer system performance. SPNs consist of places and transitions as well as functions. The basic functions are input, output and weight functions. The initial state is represented by the initial marking. SPNs can be represented graphically, with places represented as circles, transitions as rectangles, and input and output functions as directed arcs.

SPNs have discrete state spaces, defined by the number of objects in each place (the marking). Places can be linked to transitions as input places, and transitions can be linked to output places. Transitions are enabled when there are enough objects in each of the input places. The enabled transitions can fire, removing objects from their input places and adding objects to their output places.

2.4.3 Stochastic Process Algebra

System behaviour generally consists of processes and data. Processes are the control mechanisms for the manipulation of data [10]. Process algebra constitutes a framework for formal reasoning about the process and data. The building blocks are processes and actions. Most process algebras contain basic operators to build finite processes and capture infinite behaviour.

Stochastic process algebra (SPA) is an extension of classical process algebra. In order to describe and analyze both functional and performance properties of software specifications within the same framework, SPA has added several new features to the performance modeling. These features, such as compositionality and formality, allow performance models to be constructed in a way natural to modellers. On the other hand, modellers are required to be able to specify the software specifications using SPA and to

associate the performance parameters to actions. A number of stochastic process algebras have been defined, such as PEPA (Performance Evaluation Process Algebra), EMPA (Extended Markovian Process Algebra), and TIPP (TIme Processes and Performability evaluation).

2.5 UML Performance Profile

The definition of the CSM meta-model is partly inspired from the UML performance profile "UML Profile for Schedulability, Performance, and Time Specification" [17], also called simply performance (or SPT) profile in this thesis. This profile determines the structural properties for performance domain model, and the CSM meta-model is to capture the essential entities in the performance domain, therefore, CSM meta-model has to be inferred from the UML and the UML performance profile [26].

The performance profile has been adopted as a specification of the Object Management Group (OMG). The goal of the specification is to provide a framework for conducting quantitative performance analysis of UML model. It defines a general resource model, time modeling, concurrency modeling, schedulability and performance modeling. Particularly in its eighth chapter, this profile provides a mapping from performance domain concepts to UML equivalent concepts. By doing performance modeling, designers can capture performance requirements, associate performance constraints with selected UML model elements, specify execution demands, and present performance results [17]. In this section, we describe a high-level overview of relevant performance modeling concepts, and will cover more information in chapter 3.

In performance-related models, there are two basic abstractions: scenario and resource. Scenarios define response paths with externally visible end points, and can have QoS requirements such as response time and throughputs. Determining whether these QoS requirements are met is the main objective of performance analysis. Each scenario is executed by a job class with open or closed load intensity. Scenarios are composed of one or more scenario steps. Steps are the elementary operation units and are joined in a sequence, with predecessor-successor relationships, such as forks, joins, loops and so on. Each step also has a mean execution count, which is the average number of repetition times. A step may be redefined by a sub-scenario, or it can invoke other activities. When a step is executed, the execution time taken on its host device is called a host execution

demand for a step. The total host execution demands, including the demands of all its sub-steps, are the resource demands by a step.

There are two kinds of resources in performance models: active resources and passive resources. An active resource usually refers to the processing resource, such as a processor, interface device or storage device. Active resources have service times, which are defined as the host execution demand of the steps that are hosted by the resources. Therefore, the service time of an active resource depends on the set of operations during which the resource must be held. Different classes of steps may have different service times. A passive resource is protected by an access mechanism and is acquired at the beginning of the execution of an operation, and is released at the end. The difference between the start and the end is called the holding time. The passive resource may be shared by multiple concurrent resource operations.

To represent these performance domain concepts in UML, the most widely used design notation, SPT profile discusses the mappings to UML in two ways: a collaboration-based approach and an activity-based approach; it extends the UML by providing stereotypes and tagged values to represent performance requirements, resources and some behaviour parameters.

2.6 XML Schema

XML, the eXtensible Markup Language, is used to describe documents containing structured information. XML is different from HTML in terms of tag semantic and tag sets. In HTML, both tag semantic and tag sets are fixed, for example, an <h1> is always the first level heading, whereas XML specifies neither tag semantic nor tag sets. In other words, XML provides a facility to define tags and the structural relationships between tags. This facility is either a Document Type Definition (DTD) or an XML schema.

DTD is the original XML schema language included in the XML 1.0 specification. However, DTD has many limitations for supporting data interchange in complex applications. The new schema specification extends the capabilities for validating documents and exchanging information with other non-XML system components. Therefore, in this section, we introduce the role of XML schema and take a quick tour of XML schema.

2.6.1 The Role of an XML Schema

XML schema is a XML definition language for describing and constraining the content of XML documents [34]. An XML schema:

- Defines and documents the vocabulary for all users;
- Describes the structure information for XML documents;
- Validates documents when using XML parsers;
- Defines specific data types for elements and attributes;
- Provides default values for elements and attributes;
- Defines the order and the number of child elements.

Although some well formed XML documents may not need an XML schema to be validated, schemas can provide the ability to validate the structural and semantic accuracy of the documents. Comments can also be provided within the text of the schema and supplementary text documentation is written to explain a schema's purpose and usage.

2.6.2 A Quick Tour of XML Schema

In this section, we provide a quick tour of the main components of XML schema [34]. We also introduce a simple example of a schema.

2.6.2.1 An Example Schema

The purpose of a schema is to define a class of XML documents, and the term "instance document" is often used to describe an XML document that conforms to a particular schema. Let us suppose we have an instance document shown in the following:

```
<product effDate="2004-08-31">
    <number>888</number>
    <size>28</size>
</product>
```

This instance document consists of a main element, *product*, and the sub elements *number* and *size*. The main element also has an attribute *effdate*.

Before examining the schema for this instance document, we have to mention the association between the instance document and the XML schema. As we can see by inspecting the above instance document, the XML schema is not mentioned. XML schema specification has explicit mechanisms for associating instances and schemas, and to keep

the instance document simple, here we just assume that any processor of the instance document can obtain the schema without any information from the instance document.

A product schema, which might be used to validate the above instance document, is the following:

```
<xsd: schema xmlns: xsd= "http://www.w3.org/2001/XMLSchema">
  <xsd: element name= "product" type= "ProductType" />
  <xsd: complexType name= "ProductType">
    <xsd: sequence>
      <xsd: element name= "number" type= xsd: "integer" />
      <xsd: element name= "size" type= "SizeType" />
    </ xsd: sequence>
     <xsd: attribute name= "effDate" type= "xsd: date" />
  </ xsd: complexType>
  <xsd: simpleType name= "SizeType">
     <xsd: restriction base= "xsd: integer">
       <xsd: minInclusive value= "2" />
       <xsd: maxInclusive value= "20" />
     </ xsd: restriction>
   </ xsd: simpleType>
</ xsd: schema>
```

The product schema consists of a *schema* element and several subelements, *element*, *complexType* and *simpleType*. Each of the elements in product schema has a prefix *xsd:* which is associated with the XML schema namespace through the declaration in the first line of product schema. The same prefix also appears on the names of built-in simple types, e.g. xsd: integer. The purpose of the prefix is to denote the XML schema namespace and to identify the elements and simple types as belonging to the vocabulary of the XML schema language. More detail information about those elements is covered in the next section.

2.6.2.2 The Component of XML Schema

XML schema is made up of a number of different kinds of components. The most basic building blocks are: element, attribute, simple type and complex type.

Element declarations in schema are used to assign names and data types to elements in instance documents. This is accomplished by using global or local declarations.

Global element declarations appear at the top level of the schema document and must have a *schema* element as their parent. The qualified names used by global element declarations must be unique in the schema. This includes any other schema documents that are used with it together. The global elements can be referenced by other element declarations. On the other hand, local element declarations appear entirely inside complex type definition and can only be used in that type definition block. Local element can not be referenced by other element declarations in other complex type definitions. The qualified names used by local element declarations are scoped to the complex type definition blocks. You can have two completely different local element declarations with the same name, as long as they are in different complex type definitions. You can even have two local elements with the same element-type name in the same complex type definition, provided they have the same data type.

Attribute declarations are used to name attributes and associate them with type declarations. Attribute declaration can be global or local. Same as the global element declarations, global attribute declarations appear at the top level of the schema document and must have a *schema* element as their parent. The qualified names used by global attribute declarations must be unique in the schema. This includes any other schema documents that are used with it together. The global attributes can be referenced in complex type definitions. On the other hand, local attribute declarations appear entirely inside complex type definition and can only be used within that type definition block. The qualified names used by local attribute declarations are scoped to the complex type definition blocks. It is illegal to have two local attribute declarations with the same name. To indicate whether an attribute is required or optional, we can set the *use* attribute. Regardless of whether they are local or global, all attribute declarations have a simple type for an attribute, since attributes cannot themselves have child elements or attributes.

Both elements and attributes can use simple types and complex types to describe their data contents. A basic difference between simple type and complex type is: complex type can have elements in its content and may carry attributes, and simple type can not have element content and can not carry attribute.

There are three kinds of simple type in XML schema specification: atomic types, list types, and union types. Restrictions can be added to another simple type, known as its base type. A simple type restricts its base type by applying facets to restrict its value. XML schema defines fifteen facets. More information about facets can be found in [34].

Complex types are widely used for users to define their own data types. Complex types may be either named or anonymous. Named types are defined globally and can be used by multiple element and attribute declarations. On the other hand, anonymous types can not have names and are always defined entirely within an element declaration. Since they have no name, they may only be used once by that declaration. There are three kinds of content model for complex types: sequence groups, choice groups and all groups. Sequence groups indicate that the order in which the corresponding elements should appear. Choice groups indicate that only one of the corresponding elements must appear. All groups indicate that all of the corresponding elements should appear, in any order, but no more than once each.

2.6.3 Validation Based on Schema

XML is widely used for web development, database development, and documentation. There are many benefits of using XML to interchange information, such as ease of reading and processing, platform and application independence, and so on. When people design the XML documents, they want to make sure that XML documents are formalized, concise and reasonably clear. One way to achieve these goals is to validate XML schema and XML instance documents.

Validation determines whether XML documents conform to all of the constraints described in the schema. The validation process checks the following aspects against schema:

- *Correctness of the data*. Validation can signal invalid formats or out-of-range values.
- *Completeness of the data*. Validation can check that all the required data is present.
- *Shared understanding of the data*. Validation can ensure the same understanding of the data.

There are lots of validation tools available nowadays. They can be executed either via the web or downloaded for local execution. In general, these tools can be divided into three categories: SAX parsers, DOM parsers, and XSD validators. SAX parsers parse documents and validate them; DOM parsers parse documents, build DOM structures and validate them; XSD validators validate the document without having to write an explicit parser.

Among the most popular XML validators we find XSV, Xerces, Oracle XDK, and Microsoft MSXML. In our thesis work, we are using the on-line Xerces parser [5] to validate our schema and instance documents.

2.7 Chapter Summary

This chapter covers the background information that will be used throughout this thesis. The transformation from UCM notation to CSM representation, and ultimately, to various kinds of performance models, involves many most popular specifications, notations, and languages.

Section 2.1 recalls the basic elements of UCM notation. As the input to our transformation algorithm, UCM models have to be interpreted clearly. Section 2.2 and 2.3 explore the UML2.0 activity diagram and interaction diagram notations, and this knowledge, along with the understanding of UCM, provides us the foundation to build our CSM meta-model. Different performance modeling languages and performance profile are also discussed in Section 2.4 and Section 2.5, and we can make use of them to add the performance related information to our CSM model. XML allows us to exchange our model information easily and is discussed in Section 2.6.

With the above background information, we are ready to compare different design notations, extract common properties, and finally define our meta-model for the Core Scenario Model (CSM).

3. Core Scenario Model (CSM)

The Core Scenario Model (CSM) will play an important part in deriving various performance models from different design models. This chapter presents the motivations behind the CSM model, the definition of CSM, and the benefits of using CSM model in transformation processes.

3.1 Overview

In the transformation process from software specifications to performance models, there are myriads of approaches and methodologies. Balsamo *et al*. [6] summarized many of these approaches. From the review in Chapter 1, we can see that many approaches use different combinations of UML diagrams and UCM notation to describe software architectures and behaviours. Each approach is trying to develop its own transformation tool to automate the transformation process. One major problem here is the close coupling between design specification and performance modeling languages. When users use a specific transformation tool, theirs choices are limited to only one required specific design specification and one specific performance language. On the other hand, the drawback of having many different transformation tools is the inconvenience and uncertainty in choosing between different approaches. To use such a transformation tool, a software developer has to know in advance what is the target performance model, what is the input requirement, and so forth. Each transformation tool has its own input/output limitations and requirements. When using another transformation tool, software designers may have to change their software design models to meet the different input requirements. This obviously delays the deployment of transformation process and performance evaluations.

At the same time, we can also see that all transformation processes have many common properties. The transformation tools are based on the availability of suitable abstraction of the final software system. The general tendency is to make use of scenarios to describe software behaviours. Almost all the approaches mentioned before start from UML activity diagrams, UML sequence diagrams, and Use Case Maps. All these three design notations have common concepts. This provides us with a possibility to consolidate all the different transformation processes and get a unified approach.

Recently, Woodside *et al.* proposed a unified approach [25] to integrate early performance analysis with software development life cycle. The main concept is Core Scenario Model, which acts like a standard interface between different software specifications and different performance models. The benefits are very direct.

The first benefit is the separation of concern. CSM acts as a Façade unified interface for performance models. CSM provides an entry point to different types of performance models. By using CSM, we can simply decouple the dependencies between design specifications and performance models.

The transformation process from scenario design models to CSM models is simpler and less time consuming than having to go directly to performance models. The gap between scenario design models and CSM models is very small. Software designers are not always performance analysis experts. They can easily find out the mappings between scenario specifications, and they may have problems or takes more time and effort to find out the appropriate mappings between scenario specifications and performance model concepts, especially when integrating the feedback from performance analysis into the design decisions. By using CSM models, software designers and tool developers can divide the transformation process into two phases, and they do not need to worry about the mapping from CSM models to performance models any more.

Aside from simpler transformation, having an intermediate CSM representation can reduce the number of transformation tools needed. Suppose there are N1 different kinds of scenario design models to be transformed into N2 kinds of performance models one by one, then we need N1 * N2 transformation tools (See Figure 9) to be developed. On the other hand, if we make use of the CSM model, then we need N1 transformations between scenario design models and CSM models, and N2 transformations between CSM models and performance models. The total number of transformation tools needed becomes N1+N2 (See Figure 10).

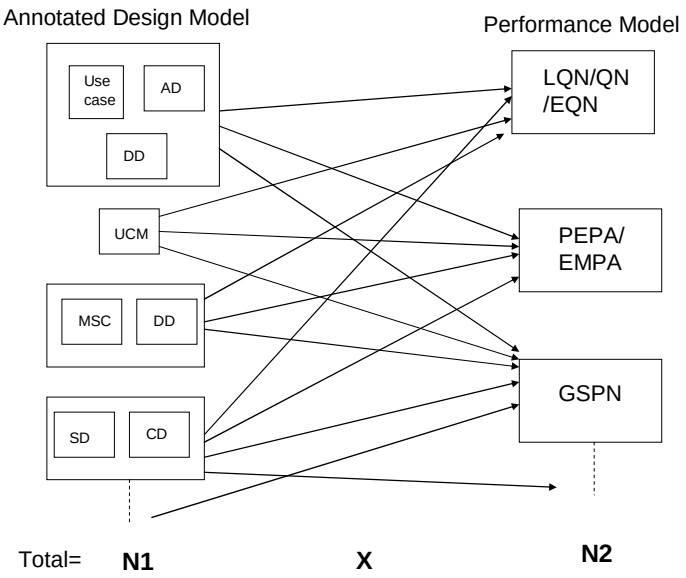


Figure 9 Transformations without CSM

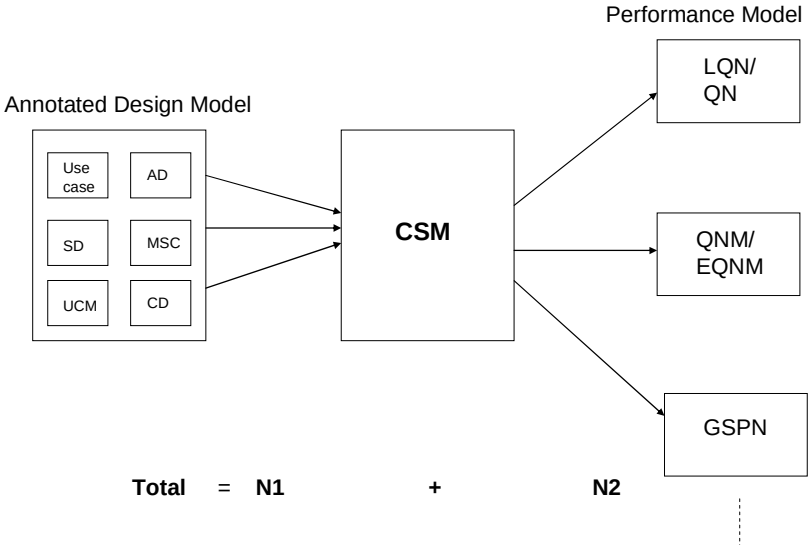


Figure 10 CSM Unified Interface

3.2 Comparison of UCM and UML Diagrams

CSM provides a unified interface to different performance modeling languages, and CSM also has to encompass the common properties between UML and UCM in order to allow different combinations of design specifications as inputs. Therefore, we first need to find out what are the common properties among different design notations. In chapter 2, we have discussed three popular design notations, and we can easily find out there are many concepts having the same semantics. In Table 1, we compare the common properties

among UCMs, UML activity diagrams, and UML sequence diagrams. These will be used to determine what concepts are necessary for the definition of CSM.

Table 1 Commonalities between UCM, Activity and Sequence Diagrams

Commonalities	**UCM**	**Activity Diagram**	**Sequence Diagram**
1. Contents	Integrated view of behaviour and structure at the system level	Show the dynamic behaviour of system in terms of operations	Show dynamic aspects of the system, emphasizing the time ordering of messages
2.Function and action performed	Responsibility Responsibilities are processing tasks (e.g. procedures, functions, actions, etc.) that are referenced by scenarios and by components	Activity (action) Action is the fundamental unit of executable functionality, and an action represents a single step within an activity; activity is a grouping of actions, and an activity represents a behaviour which is composed of actions.	ExecutionOccurrence and message. ExecutionOccurrence represents actions, which are similar as actions in activity diagrams). An executionOccurrence is an instantiation of a unit of behavior within the lifeline; it is represented by the start eventOccurrence and the finish eventOccurrence. The sequences of eventOccurrence along lifeline are the meanings of interactions
3. Entities	Component	Swimlane / Partition	Lifeline

Commonalities	**UCM**	**Activity Diagram**	**Sequence Diagram**
4. Scenario representation	Path (scenario) A possible sequence of execution of actions; path chains responsibility in cause-effect sequence.	The flow of execution is modeled as activity nodes connected by activity edges	Interactions. Message interchange between a number of lifelines.
5. Scenario start and stop	Start point & end point	InitialNode & FinalNode (Flow Final)	Stop node to define the termination of the instance specified on a given lifeline
6. Alternative scenario	Stub Container for plug-in maps	Invoked action Multiple incoming and outgoing points	Interaction Operator *ref* and gate. A gate is a connection point for relating a message outside an interaction with a message inside the interaction. The only purpose of gates is to define the source and the target of messages or general order relations
7. Reusable scenario	Plug in maps	Activity	InteractionOccurrence

8. Concurrency flow	And Fork	ForkNode A fork node splits a flow into multiple concurrent flows.	InteractionOperator *par* represents a parallel concurrency between behaviours
9. Alternative flow	Or Fork	DecisionNode A decision node has one incoming edge and multiple outgoing activity edges, and token can transverse only one outgoing edge.	InteractionOperator *alt* represents a choice of behaviour. Guard expression must be evaluated to true.
10. Time ordering sequence	Causal path sequence	ActivityEdge. Control flow to model the sequencing of behaviours that does not involve the flow of objects. Object flow to model the flow of data and objects in an activity	GeneralOrdering. Represents a binary relation between two EventOccurrences and provides the ability to define partial order of events

11. Loop	Loop Loop element has two source paths and two target paths; exit-condition specifies the condition under which loop exits.	LoopNode is an activity node, representing a loop with setup, test, and body sections.	InteractionOperator *loop* represents a loop. The guard may include a lower and upper number of iterations of the loop as well as a Boolean expression.
12.Alternative merge	Or Join	MergeNode A merge node brings together multiple alternate flows.	InteractionOperator *alt* represents a choice of behaviour. Guard expression must be evaluated to true.
13. Synchronizing concurrent flow	And Join	JoinNode A join node synchronizes multiple flows, and has multiple incoming edges and one outgoing edge.	InteractionOperator *par* represents a parallel merge between behaviours

3.3 Requirements for CSM

Through the comparison in the previous section, we have drawn the common information from UCM, UML2 activity diagram and sequence diagram. By covering those common properties and adding features special to each notation mentioned above, CSM will be able to accept different design specifications as inputs.

However, the thesis mainly focuses on the transformation from UCM to CSM, and CSM is a language with a broad scope that covers the most popular design notations. This suggests that we select only parts of the CSM that are of direct interest, and make sure all of CSM modeling capabilities in this subset are well supported by our transformation. By doing this, we enhance the ability of CSM as a unified interface to accept scenario design notation. For that reason, in this section we, from the UCM point of view, outline the requirements for CSM language according to our previous comparisons.

3.3.1 Scenario Information

3.3.1.1 CSM shall capture actions.

3.3.1.2 CSM shall be able to refine one action as a sequence of actions.

3.3.1.3 CSM shall be able to define communication messages between components.

3.3.1.4 CSM shall be able to group a sequence of one or more actions as scenarios**.**

3.3.1.5 The sequence of actions shall be ordered.

3.3.1.6 CSM shall be able to specify the time ordering of communication messages.

3.3.1.7 CSM shall be able to specify the start point of a sequence of actions.

3.3.1.8 CSM shall be able to specify the end point of a sequence of actions.

3.3.1.9 Two or more concurrent sequences may merge as one sequence.

3.3.1.10 Two or more alternative sequences may join as one sequence.

3.3.1.11 One sequence may fork as two or more concurrent sequences.

3.3.1.12 One sequence may split as two or more alternative sequences.

3.3.1.13 CSM shall be able to specify the guide condition of each alternative sequence.

3.3.1.14 CSM shall be able to capture repetitive actions.

3.3.1.15 CSM shall be able to specify number of iterations for each repetitive action.

3.3.1.16 CSM shall be able to specify guarding conditions for each repetitive action.

3.3.1.17 CSM shall be able to group related scenarios.

3.3.1.18 CSM shall be able to abstract related scenarios into one action.

3.3.1.19 CSM shall be able to reference a group of related scenarios.

3.3.1.20 CSM shall be able to specify preconditions at scenario start points.

3.3.1.21 CSM shall be able to specify post-conditions at scenario end points.

3.3.1.22 CSM shall be able to specify preconditions at the entry points of scenario groups.

3.3.1.23 CSM shall be able to specify post-conditions at the exit points of scenario groups.

3.3.2 Structure Information

3.3.2.1 CSM should be able to define actors responsible for executing each action.

3.3.2.2 CSM should be able to allocate actions to components.

3.3.2.3 CSM should be able to specify the containment relationship of components.

3.3.3 Performance–Related Information

3.3.3.1 CSM should be able to define hardware devices.

3.3.3.2 CSM should be able to define software resources.

3.3.3.3 CSM should be able to describe workload for each scenario at the start point.

3.3.3.4 CSM should be able to define stream of requests for open workload.

3.3.3.5 CSM should be able to define number of jobs for closed workload.

3.3.3.6 CSM should be able to define think time for closed workload.

3.3.3.7 CSM should be able to capture response time for each scenario.

3.3.3.8 CSM should be able to capture throughput for each scenario.

3.3.3.9 CSM should be able to specify mean number of execution times associated with each action.

3.3.3.10 CSM should be able to specify execution time taken on host device for each action.

3.3.3.11 CSM should be able to define resource acquiring points.

3.3.3.12 CSM should be able to define elapsed execution time for each acquired resource.

3.3.3.13 CSM should be able to define number of resource-unit for each acquired resource.

3.3.3.14 CSM should be able to define resource releasing points.

3.3.3.15 CSM should be able to define number of resource-unit for each released resource.

3.3.4 Deployment Information

3.3.4.1 CSM should be able to allocate components to computer hardware.

3.3.4.2 CSM should be able to allocate resources to software components.

3.3.5 Traceability and Performance Results

3.3.5.1 CSM should be able to define clear performance requirement values.

3.3.5.2 CSM should be able to record the elapsed time for each scenario.

3.3.5.3 CSM should be able to record computer resource utilization.

3.3.5.4 CSM should be able to record software resource utilization.

3.3.5.5 CSM should be able to trace back to resource elements.

3.3.5.6 CSM should be able to trace back to scenarios.

3.4 Definition of CSM

By supporting the common properties from UML2 activity diagram, sequence diagram and UCM, CSM aims to improve early performance analysis with reduced transformation time, cost and effort. In this section, we define the concepts of CSM in terms of UML class diagrams and then give detailed descriptions of those concepts.

3.4.1 CSM Metamodel Overview

In CSM, there are four packages (See Figure 11) to organize model elements into groups, making our metamodel easier and simpler to understand. Each package in Figure 11 would lead to a more detail diagram. Figure 11 also depicts the dependencies of the packages. XSD_Datatypes package defines the primitive types, such as integer, double, string and so on. It is a standard package, using for the schema validation purpose. The function and content of other packages are described in the following sections.

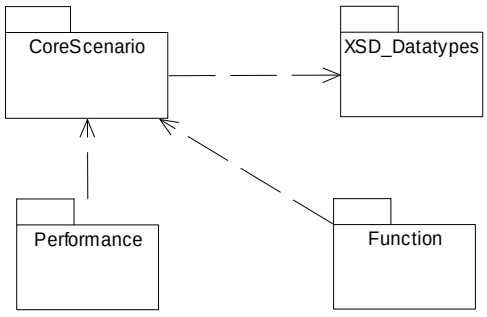


Figure 11 Package diagram of CSM

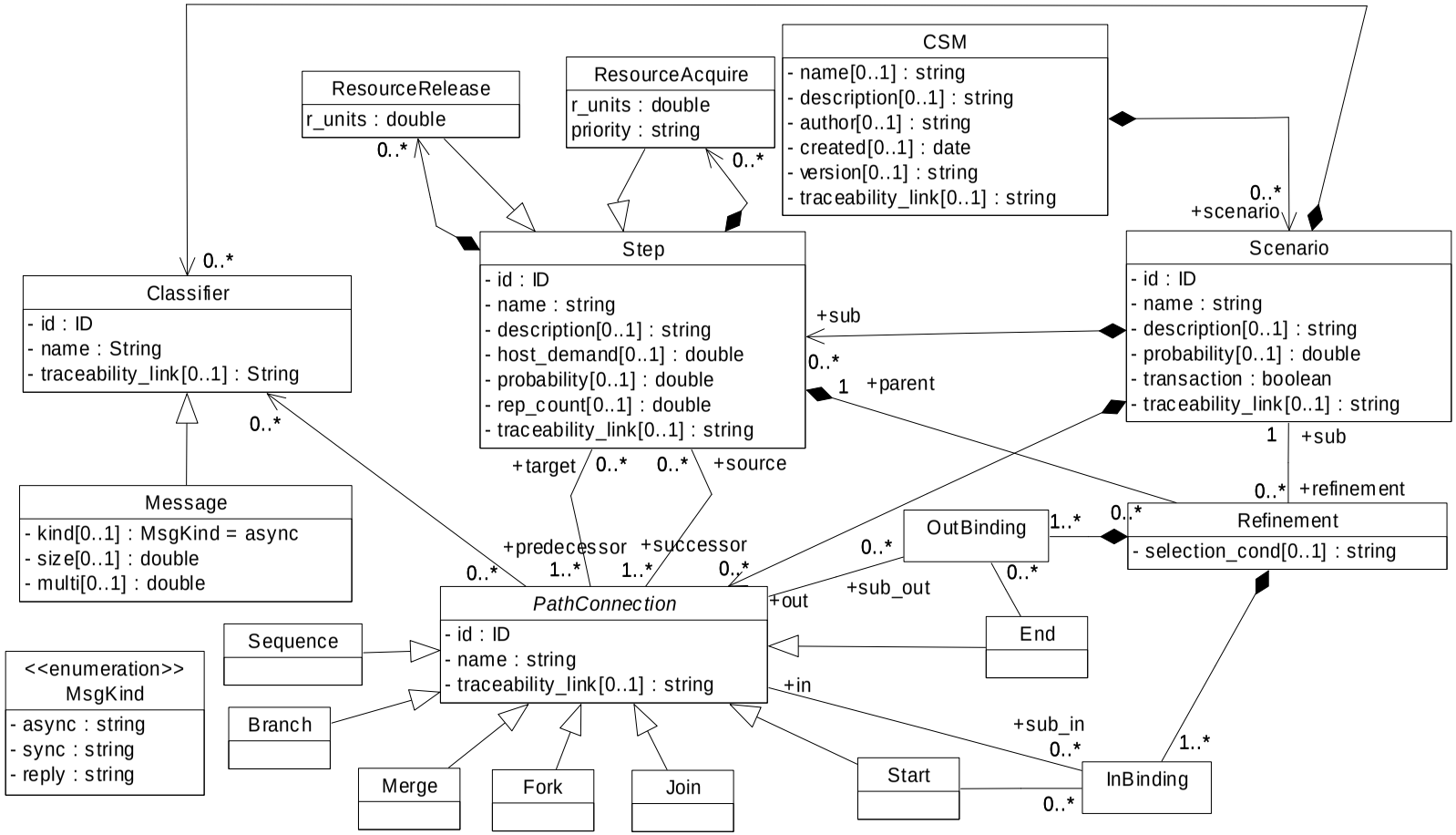


Figure 12 CoreScenario package of CSM

3.4.1.1 CoreScenario Package

The CoreScenario package in Figure 12 is the most important package in CSM. It defines the basic elements used to construct CSM models. Detailed information about each class in this package can be found in the class description section.

3.4.1.2 Performance Package

The Performance package in Figure 13 collects all the elements related to performance information, such as resource, performance requirement, performance result, and so on. Detailed information about each class in this package can be found in the class description section.

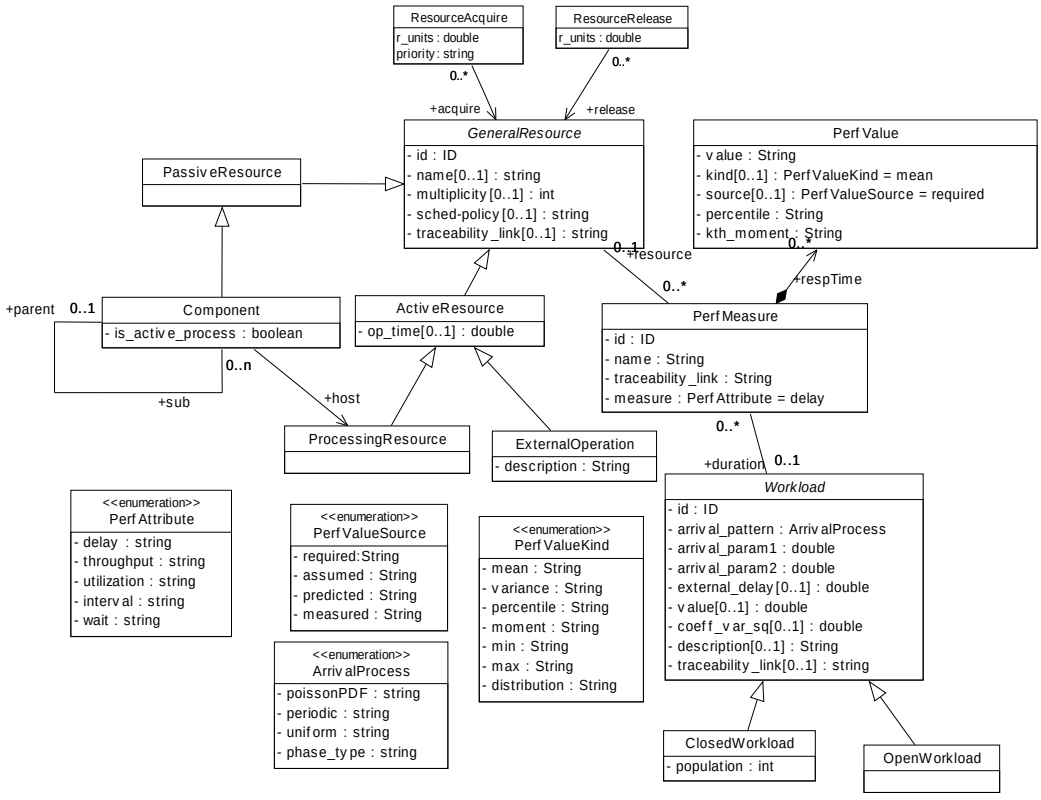


Figure 13 Performance package of CSM

3.4.1.3 Function package

The Function package in Figure 14 adds information about constraint and object flow. A constraint indicates that a restriction must be satisfied by means of precondition and/or postcondition. Input and output sets are collections of input and output tokens to/from

steps. Detail information about each class in this package can be found in the class description section.

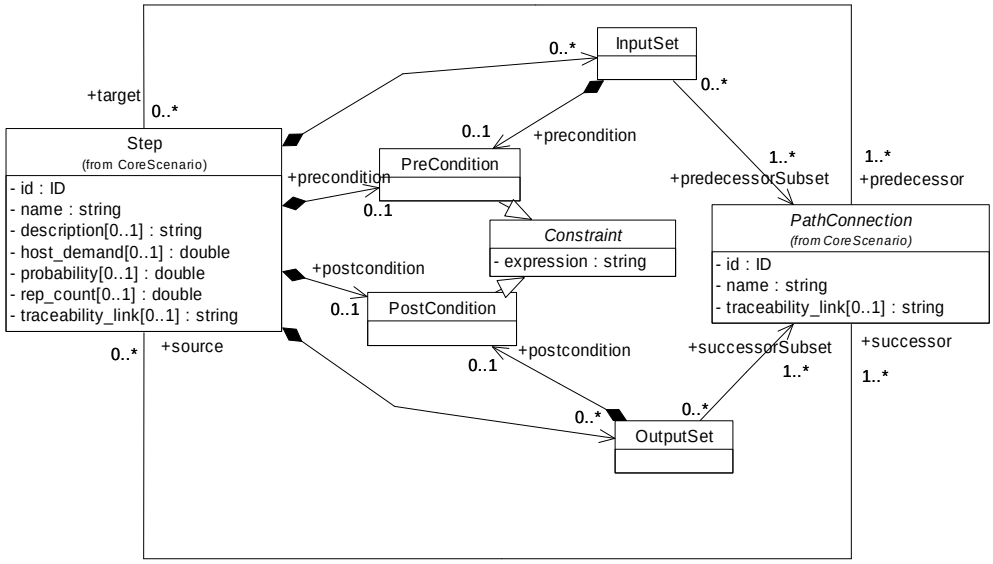


Figure 14 Function package of CSM

3.4.2 Class Description

3.4.2.1 CSM

Description

A Core Scenario Model (CSM) is the top-level element in the core scenario meta-model, which contains the core concepts from Use Case Maps (UCM) and UML 2.0 activity diagram and sequence diagram. It is composed of scenario model(s), and a collection of components, responsibilities, path connections, resources, and devices.

Attributes

Name: string [0..1]	the model name
Description: string [0..1]	detail information of the model
Author: string [0..1]	the model author
Created: date [0..1]	the date of creation
Version: string [0..1]	version number
Traceability_link: string [0..1]	links to the original model

Associations

Scenario: Scenario [*]	scenario contained in the model
Resource: GeneralResource [*]	resources contained in the model

Constraints

None

3.4.2.2 Scenario

Description

Scenario is the basic composition of CSM model. Scenario defines a group by specifying the elements that make up the scenario and the connections between elements.

Attributes

ID: string	the identity of Scenario
Name: string	the name of Scenario
Description: string [0..1]	the detail information of Scenario
Probability: double [0..1]	the probability of this scenario happens
Transaction: boolean [0..1]	indicates if scenario acts like a transaction
Traceability_link: string [0..1]	links to the original model

Associations

Sub: Step [*]	a collection of sub-scenario(s) owned by parent scenario
Path: pathconnection [*]	a collection of path connection(s) between scenario elements owned by parent scenario
Object: classifier [*]	a set of instances that have features in common
Refinement: refinement [*]	a collect ion of refined elements

Constraints

A scenario may not directly or indirectly own itself.

3.4.2.3 Component

Description

A component is the child of PassiveResource. Components represent abstract entities such as: actors, objects, resources, processes, containers, and so on.

Attributes

ID: string	the identity of component
Name: string [0..1]	the name of component
Multiplicity: int [0..1]	the property of being multiple
Schedule_policy: string [0..1]	the type of schedule policy

Traceability_link: string [0..1] links to the original model

Is_active_process: boolean [0..1] indication of active process

Associations

Host: ProcessingResource [*] the host of component

Parent: component [0..1] the parent of current component

Sub: component [*] the children of current component

Constraints

None

3.4.2.4 Step

Description

A Step describes the processing tasks (e.g. procedures, functions actions, etc.) that a system must perform. Steps declare the behavioural characteristic of instances of classifiers.

Attributes

ID: string the identity of step

Name: string the name of step

Description: string [0..1] detail information of step

Traceability_link: string [0..1] links to the original model

Host_demand: double [0..1] service request for the host

Probability: double [0..1] the percentage of service request

Rep_count: double [0..1] the count of repetitions

Associations

Predecessor: PathConnection [*] the predecessor of current step

Successor: PathConnection [*] the successor of current step

Component: Component [0..1] the component that contains current step

Parent: Scenario [0..1] the scenario that contains current step

Perfmeasuretrigger: PerfMeasure [0..1] the start point to measure the performance

Perfmeasureend: PerfMeasure [0..1] the end point to measure the performance

Resourceacquire: ResourceAcquire [0..1] the resource acquire action

Resourcerelease: ResourceRelease [0..1] the resource release action

Refinement: Refinement [0..1] a collection of binding connections to refine current step

Externaldemand: ExternalDemand [0..1] service request for an active resource

Precondition: PreCondition [0..1] constraint that must hold when the execution starts

Postcondition: PostCondition [0..1] constraint that must hold when the execution completes

Inputset: InputSet [0..1] a set of input values to the execution

Outputset: OutputSet [0..1] a set of result values from the execution

Constraints

None

3.4.2.5 ResourceAcquire

Description

A ResourceAcquire element defines an execution point on which the action can acquire the resource. It is a kind of step with additional attributes.

Attributes

R_units: double the number of resources being acquired

Priority: string the priority of acquiring action

Other attributes seen Step.

Associations

Acquire: GeneralResource resource that have been acquired

Constraints

[1] ResourceAcquire element can not acquire any resource that is not released.

[2] ResourceAcquire element must be paired with ResourceRelease element.

3.4.2.6 ResourceRelease

Description

A ResourceRelease element defines an execution point on which the action can release the resource. It is a kind of step with additional attributes.

Attributes

R_units: double the number of resources being released

Associations

None

Constraints

[1] ResourceRelease element can not release any resource that is not acquired.

[2] ResourceRelease element must be paired with ResourceRelease element.

3.4.2.7 Refinement

Description

A Refinement defines a connection between a single step and its referenced scenario(s). The execution of a single step may involve the execution of the complex scenario and its steps, in which case the single step acts like an invocation action in UML 2.0 activity diagram or stub in Use Case Maps (UCM). As a consequence, a complex scenario can be reused in many places as a single step.

Attributes

Selection_cond: string [0..1]	the boolean condition to select different scenarios

Associations

Parent: Step [1]	the invocation action
Sub: Scenario [1]	the referenced scenario
Inbinding: InBinding [*]	the input connection lists
Outbinding: OutBding [*]	the output connection lists

Constraints

One refinement can connect only one step and one scenario.

3.4.2.8 InBinding

Description

An InBinding joins an entry point of a step with a start point from a scenario.

Attributes

id: ID	the identity of an inbinding element

Associations

in: PathConnection [1]	the entry point of invocation action
start: Start [1]	the start point of the referenced scenario

Constraints

The InBinding can connect only pathconnection element and start element.

3.4.2.9 OutBinding

Description

An OutBinding joins an exit point of a step with an end point from a scenario.

Attributes

id: ID	the identity of an outbinding element

Associations

out: PathConnection [1]	the exit point of invocation action
end: End [1]	the end point of the referenced scenario

Constraints

The OutBinding can connect only pathconnection element and end element.

3.4.2.10 PathConnection

Description

A PathConnection element is an abstract class for the connections between two steps of a scenario.

Attributes

Id: ID	the identity of path connection element
Description: String [0..1]	detail information of path connection element
Traceability_link [0..1]	the link to original node

Associations

Source: Step [*]	step element from which the path connection starts
Target: Step [*]	step element at which the path connection ends
Classifier: Classfier [*]	classifiers can be attached to the path flow
subIn: InBinding [*]	the entry point to sub-scenario
subOut: OutBinding [*]	the exit point from sub-scenario

Constraints

[1] The source and target of a path connection element must be in the same scenario.

[2] The path connection can connect only steps.

3.4.2.11 Start

Description

A Start element is where scenario is invoked. A scenario may have more than one start element.

Attributes

See PathConnection element.

Associations

inBinding: InBinding [1]	the entry point of invocation action
workload: Workload [1]	the stream of requests attached to the scenario

Constraints

[1] A start element has no incoming edges.

[2] A start element must have only one target hyperedge.

3.4.2.12 End

Description

An End element is where scenario is stopped, where the post-conditions are satisfied. A scenario may have more than one end element.

Attributes

See PathConnection element.

Associations

outBinding: OutBinding [1]	the stop point of scenario

Constraints

[1] An end element has no target edges.

[2] An end element must have only one source hyperedge.

3.4.2.13 Sequence

Description

Sequence elements specify the sequencing of steps. A sequence element connects two steps as a source and a target.

Attributes

See PathConnection element.

Associations

No additional associations.

Constraints

[1] The source and target of a sequence element must be in the same scenario.

[2] The source and target of a sequence element must be steps.

3.4.2.14 Branch

Description

A Branch is an element that represents alternative paths. The conditions can be attached to branch elements so that a choice is determined when the condition is true.

Attributes

See PathConnection element.

Associations

No additional associations.

Constraints

The branch element must have only one source and two or more targets.

3.4.2.15 Merge

Description

A Merge is an element that represents the merging of two or more independent alternative paths. It is not used to synchronize concurrent paths but to accept one among several alternative paths.

Attributes

See PathConnection element.

Associations

No additional associations.

Constraints

The branch element must have only one target and two or more sources.

3.4.2.16 Fork

Description

A Fork is an element that splits a path into two or more concurrent paths.

Attributes

See PathConnection element.

Associations

No additional associations.

Constraints

The fork element must have only one source and two or more targets.

3.4.2.17 Join

Description

A Join is an element that synchronizes two or more concurrent scenario paths.

Attributes

See PathConnection element.

Associations

No additional associations.

Constraints

The join element must have only one target and multiple sources.

3.4.2.18 Classifier

Description

A Classifier describes a set of instances that have common features.

Attributes

ID: string	the identity of classifier
Name: string [0..1]	the name of classifier
Traceability_link: string [0..1]	links to the original model

Associations

None

Constraints

None

3.4.2.19 Message

Description

A Message is a kind of Classifier that defines the communications between steps.

Attributes

id: ID	the identity of a message
kind: Msgkind [0..1]	the type of message
size: double [0..1]	the size of message
multi: double [0..1]	the multiplicity

Associations

None

Constraints

None

3.4.2.20 GeneralResource

Description

A GeneralResource is an abstract class that represents the available source of servers.

Attributes

id: ID	the identity of general resource element
name: string	the name of general resource element
multiplicity: int [0..1]	the property of being multiple
shced_policy: string [0..1]	the type of schedule policy
Traceability_link: string [0..1]	links to the original model

Associations

perfMeasure: PerfMeasure [1]	the performance measurement attached to resource

Constraints

None

3.4.2.21 ActiveResource

Description

An ActiveResource represents the available server in the performance model.

Attributes

id: ID	the identity of active resource element
name: string	the name of active resource element
multiplicity: int [0..1]	the property of being multiple
shced_policy: string [0..1]	the type of schedule policy
traceability_link: string [0..1]	links to the original model
op_time: double [0..1]	service time

Associations

perfMeasure: PerfMeasure [1]	the performance measurement attached to resource

Constraints

None

3.4.2.22 PassiveResource

Description

A PassiveResource represents the resources that can be acquired and released.

Attributes

id: ID	the identity of passive resource element
name: string	the name of passive resource element
multiplicity: int [0..1]	the property of being multiple
shced_policy: string [0..1]	the type of schedule policy
traceability_link: string [0..1]	links to the original model

Associations

None

Constraints

None

3.4.2.23 ProcessingResource

Description

A ProcessingResource is a kind of ActiveResource that acts like processor.

Attributes

id: ID	the identity of processing resource element
name: string	the name of processing resource element
multiplicity: int [0..1]	the property of being multiple
shced_policy: string [0..1]	the type of schedule policy
traceability_link: string [0..1]	links to the original model

Associations

None

Constraints

None

3.4.2.24 ExternalOperation

Description

An ExternalOperation is a kind of ActiveResource that represents the services provided by external devices.

Attributes

id: ID	the identity of external operation element
name: string	the name of external operation element
multiplicity: int [0..1]	the property of being multiple
shced_policy: string [0..1]	the type of schedule policy
traceability_link: string [0..1]	links to the original model

Associations

None

Constraints

None

3.4.2.25 Workload

Description

A Workload is an abstract class that represents the load intensity applied to a scenario.

Attributes

id: ID	the identity of workload element
arrival_pattern: ArrivalProcess	the pattern in which requests arrive
arrival_param1: double [0..1]	the parameter for workload
arrival_param2: double [0..1]	the parameter for workload
external_delay: double [0..1]	the thinking time
value: double [0..1]	the value
coeff_var_sq: double [0..1]	the coefficient
description: string [0..1]	the detail information
traceability_link: string [0..1]	links to the original model

Associations

None

Constraints

None

3.4.2.26 OpenWorkload

Description

An OpenWorkload represents the streams of requests, which arrive at a given rate in some predetermined pattern.

Attributes

See Workload class

Associations

None

Constraints

None

3.4.2.27 CloseWorkload

Description

A CloseWorkload represents the fixed number of active users, which cycle between executing the scenario.

Attributes

Population: int	the number of active users

See Workload class

Associations

None

Constraints

None

3.4.2.28 PerfMeasure

Description

A PerfMeasure represents the measurement such as delay, throughput and utilization between executing the scenario.

Attributes

measure: perfAttribute	the type of measurement

Associations

Trigger: Step [0..1]	the step to start the performance measurement
End: Step [0..1]	the step to end the performance measurement
Duration: Workload [0..1]	the duration in time
Resource: GeneralResouce [0..1]	the resource involved in performance measurement

Constraints

[1] The trigger and end must be used in a pair.

[2] One of trigger, duration, or resource association is required.

3.4.2.29 PerfValue

Description

A PerfValue represents the estimated value, which is calculated by a performance analysis tool, required value that comes from system requirements, assumed value, which comes from experience, or a measure value.

Attributes

value: string	the calculated result
kind: PerfValuekind [0..1]	the type of result value
source: PerfValueSource [0..1]	the source of value
percentile: string [0..1]	the value of percentile
kth_moment: string [0..1]	the expected value of distribution

Associations

None

Constraints

None

3.4.3 XML Schema Generation

The CSM class diagrams in 3.4.1 represent the essential classes of our CSM meta-model and their relationships. They define the vocabulary we use in our transformation. However, in a practical transformation process, we need an interchange file format that provides a concrete syntax for the concepts in the meta-model. We use XML here as many XML tools are available. In addition, we propose an XML schema to validate our XML files.

An Eclipse plug-in tool called *HyperModel* [8] has been used to automatically generate an XML schema from the CSM UML class diagram, produced with Rational Rose Enterprise Edition. This tool enabled us to get a quick interpretation of CSM meta-model in XML format. We have slightly adapted the XML schema manually by relaxing the sequencing of some XML elements (e.g., CSMElement), to comply with validation tools (XML Spy and Xerces' parser). The resulting schema is presented in Annex A.

3.5 Benefits of CSM

The Core Scenario Model fits between software specification models and performance models. It separates the concerns of performance modeling from software design phases.

Software developers who do not have good knowledge of performance models can have a good understanding about the transformation to performance models through the CSM models. In fact, CSM shields the software developer and especially tool builders from knowing the different types of performance modeling languages, thereby reducing the number of transforming tools needed. CSM is designed to provide a uniform API to various performance models and design specifications. It hides differences and discrepancies between the software design modeling languages and the performance modeling languages. Therefore, the uniform API promotes a weak coupling between software specifications and performance models. One advantage is the fast production of transformation tools, and a side effect is that having more tools will enable simpler sensitivity analysis for the same software specification using different types of performance models.

Furthermore, CSM-based transformations can provide default values when the information required to perform analysis is not available at the current stage. Since the model-based performance analysis approaches can be applied to any phase of the software life cycle, sometime the information needed to performance analysis may not be obvious, especially in the early software requirement and design phase, in which case default values have to be provided to let the whole process carry out smoothly. Those default values may come from the experiments or from estimates based on experience.

Last but not least, CSM supports quick and meaningful design decisions as feedback from performance analysis results. There are direct correspondences between CSM and performance models. In CSM, we define some useful elements which relate directly to performance evaluation results, and those elements will support a simpler feedback process.

3.6 Compliance Statement

A product needs to be checked against its design, which needs to be check against its requirements. In this section we show how CSM complies with the requirements expressed earlier with a compliance table. To simplify the comparison and focus on the main issues, we only show the CSM classes and association relationships, and do not cover the attributes in each class. For the purpose of easy identification, we set the name of each CSM class in **Arial Black** characters and the association relationship in Arial Narrow characters. Table 2 shows the correspondences between the CSM requirements and the CSM definitions from the meta-model.

Table 2 Correspondences between requirements and implementations

Req. ID	Requirements Contents	CSM Links
3.3.1.1	capture actions	**Step**
3.3.1.2	refine action as a sequence of sub actions	**Refinement**
3.3.1.3	define communication message between actors	**Message**
3.3.1.4	define groups of actions	**Scenario**
3.3.1.5	define the sequence of actions	Source/target
3.3.1.6	temporal ordering of messages	ordered classifier
3.3.1.7	start of scenario	**Start**
3.3.1.8	end of scenario	**End**
3.3.1.9	merge concurrent sequences	**Merge**
3.3.1.10	merge alternative sequences	**Merge**
3.3.1.11	split one sequence as two or more concurrent sequences	**Branch**
3.3.1.12	split one sequence as two or more alternative sequences	**Branch**
3.3.1.13	specify branch conditions	**Conditions**
3.3.1.14	specify repetitive actions	**Loop**
3.3.1.15	specify number of iterations for each repetitive action.	**Conditions**
3.3.1.16	specify guarding conditions for each repetitive action	**Conditions**
3.3.1.17	group related scenarios	**Scenario/Step**
3.3.1.18	abstract related scenarios into one action	**Step**
3.3.1.19	to reference a group of related scenarios	**Step**
3.3.1.20	specify preconditions at scenario start points	**Precondition**
3.3.1.21	specify post-conditions at scenario end points	**Postcondition**
3.3.1.22	specify preconditions at the entry points of a group scenario	**Precondition**
3.3.1.23	specify post-conditions at the exit points of a group scenario	**Postcondition**
3.3.2.1	define actors	**Component**

Req. ID	Requirements Contents	CSM Links
3.3.2.2	allocate actions to actors	Component
3.3.2.3	containment relationship between actors	Sub/parent
3.3.3.1	define hardware devices	**ProcessingResource**
3.3.3.2	define software services	**ExternalOperation**
3.3.3.3	define workload attributes	**CloseWorkload/ OpenWorkload**
3.3.3.4	Stream request for open workload	**ArrivalProcess**
3.3.3.5	define number of jobs for close workload	Attributes of **Workload**
3.3.3.6	define think time for close workload	Attribute of **Workload**
3.3.3.7	response time for each scenario	**PerfValue/PerfMeasure**
3.3.3.8	throughput for each scenario	**PerfValue/PerfMeasure**
3.3.3.9	number of executing times for a step	Attribute of **Step**
3.3.3.10	execution time on host device	Attribute of **Step**
3.3.3.11	define resource acquiring points	**ResourceAcquire**
3.3.3.12	service time of acquired resource	Attribute of **Step**
3.3.3.13	units of resource being required	Attribute of **ResourceAcquire**
3.3.3.14	define resource releasing points	**ResourceRelease**
3.3.3.15	units of resource being released	Attribute of **ResourceRelease**
3.3.4.1	allocate component to processing resource	Host
3.3.4.2	allocate resources to software components	**ExternalOperation**
3.3.5.1	define performance requirement values	**PerfValue**
3.3.5.2	record elapsed time for each scenario	**PerfValue**
3.3.5.3	utilization of hardware resource	**PerfValue**
3.3.5.4	utilization of software resource	**PerfValue**
3.3.5.5	Trace back to resource element	Attribute of **GeneralResource**
3.3.5.6	Trace back to scenarios	Attribute of **Scenario**

3.7 Chapter Summary

This chapter defines a Core Scenario Model (CSM) as an interface between design models and performance models. CSM promotes a loose coupling between design specifications and performance modeling languages, thus helping to increase transformation flexibility. With this CSM, we are ready to support many transformations from different design specifications to CSM and from CSM to different performance models. In the rest of this thesis, we will focus on the transformation from UCM design specifications to CSM models.

4. Transformation

The automatic integration of software specification to performance models is a key factor for the applicability of early performance analysis. As discussed in chapter 3, we developed a CSM meta-model as a unified interface for integrating software design specifications and performance models. There are two phases involved in such integration. The first phase refers to the mappings from the design models to the CSM representation, and the second phase is concerned with transformations from a CSM model to performance models. In this chapter, we give a detailed mapping between UCM specifications and CSM models and present algorithms to automatically transform UCM to CSM.

4.1 Mapping from UCM to CSM

A mapping from UCM to CSM is the process of finding out the elements that have equivalent concepts between the two different models. Those elements are called *correspondences.* We divide the correspondences into two groups: elements having *explicit* mappings and others having *implicit* mappings. In this section, we discuss these correspondences and the transformation strategy in detail.

4.1.1 Explicit Mappings

There are some constructs in the UCM and CSM notations which have one-to-one mapping relationships. These constructs are the basic building blocks with simple semantics. The following sub-sections provide the detailed mappings of these classes and their attributes.

Annotated UCM design model also contains the required performance attributes for performance analysis. These attributes, such as service request, workload, and operation time of devices, can be mapped to CSM classes and attributes. We can check the completeness of these required attributes using one function called “check annotation completeness” in the UCMNav. If the required performance attributes are empty in the UCM design model, the default values are filled in by our transformation. The following sub-sections also list the required performance attributes and default values for transformation.

4.1.1.1 UCM-design Class

The UCM-design class is the root element in the UCMNav meta-model, and it can be mapped to the root element in the CSM meta-model. Some attributes in CSM, such as author, created and version, are not available in UCM, but they are optional, so no default values are provided. Containment associations are mapped to the same containment structures in CSM, i.e. all the contained classes are enclosed between the corresponding XML opening tag and closing tag. Details will be covered in forthcoming sub-sections. The detailed mappings are as follows:

Table 3 UCM-design mapping table

	UCM	***CSM***
Class	UCM-design	CSM
Attributes	design-id design-name description N/A	traceability_link name description author, created, version (*optional*)
Associations	details covered later	details covered later

4.1.1.2 Model Class

The model class in the UCMNav meta-model is mapped directly to scenario class in the CSM meta-model. The required attribute called transaction is not available in UCM and a default *false* value is provided. The detailed mappings are as follows:

Table 4 Model class mapping table

	UCM	***CSM***
Class	model	scenario
Attributes	model-id model-name description title N/A N/A	traceability_link name description *ignored* transaction = false probability (*optional*)
Associations	submap	scenario class

4.1.1.3 Component Class

Component definitions in the UCM model are not transferred directly to CSM. However, UCM component *references* are explicitly mapped to CSM components. The required attribute called is-active-process is not available in UCM, but its value can be inferred from the UCM component type (teams, processes, and agents are active, the other types are not). The detailed mappings are as follows:

Table 5 Component class mapping table

	UCM	***CSM***
Class	component-ref	component
Attributes	component-ref-id component-name description role, anchored, fixed referenced-component→type N/A N/A	traceability_link name description *ignored* is-active-process = true if type is process, team or agent (and false otherwise) multiplicity (*optional*) sched-policy (*optional*)
Associations	component-parent component-child referenced-component→device	parent sub host

4.1.1.4 Device Class

The device class in the UCMNav meta-model is mapped directly to the processingresouce class in the CSM meta-model. The type of the device is ignored in CSM. Two optional attributes are not available in UCM and no default values are provided. The detailed mappings are as follows:

Table 6 Device class mapping table

	UCM	***CSM***
Class	device	processingresource
Attributes	device-id device-name	traceability_link name

	description operation-time device-type N/A N/A	description operation-time *ignored* multiplicity (*optional*) sched-policy (*optional*)
Associations	N/A	N/A

4.1.1.5 Responsibility-ref Class

The responsibility-ref class in the UCMNav meta-model is mapped directly to the step class in the CSM meta-model. The service request association in UCM is mapped to the host-demand attribute of the step class in CSM. Two optional attributes are not available in UCM and no default values are provided. The detailed mappings are as follows:

Table 7 Responsibility-ref class mapping table

	UCM	***CSM***
Class	responsibility-ref	step
Attributes	hyperedge-id hyperedge-name description direction arrow-position N/A N/A	traceability_link name description *ignored* *ignored* probability (*optional*) rep-count (*optional*)
Associations	service-request hyperedge-connection class hyperedge-connection class	host-demand attribute predecessor successor

4.1.1.6 Start Class

The start class in the UCMNav meta-model is mapped directly to the start class in the CSM meta-model. The stream-type attribute determines whether an openworkload class or a closeworkload class from CSM will be created. The arrival attribute is mapped to the arrival-pattern attribute of the workload class in the following way

- exponential → PoissonPDF (exponential not supported by CSM)
- deterministic → PoissonPDF (deterministic not supported by CSM)

- uniform → uniform
- Erlang → PoissonPDF (Erlang not supported by CSM)
- expert → PoissonPDF (user-defined arrivals not supported by CSM)

The detailed mappings are as follows:

Table 8 Start class mapping table

	UCM	***CSM***
Class	start	start
Attributes	hyperedge-id	traceability_link
	hyperedge-name	name
	description	description
	arrival	arrival-pattern of workload class (PoissonPDF or uniform)
	stream-type == open	open workload class
	stream-type == closed	close workload class
	logical-condition	*ignored*
	population-size	population of closeworkload class
	mean	arrival_param1 of workload class
	value	arrival_param1 of workload class
	low	arrival_param1 of workload class
	high	arrival_param2 of workload class
	kernel	arrival_param2 of workload class
	expert-distribution	*ignored*
	label-alignment	*ignored*
Associations	covered in plugin-binding class	inbinding
	hyperedge-connection class	source
	hyperedge-connection class	target

4.1.1.7 End-point Class

The end-point class in the UCMNav meta-model is mapped directly to the end class in the CSM meta-model. The detailed mappings are as follows:

Table 9 End-point class mapping table

	UCM	***CSM***

Class	end-point	end
Attributes	hyperedge-id hyperedge-name description label-alignment	traceability_link name description *ignored*
Associations	covered in plugin-binding class hyperedge-connection class hyperedge-connection class	outbinding source target

4.1.1.8 Or-Fork Class

The or-fork class in the UCMNav meta-model is mapped directly to the branch class in the CSM meta-model. At the moment, conditions and probabilities associated with branches of an or-fork are not mapped to CSM. The detailed mappings are as follows:

Table 10 Or-fork class mapping table

	UCM	***CSM***
Class	or-fork	Branch
Attributes	hyperedge-id hyperedge-name description orientation	traceability_link name description *ignored*
Associations	hyperedge-connection class hyperedge-connection class	source target

4.1.1.9 And-Fork Class

The and-fork class in the UCMNav meta-model is mapped directly to the fork class in the CSM meta-model. The detailed mappings are as follows:

Table 11 And-fork class mapping table

	UCM	***CSM***
Class	and-fork	Fork
Attributes	hyperedge-id hyperedge-name description	traceability_link name description

	orientation	*ignored*
Associations	hyperedge-connection class hyperedge-connection class	source target

4.1.1.10 Or-Join Class

The or-join class in the UCMNav meta-model is mapped directly to the merge class in the CSM meta-model. The detailed mappings are as follows:

Table 12 Or-join class mapping table

	UCM	***CSM***
Class	or-join	merge
Attributes	hyperedge-id hyperedge-name description orientation	traceability_link name description *ignored*
Associations	hyperedge-connection class hyperedge-connection class	source target

4.1.1.11 And-Join Class

The and-join class in the UCMNav meta-model is mapped directly to the join class in the CSM meta-model. The detailed mappings are as follows:

Table 13 And-join class mapping table

	UCM	***CSM***
Class	and-join	Join
Attributes	hyperedge-id hyperedge-name description orientation	traceability_link name description *ignored*
Associations	hyperedge-connection class hyperedge-connection class	source target

4.1.1.12 Empty-point Class

The empty-point class in the UCMNav meta-model is mapped directly to the sequence class in the CSM meta-model. The detailed mappings are as follows:

Table 14 Empty-point class mapping table

	UCM	***CSM***
Class	empty-point	sequence
Attributes	hyperedge-id hyperedge-name description	traceability_link name description
Associations	hyperedge-connection class hyperedge-connection class	source target

4.1.1.13 Stub Class

The stub class in the UCMNav meta-model is mapped directly to the step class in the CSM meta-model. The detailed mappings are as follows:

Table 15 Stub class mapping table

	UCM	***CSM***
Class	stub	step
Attributes	hyperedge-id hyperedge-name description type shared selection-policy N/A N/A N/A	traceability_link name description *ignored* *ignored* *ignored* host-demand =1 probability (*optional*) rep-count (*optional*)
Associations	hyperedge-connection class hyperedge-connection class	predecessor successor

4.1.1.14 Plugin-binding Class

The plugin-binding class in the UCMNav meta-model is mapped directly to the refinement class in the CSM meta-model. The detailed mappings are as follows:

Table 16 Plugin-binding class mapping table

	UCM	***CSM***
Class	plugin-binding	refinement
Attributes	plugin-binding-id hyperedge-name description branch-condition probability	id name description selection-condition (*optional*) *ignored*
Associations	parent submap	parent sub

4.1.1.15 In-connection Class

The in-connection class in the UCMNav meta-model is mapped directly to the inbinding class in the CSM meta-model. The detailed mappings are as follows:

Table 17 In-connection class mapping table

	UCM	***CSM***
Class	in-connection	inbinding
Attributes	N/A	N/A
Associations	stub-entry class start class in submap	in start class

4.1.1.16 Out-connection Class

The out-connection class in the UCMNav meta-model is mapped directly to the outbinding class in the CSM meta-model. The detailed mappings are as follows:

Table 18 Out-connection class mapping table

	UCM	***CSM***
Class	out-connection	outbinding
Attributes	N/A	N/A

Associations	stub-exit class end class in submap	out end class

4.1.1.17 Other Classes

Other classes in the UCMNav meta-model are ignored, because they are concepts unique to UCMs (such as graphic representation classes) that have no corresponding classes defined in CSM.

4.1.1.18 Explicit Mapping Example

In this sub-section, we demonstrate how to use the previous transformation rules to transform a simple UCM model to its CSM representation. Figure 15 shows a UCM model with one simple plugin submap. We show the contents in terms of the UCMNav meta-model concepts, apply the transformation rules, and show the transformation results in terms of CSM meta-model concepts. The corresponding CSM output file is shown in Annex C.

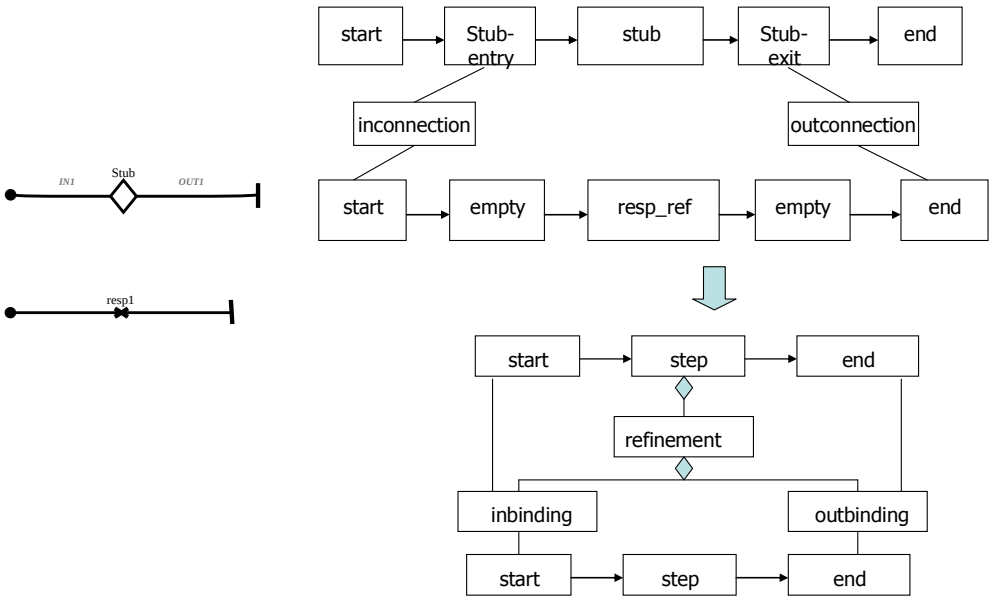


Figure 15 Explicit mapping example

4.1.2 Implicit Mappings

There are some elements defined in CSM with no direct mappings from UCM, such as resource acquire and resource release elements. Although there are no explicit definitions

in UCM for those elements, the UCM paths capture some of this information implicitly. This section discusses the indirect conversions required to support implicit mappings.

4.1.2.1 ResourceAcquire Element

Resources are acquired as scenarios evolve. In CSM, we define the ResourceAcquire element to capture the resource acquisition action. Acquiring resources may require one or many ResourceAcquire steps. The acquisition of a resource happens before we can perform operations on it, and this resource is held for a period of time (until it is released). We explicitly define the start of the holding time by using the ResourceAcquire element.

In UCM, there is no explicit element to define the start of resource holding time. This kind of information is however implied by the UCM paths (Figure 16). When the incoming path is entering the boundary of one component, this component is acquired implicitly. We define a path from hyperedge A to hyperedge B as an *incoming path* by comparing the enclosing components of A and B. Hyperedge A must have no enclosing component or a different enclosing component from that of hyperedge B. In most cases, there is an empty point hyperedge involved as A or B because of the internal structure of UCMNav.

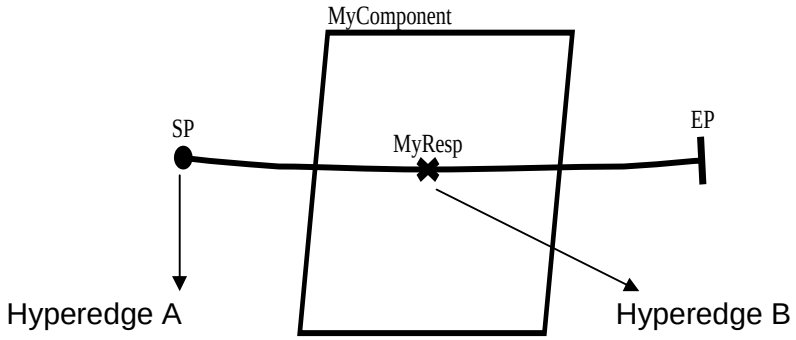


Figure 16 Incoming Path

When there is an incoming path crossing the boundary of a component, a ResourceAcquire element is inserted in the CSM model and the acquired component is set as the attribute of the ResourceAcquire element in the corresponding XML file.

4.1.2.2 ResourceRelease Element

In CSM, we define the ResourceRelease element to capture the resource release action. Releasing resources may require one or many ResourceRelease steps. The release of a

resource happens after we have performed operations on it. We explicitly define the end of the holding time by using the ResourceRelease element.

In UCM, there is no explicit element to define the end of resource holding time. This kind of information can again be inferred from the UCM paths (Figure 17). When the outgoing path is leaving the boundary of one component, this component is released implicitly. We define a path from hyperedge A to hyperedge B as an *outgoing path* by comparing the enclosing components of A and B. Hyperedge B must have either no enclosing component or an enclosing component different from but not included in that of hyperedge A. In most cases, there is an empty point hyperedge involved as A or B because of the internal structure of UCMNav.

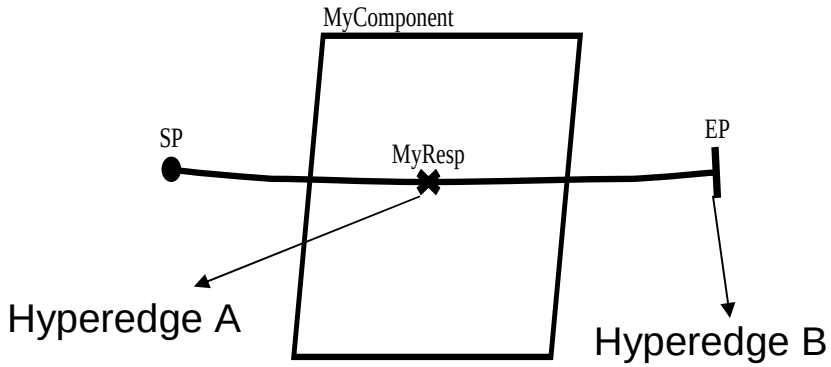


Figure 17 Outgoing Path

When there is an outgoing path crossing the boundary of one component, a ResourceRelease element will be inserted in the CSM model, and the corresponding output file will have the released component set as the attribute of the ResourceRelease element.

4.1.2.3 Implicit Mapping Example

We show in Figure 18 the transformation process in terms of meta-model concepts for the previous examples (Figure 16 and Figure 17).

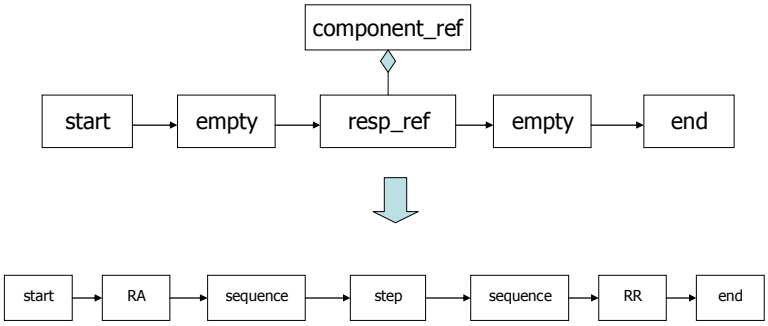


Figure 18 Implicit mapping example

4.1.2.4 Discussion on Resource Acquisition and Release

According to the definitions of ResourceAcquire and ResourceRelease elements, we can see that there are no such explicit element definitions in UCM notation. In the CSM meta-model, ResourceAcquire and ResourceRelease elements are steps, and have attributes such as ID and name together with predecessor and successor associations (which also map to attributes in the XML schema). Since these kinds of attributes are not included in the UCM input model, we have to add them during our transformation.

It is hard to add that information without changing some of the links found in the original UCM model. The reason is that the insertion of resource acquire element or resource release element will inevitably change the flow relationship along the path. For example, we refine Step1 in Figure 19 to include the resource acquire element RA and resource release element RR. This refinement will have side effects on Step1's predecessor (Start point) and successor (End point). The successor of the Start point is changed from Step1 to RA, whereas the predecessor of the End point is changed to RR. The transformation strategy will discuss how this is achieved in our prototype tool.

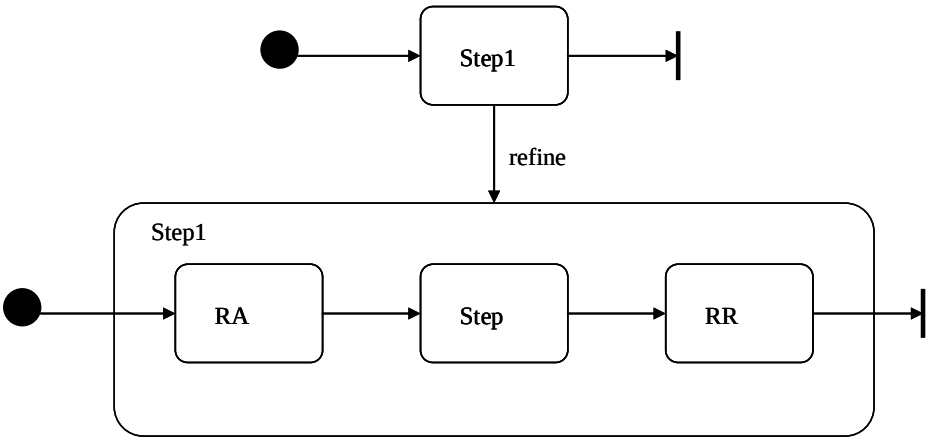


Figure 19 ResourceAcquire and ResourceRelease as independent Steps

4.1.3 Transformation Strategy

To transform these direct and indirect mapping elements into CSM elements, we have developed a set of transformation algorithms. Having defined the correspondences between UCM and CSM models, the transformation goal now consists in applying a programming methodology to automate the transformation process. The input to the transformation tool will be a UCM object model (read by UCMNav from a XML file), and the output will be an XML representation of the corresponding CSM model, which will be valid with respect to the CSM schema. In this section, we list the transformation algorithms for explicit and implicit mappings and the assumptions for each algorithm.

4.1.3.1 Direct Mappings

By applying the transformation rules in section 4.1.1 and its sub-sections, we can transform direct mapping elements to CSM representations. The process is similar to saving a UCM object model to an XML file, as currently done in UCMNav. However, we need to generate the XML CSM representation rather than the UCMNav representation (for which a DTD exists [31]). This is hence a syntactic transformation of the UCM model. Section 4.1.1.1 to section 4.1.1.16 list the corresponding classes and attributes from UCM to CSM representations. The containment relationships are mapped to the same structures in CSM. That means that in the output XML document all the contained elements are enclosed by the containing element. If an element's attributes required by CSM schema are not presented in the UCM design model, the transformation algorithm will generate the default values for those required attributes. In the most cases, these required attributes are necessary for performance model analysis, and the common default value used in performance modeling is a unit. The direct mapping procedure can be described generically as follows:

For each direct mapping element in the UCM model:

1. Generate the CSM XML opening tag corresponding to the element.
2. Generate the XML attributes of the element as required by the CSM schema.
3. Generate the closing tag.

4.1.3.2 Resource Acquisition Algorithm

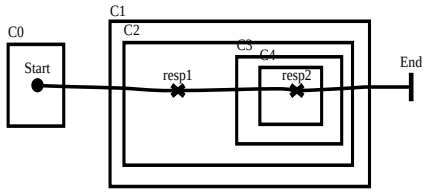
A resource acquisition occurs when an incoming UCM path crosses the border of a component. From the definition of the incoming path, we know that the first hyperedge resides in a component different from the second hyperedge's, or has no enclosing component. The component(s) containing the second hyperedge will be acquired. Therefore, the

acquisition always happens at the second hyperedge. This information can help us express our algorithm in simple terms. Instead of checking every hyperedge in a model, we only need to focus on the edges that have enclosing components.

The input to the algorithm is a hyperedge in UCM model, and the algorithm updates the set of resource acquire elements. The resource acquisition (RA) algorithm is described on the next page. It makes use of a *FindParentsRA* utility algorithm that updates a stack of containing parents, with the outermost one at the top of the stack.

Figure 20 shows an example for resource acquisition. There are four stacks used to store the enclosing components for each hyperedge. The difference between two stacks describes the components to be acquired. The transformation results are also shown.

Start_stack: C0
Resp1_stack: C2, C1
Resp2_stack: C4,C3,C2,C1
End_stack: null



Start->RR(C0)->Seq ->reps1->Seq->resp2->Seq->RR(C4)->Seq->RR(C3)->Seq->RR(C2)->Seq->RR(C1)->End

Figure 20 RA example

Name: FindParentsRA Algorithm
Input: A component
Output: None
Modified: A stack of components

```
FindParentsRA (current: Component, modified s:Stack): void
// The top of the stack is the outermost component
{
    p:Component = parent of current
    if (p <> null)
    {
        s.push(p)
        FindParentsRA(p, s)
    } // if
}
```

Name: Resource Acquisition (RA) Algorithm

Input: A hyperedge
Output: None
Modified: Hypergraph containing the hyperedge

```
RA(curr_edge: Hyperedge, modified map:Hypergraph): void
{
    // Current edge must be in a component
    if (curr_edge is inside a component)
    {
        prev_edge_list: HyperedgeList = previous hyperedges of curr_edge
        curr_edge_comp: Component = the innermost component of curr_edge
        curr_edge_s: Stack = null
        curr_edge_s.push(curr_edge_comp)
        FindParentsRA(curr_edge_comp, curr_edge_s)

        if prev_edge_list is not empty then
        {
            forall prev_edge: Hyperedge in prev_edge_list
            {
                // Previous edge must be in a different component
                prev_edge_comp: Component = the innermost component of prev_edge
                if (prev_edge_comp <> curr_edge_comp)
                {
                    // Find which parents of curr_edge are not included in those of prev_edge
                    prev_edge_s: Stack = null
                    prev_edge_s.push(prev_edge_comp)
                    FindParentsRA(prev_edge_comp, prev_edge_s)
                    // Difference, keeps outside components
                    outside_comp_s: Stack = curr_edge_s – prev_edge_s

                    // Acquire the components of the parents
                    while outside_comp_s is not empty
                    {
                        comp: Component = outside_comp_s.pop
                        acquire comp // Create a ResourceAcquire and an empty point
                    } // while

                } // if
            } // forall
        else
        {
            // Must be a start point, acquire the component(s)
                while curr_edge_s is not empty
                {
                    comp: Component = curr_edge_s.pop
                    acquire comp // Create an empty point and a ResourceAcquire
                } // while

        } // else
    } // if
}
```

4.1.3.3 Discussion of Resource Acquisition Algorithm

Our resource acquisition algorithm is based on the idea of incoming path (crossing a component), and we always check both the current hyperedge and the previous hyperedge to determine if that path is an incoming path. This mechanism may imply that we have to follow a predefined sequence to cover all the hyperedges in the UCM model, but our resource acquisition algorithm does not require any specific strategy to cover all the hyperedges.

The reason is that we will apply the RA algorithm to each hyperedge in the UCM model once. Every time there is a resource acquisition action, we insert a RA object between the previous hyperedge and current hyperedge, without having to check if components in the previous hyperedge have been acquired. After we cover all the hyperedges, the correct flow relationship along the path is built in the UCM model. That is why we can only focus on the current edge and the previous hyperedge, and do not need to consider the edges before the previous hyperedge or after the current hyperedge.

One assumption is that resource acquire actions can happen at start point edges. A start point has no predecessor, but the resource acquire action in our algorithm is, in fact, the comparison of difference in components between two consecutive edges. If all start point edges are inside components, those components need to be acquired all the time, which is a situation covered by the else clause in the RA algorithm.

Resource acquire actions occur when the incoming path is crossing the boundary of one component no matter if there is any responsibility inside that component (as long as a hyperedge is included). There are several shortcomings with this incoming path view. First, it is unclear who the primary user of this resource is and why this user needs to acquire it. Second, it is also difficult to define the incoming path and relate it to resource acquire elements if no hyperedge is included (and modellers may or may not take this into consideration). Empty points are valid hyperedges and can be used in such situation, but they have no meaning in CSM, and may lead to unnecessary RA/RR actions. Instead we can explicitly make use of a step (i.e., responsibility) element.

We could hence make one more assumption, where resource acquisition actions cannot happen when there is no responsibility or stub inside a component. A resource acquire element in CSM actually is a child of step element, and no other CSM element can acquire resources or components. The reason why we could want such an assumption is

because resource operations need to be triggered by some activities or actors. Without any activity happening, there is no need for resources to be acquired.

If the same component happens to be present in parent map and plug-in map, the current resource acquire algorithm will acquire the same component twice. UCMs allow the same component to be present in both a parent map and its sub maps, and that component does not need to be acquired again in sub maps since it is already acquired in the parent map. The current RA algorithm is based on the incoming path crossing the border of components, and has difficulty to detect if the current component is already acquired in some other places.

The potential solutions for this situation are the following:

- We could provide design guidelines for UCM modellers to prevent this kind of situation from happening.
- We could require plug-in maps to always be in the same component context or be used only once.
- We could also duplicate the plug-in maps in case the parent maps contexts are different; each RA action would hence acquire a copy of the same component.
- We could extend the UCM notation with a concept describing explicitly a “parent component”, to be used in plug-in maps. This parent component would not need to be re-acquired.

4.1.3.4 Resource Release Algorithm

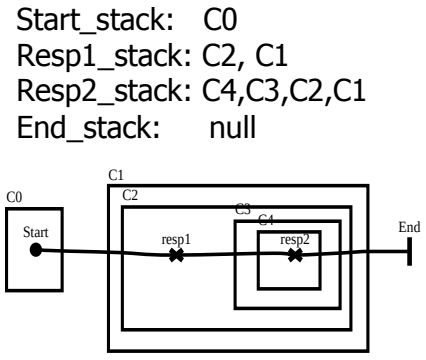
Resource releasing action happens when the outgoing UCM path crosses the border of a component. From the definition of an outgoing path, we know that the first hyperedge is inside a component and the second hyperedge is not inside the same component (or its ancestors) or is not bound to any component. Therefore, the releasing action can happen once we know the components enclosing the second, and the components that enclose the first hyperedge will be the candidates to be released. This information helps us create a simple algorithm. Instead of systematically checking pairs of consecutive hyperedges on a path and find their parents, we only need to focus on first edges that have an enclosing component. If a first hyperedge does not reside in any component, there will be no resource to release.

The resource releasing algorithm (RR) is described on the next page. It makes use of a *FindParentsRR* utility algorithm that updates a stack of containing parents, with the innermost one at the top of the stack:

Name: FindParentsRR Algorithm
Input: A component
Output: None
Modified: A stack of components

```
FindParentsRR (current: Component, modified s:Stack): void
// The top of the stack is the outermost component
{
    p:Component = parent of current
    if (p <> null)
    {
       FindParentsRR(p, s)
       s.push(p)
   } // if
}
```

Figure 21 shows an example for resource release. There are four stacks used to store the enclosing components for each hyperedge. The difference between two stacks describes the components to be released. The transformation results are also shown.



Start->RR(C0)->Seq ->reps1->Seq->resp2->Seq->RR(C4)->Seq->RR(C3)->Seq->RR(C2)->Seq->RR(C1)->End

Figure 21 RR example

Name: Resource Release (RR) Algorithm
Input: A hyperedge
Output: None
Modified: Hypergraph containing the hyperedge

```
RR(curr_edge: Hyperedge, modified map:Hypergraph): void
{
    // Current edge must be in a component
    if (curr_edge is inside a component)  then
    {
        next_edge_list: HyperedgeList = next hyperedges of curr_edge
        curr_edge_comp: Component = the innermost component of curr_edge
        curr_edge_s: Stack = null
        FindParentsRR(curr_edge_comp, curr_edge_s)
        curr_edge_s.push(curr_edge_comp)

        if next_edge_list is not empty then
        {
            forall next_edge: Hyperedge in next_edge_list
            {
                // Next edge must be in a different component
                next_edge_comp: Component = the innermost component of next_edge
                if (next_edge_comp <> curr_edge_comp)
                {
                    // Find which parents of next_edge are not included in those of curr_edge
                    next_edge_s: Stack = null
                    FindParentsRR(next_edge_comp, next_edge_s)
                    next_edge_s.push(next_edge_comp)
                    // Difference, keeps outside components
                    outside_comp_s: Stack = curr_edge_s – next_edge_s

                    // Release the components
                    while outside_comp_s is not empty
                    {
                        comp: Component = outside_comp_s.pop
                        release comp // Create an empty point and a ResourceRelease
                    } // while
                } // if
            } // forall
        } // if
        else
        {
            // Must be an end point, release the component(s)
            release curr_edge_comp // Create a ResourceRelease and an empty point
                while curr_edge_s is not empty
                {
                    comp: Component = curr_edge_s.pop
                    release comp // Create a ResourceRelease and an empty point
                } // while
        } // else
    } // if
```

4.1.3.5 Discussion of Resource Release Algorithm

In the resource release algorithm described above, we assume that resource release action only happens at the specific point, i.e. only after we get the information about the enclosing component(s) of the next hyperedge. This assumption helps us to simplify the resource release algorithm. The current hyperedge must have at least one enclosing component, otherwise, there is nothing to release. In our implementation, we use this condition to accelerate the process we cover all the hyperedges. We also need to know the difference of components between the current hyperedge and the next hyperedge, and then the differences, which must enclose the current hyperedge, will be released. There is no other easy ways to get the difference except finding out all the components and comparing them. But we can limit the times we do such calculations by focusing on the consecutive edges that have different enclosing components. If two consecutive edges are in the same component, we simply ignore the first hyperedge.

Another assumption is that resource release actions can happen at end point edges. An end point has no successor, but the resource release action in our algorithm is, in fact, the comparison of difference in components between two consecutive edges. If an end point is inside a hierarchy of components, then these components will be released by the else clause in the RR algorithm.

The position of the empty point in the UCM design model can determine where the resource release action is. In UCMNav, every time we add a responsibility, one or two empty point edges are automatically inserted at the default positions. The default position may be inside the same component as the current hyperedge, in which case the border-crossing event will not be detected and resource release action can not happen. After allowing the empty point hyperedge to release resource, we can release the acquired resource in the next hyperedge (most likely is an empty point hyperedge) if the current hyperedge and the next hyperedge happen to be in the same component.

If the same component happens to be present in parent map and plug-in map, the current resource release algorithm will release the same component twice. UCM allows the same component present in both parent map and sub maps, and that component needs not to be released again in parent maps since it is already released in the sub map. The current RR algorithm is based on the outgoing path crossing the border of components, and has difficulty to detect if the current component is already released in some other places.

The potential solutions for this situation are: we could provide design guidelines for UCM designers to prevent this kind of situation happen. The plug-in maps should always be in the same component context or be used only once; we could also duplicate the plug-in maps in case the parent maps contexts are different, each RR will release a copy of the same component.

4.1.3.6 SaveCSMtoXML Algorithm

Before applying GenerateCSM(ucm) algorithm, we have duplicated the original UCM model. We make changes to the duplicated UCM model by inserting RA/RR objects where required. The purpose is to easily maintain the flow relationship properly in the output XML file, and after we generate the XML file, we can simply discard the intermediate model so created.

Name: SaveCSMtoXML Algorithm
Input: A UCM model
Output: A XML file containing the CSM model

```
SaveCSMtoXML (ucm:UCM): file
{
   DuplicateHyperedges(ucm) // Create an intermediate model based on the original one
   Transform(ucm) // Add resource acquire/release/dummy resp
   SaveXML(ucm)  // Generate XML tags for all elements in the intermediate model
   DeleteDuplicateModel(ucm)  // Remove the intermediate model
}
```

4.1.3.7 Transform Algorithm

For each hyperedge in the UCM model, we need to check if it is in an incoming path or an outgoing path. Therefore, we apply RA/RR algorithm as a pair. We also apply a normalization algorithm to remove redundant empty points and add required dummy step.

Name: Transform Algorithm
Input: A UCM model
Output: None
Modified: The UCM model, with additional hyperedges for RA, RR, dummy resp.

```
AddRaRr (modified ucm:UCM): void
{
    forall map:Hypergraph in ucm
    {
       forall hyperedge:Hyperedge in map
       {
          RA(hyperedge, map)  // Add resource acquire if relevant
          RR(hyperedge, map)  // Add resource release if relevant
       }
       Normalize(map) // Remove redundant empty points & add required dummy resp.
```

```
    }
}
```

4.1.3.8 Normalize Algorithm

The Normalize algorithm modifies the duplicated UCM model to remove redundant empty points and to insert dummy steps where required. The reason is that in CSM a pathconnection element cannot connect directly to another pathconnection element, and a step element cannot connect directly to another step element. Our normalize algorithm could be extended to cover some of unmapped elements (see elements not mapped). Figure 22 shows two normalize examples.

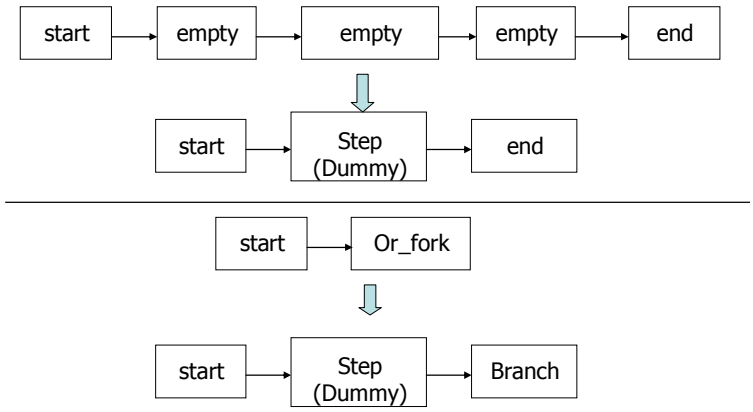


Figure 22 Normalize examples

Name: Normalize Algorithm

Input: A hypergraph
Output: None
Modified: The hypergraph, possibly with dummy responsibilities and fewer empty points

```
Normalize (modified map:Hyperedge): void
{
    forall hyperedge:Hyperedge in map
    {
        if hyperedge is an EmptyPoint then
        {
            prev:Hyperedge = hyperedge.previous
            while prev is an EmptyPoint
            {
                secondprevious:Hyperedge = prev.previous
                remove(prev)  // Remove previous empty point and adjust links
                prev = secondprevious
            } // while

            next:Hyperedge = hyperedge.next
            while next is an EmptyPoint
            {
                secondnext:Hyperedge = next.previous
                remove(next)  // Remove next empty point and adjust links
                next = secondnext
            } // while

            if prev is a ResponsibilityRef or a Stub then
            {
                if next is a ResponsibilityRef or a Stub then
                    // Keep this empty point. Will become a Seq in the CSM.
                else
                    remove(hyperedge)  // Remove current empty point and adjust links
            }
            else
            {
                if next is a ResponsibilityRef or a Stub then
                    remove(hyperedge)  // Remove current empty point and adjust links
                else
                    // Replace the empty point with a dummy responsibility
                    substituteWithDummy(hyperedge)
            } // if
        } // if
    } // forall
}
```

4.2 Tool Support

4.2.1 Overview

In UCMNav, one of the existing functionalities (the Save) exports the UCM object model to an XML file. All the classes and relationships are listed in Miga's thesis [21]. Our

transformation is from a UCM model to its CSM representation in XML format. Since there are many similarities between the two transformations, we actually reuse the existing save mechanism to implement the transformation from a UCM to CSM representation. The tool then takes a UCM object model as an input and saves that information in the CSM representation as an XML output file.

We have added one menu item called “SaveCSM” to the top-level main menu called “File” in the user interface. This is the starting point of our implementation, and the corresponding event handling functions are added to the related classes.

When users click the “SaveCSM” menu item, the first function invoked is the SaveCSMXML() from the XMLManager class (see Figure 23). In this function, the top level element <CSM> tag is written to the output file, and then each element in the device collection will be traversed.

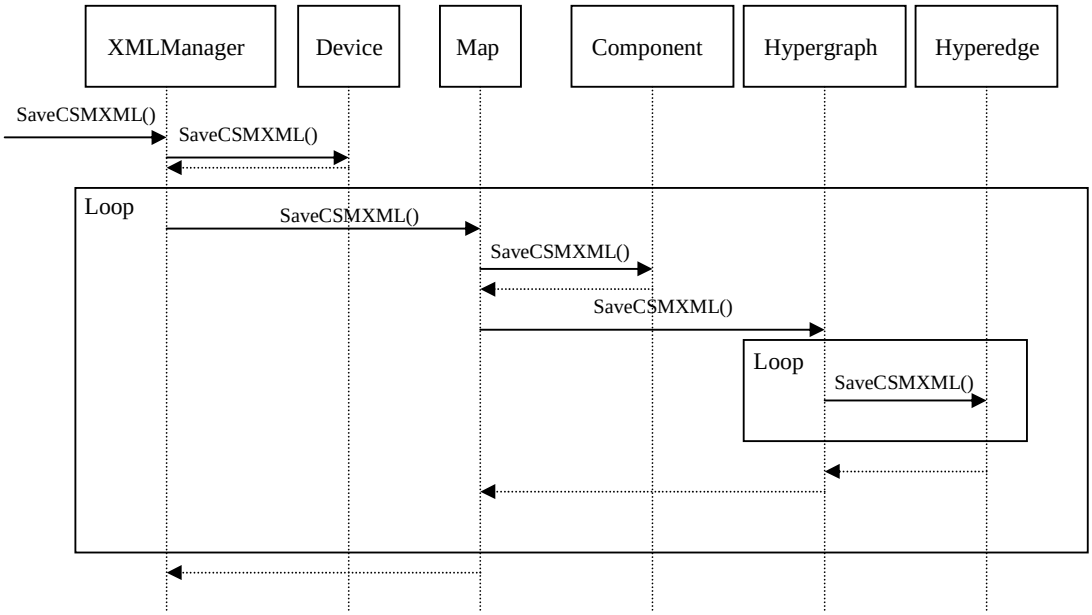


Figure 23 Sequence Diagram for algorithm implementation

After having output the device information, the tool traverses the map collection. Each map element can export its information under the block <Scenario> by invoking the SaveCSMXML() function in the Map class. Furthermore, each map object also has an hyperedge pool collection, which includes all the hyperedges that make up each map.

The most important event handler is located in the Hyperedge class. Each hyperedge element in the map is traversed. The same SaveCSMXML() function is invoked for the different hyperedge elements. In fact, the SaveCSMXML() in the Hyperedge class is

a pure virtual function. All the subclasses of Hyperedge have to provide their own implementation. By doing this, we can group related classes and their functions together.

A sample CSM output is provided in Annex B.

4.2.2 New Classes and Functions

Two new classes, ResourceAcquire (RA) and ResourceRelease (RR) were added as subclasses of Hyperedge. The purpose is to instantiate RA/RR objects along the path whenever necessary. RA/RR objects are mapped to independent steps in the CSM representation. As we discussed before, the insertion of RA/RR changes the flow relationship along a path, and it is very difficult to maintain such a relationship if we have no mechanism to record where the new objects are inserted. There are no additional methods and attributes in these two classes, except those inherited from the Hyperedge superclass.

We also added two functions, AddResourceAcquire() and AddResourceRelease(), to integrate resource acquiring and releasing actions with the transformation. These two functions are included in the Hyperedge class as virtual functions. All the subclasses of Hyperedge class override these two functions according to RA/RR algorithms.

Two new functions, SaveConnection( ) and SaveMultiConnection(), are added to deal with two relationships related to the current hyperedge element, i.e., the source & target, and the predecessor & successor relationships. These relationships are mapped to attributes of the current step/path connection elements.

When a UCM design model includes sub maps, the tool will generate both refinement and sub maps information in the output file. All the refinement information comes from the Stub object in the UCM model. Starting from the stub object, we can find a collection of plug-in maps that stub contains, and a collection of the entry bindings and the exit bindings.

4.3 Elements Not Mapped

The reader should notice that CSM is not intended to duplicate all the information found in UCMs. A consequence is that there are some concepts that are unique to UCM, and hence are not defined in CSM. These elements, therefore, are not transformed into CSM. If these elements are included in the input model to the transformation tool, the tool will not transform these elements. When the transformation algorithm meets these elements, it will just ignore them, do nothing and go to next element.

We list these concepts as the following, and remind our tool users that including those elements in the input model may not generate the expected output.

Connections: connection element links an end point to a start point or a waiting place, and captures the interaction between the two different scenario paths. The connection actually could be replaced by a sequence path (Figure 24**Error! Reference source not found.**).

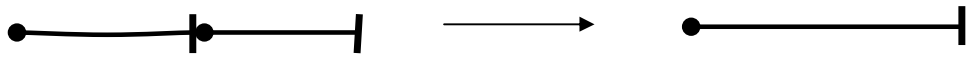


Figure 24 Connection replaced by a sequence path

Waiting place and timer: the path flow stops at a waiting place element until being restarted by a triggering event. A timer is a special waiting place where the triggering event arrives in a timely fashion. Two cases could be covered by our normalize algorithm. When an end point (or empty point) is connected to a waiting place then this combination can be mapped to a join element in CSM (Figure 25); when an end point (or empty point) is connected to a timer, it could be mapped to a join element in CSM.

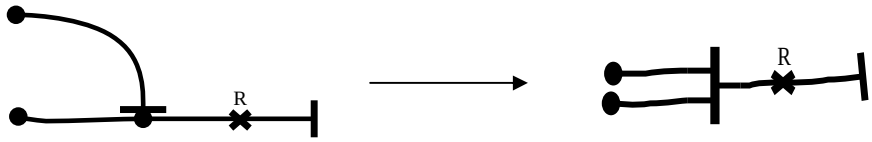


Figure 25 Waiting place normalization

Aborts: this concept is similar as the flow stop point in the UML2 activity diagram. Aborts element abort or disable the flow along another path. There is no equivalent concept in CSM.

Loops: loop element has two source and two target hyperedges. Two sources represent the original path and the end of the looping path; two targets represent the start of the looping path and the continuation of the looping path. Loop element also has an exit condition to specify the condition under which the loop will exit. The explicit loop concept could be added to CSM, since loop is a common concept for UCM, the UML2 activity diagram and interaction diagram. In the current version of CSM, a loop is represented as a step element with a repetition count attribute. The body of the loop is then captured as a sub-scenario. It could also be supported by the current branch and merge path connections found in CSM. However, our current transformation does not support this construct. How to map a loop in UCM to a step in CSM is left for future work.

Conditions and probabilities: UCM or-forks can have conditions and probabilities attached to their branches. This information can be converted to preconditions and probabilities associated with the first CSM step of each of these branches.

4.4 Generality of the Transformation Algorithm

Our RA, RR, and Normalize algorithms can be applied to other transformations such as from UML activity diagrams and sequence diagrams to CSM. In UML activity diagrams, activity partitions may be used to allocate resources among the nodes of an activity, and when a flow edge crosses the lines of swimlanes, then this implies resource acquire and release actions from the performance point of view. The Normalize algorithm contains the general mechanisms that can be applied to other object models, so other transformations to CSM can remove redundant elements and add necessary (dummy) steps.

4.5 Chapter Summary

This chapter describes the transformation process of UCM models to the CSM representation. We provided a mapping from UCM concepts to CSM. Some of the correspondences are direct mappings, and some are not. We develop the specific transformation algorithm to deal with each case. In particular, implicit mappings where developed and described algorithmically for resource acquisition and resource release. The Normalize algorithm is used to remove unnecessary empty points and add dummy steps to make the internal object model compliant with respect to the CSM schema. It can be extended further to cover some of the elements not mapped. After implementing the transformation tool, we are ready to validate it using various test cases, which is the topic of the next chapter.

5. Validation and Experimentation

5.1 Overview

After the implementation of the transformation algorithm, we need to validate if the tool works exactly as we want. The key to effective functional testing is systematic functional coverage and validation. Since the transformation algorithm deals with the mappings between UCM concepts and CSM concepts, it is not necessary to vary input values or supply invalid input values during the testing process. We simply want to gain confidence that all the mappings covered by the algorithm have been implemented correctly.

We also integrate the transformation algorithm into the UCM Navigator framework by adding a new feature to the user interface. Since the user interface for transformation is very simple in this case, we chose not to validate the user interface explicitly here.

The input in our black-box testing approach is a UCM model described as a XML file. The output of the transformation is another XML file which shows the same content using CSM concepts. This chapter presents the test cases used to validate the functionalities of our transformation algorithm. We use a simplified wireless system example from Amyot's paper to cover basic functionalities and concepts [2]. Apart from validating the basic functional requirements, we also need to verify the resource acquire and release algorithms and take into account all the possible combinations for resource acquire and resource release, since they are more complex than the basic transformation.

5.2 Test Cases

Transforming from one model to another demands a way of tracing related elements across the models. This kind of traceability can be validated through testing the transformation algorithm. In this section, we list a set of test cases that aim to test the mappings between UCM models and CSM models. Resulting CSM files in XML can be validated against the CSM schema from Annex A. They are also inspected manually to ensure that no element is missing.

5.2.1 Test Case 1: Basic Coverage

5.2.1.1 Description

In order to validate the mappings from UCM notation to CSM elements, we take an example from the wireless telephony domain. It makes use of most UCM notation elements in a context simpler than a real wireless system. The testing input is a UCM model whose root map is shown in Figure 26. This model includes start points, end points, responsibilities, or-forks, or-joins, and-forks, and-joins, static stubs, and dynamic stubs. In this map, there is no component (components and resource allocation/release will be covered in the remaining test cases). The testing output is an XML document which shows the same content using CSM concepts.

5.2.1.2 Test Procedure

1. Open the UCM model in the UCM Navigator.
2. From the file menu, choose “Save in CSM”.
3. In the pop up window, click either current map or the whole maps.
4. Specify the name of the output file and click ok.

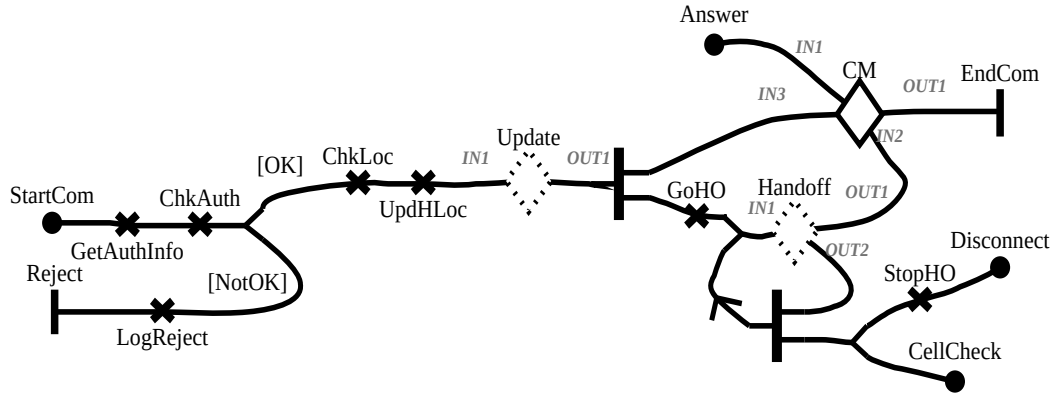


Figure 26 Basic coverage test case

The test result we expect is an XML file in which all the relevant UCM tags and attributes have been transformed to CSM tags and attributes. Once we have the output file, we can verify against the original UCM model the correctness and completeness of our transformation algorithm.

Actual test result: pass.

5.2.2 Test Case 2: No Component

5.2.2.1 Description

When there is no component in the input model, we do not expect any resource acquire and release actions to be included.

5.2.2.2 Test Procedure

Same as test case 1. The testing input is shown in Figure 27.

Figure 27 Path without component test case

The expected test result is simple XML files without any resource acquire or release element.
Actual test result: pass.

5.2.3 Test Case 3: One Component (1)

5.2.3.1 Description

When there is a component along the path with a responsibility inside the component, the incoming path implies a resource acquire action and the outgoing path implies a resource release action.

5.2.3.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 28.

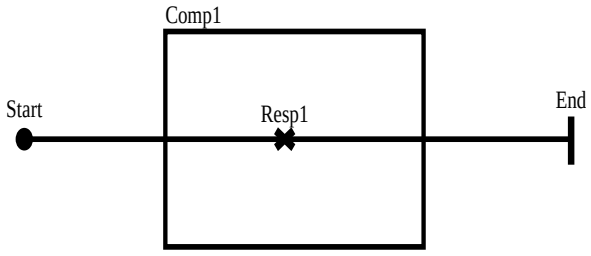


Figure 28 Path with one component test case

The expected testing result is one resource acquire element (with all the attributes) is shown before the Responsibility Resp1, and one resource release element (with all the attributes) after Resp1 element.
The actual test result: pass.

5.2.4 Test Case 4: One Component (2)

5.2.4.1 Description

When there is one component along the path with two responsibilities inside the component, the incoming path implies one resource acquire action and the outgoing path implies one resource release action. In fact, resource is acquired by the first step, and then is released by the second step. There is no any resource action between two steps since there is no component difference. They all reside in the same component.

5.2.4.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 29.

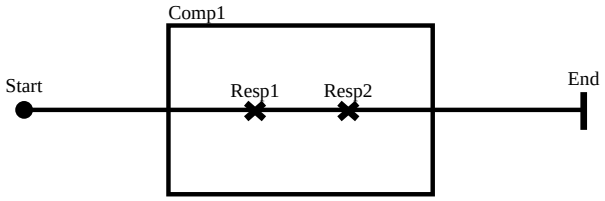


Figure 29 One component with two steps

The expected testing result is one resource acquire action (with all the attributes) is shown before the Responsibility Resp1, and one resource release action (with all the attributes) after Resp2.
The actual test result: pass.

5.2.5 Test Case 5: One Component (3)

5.2.5.1 Description

When there is one component along the path with all hyperedges (even empty points) outside the component, the incoming and outgoing paths imply no resource actions.

5.2.5.2 Test Procedure

Same as the test case 1. The testing input with an empty point inside component Comp1 is shown in Figure 30.

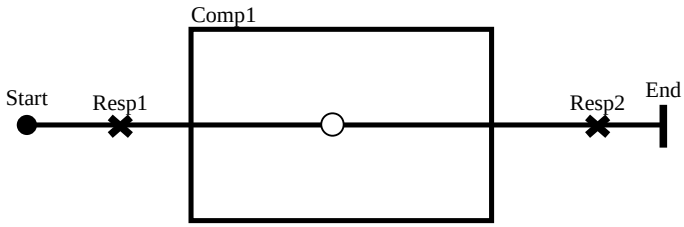


Figure 30 One component with two steps outside

The expected testing result is that resource acquiring and releasing actions happen between two Responsibilities: Resp1 and Resp2. (Since there is an empty point inside component)

The actual test result: pass.

5.2.6 Test Case 6: Two Components (1)

5.2.6.1 Description

When there are two components along the path with two responsibilities inside each component, the incoming and outgoing paths imply resource actions for each step. In fact, component Comp1 is acquired before the Responsibility Resp1 and then is released after Resp1. After that, component Comp2 is acquired by the Responsibility Resp2 and then is released after Resp2. The location of the empty point(s) located between Resp1 and Resp2 does not matter in this test.

5.2.6.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 31.

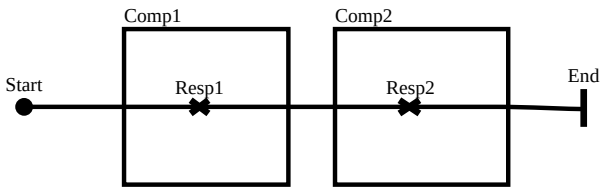


Figure 31 Two components with two steps

The expected testing result is one resource acquire element is shown before Resp1, and one resource release element is shown after Resp1; another resource acquire element is shown before Resp2, and another resource release element is shown after Resp2.
The actual test result: pass.

5.2.7 Test Case 7: Two Components (2)

5.2.7.1 Description

When there are two components along the path with one responsibility inside both components, the incoming and outgoing paths imply a sequence of resource actions for the Responsibility (the empty points are located outside the component Comp0). In fact, component Comp0 and Comp1 are acquired by the step Resp1 in sequence, and then Comp1 and Comp0 are released in sequence after Resp1.

5.2.7.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 32.

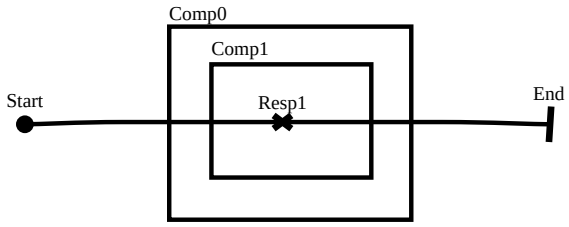


Figure 32 Two components with one step inside

The expected testing result is two resource acquire elements are shown before Resp1, and two resource release elements are shown after Resp1. The first resource acquire element has component Comp0 as its acquired component, and the second resource acquire element has Comp1. The first resource release element has Comp1 as its released component, and the second resource release element has Comp0.
The actual test result: pass.

5.2.8 Test Case 8: Two Components (3)

5.2.8.1 Description

When there are two components along the path with one responsibility inside one component (parent), and one empty point inside another component (child), the incoming and

outgoing paths imply a sequence of resource actions for the responsibility and empty point. Since component Comp0 is the parent of Comp1 and has a responsibility inside, it will be acquired by Resp1 at first. Component comp1 has one empty point inside, and it can be acquired by the empty point after Resp1. Both components are released in the reverse order.

5.2.8.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 33.

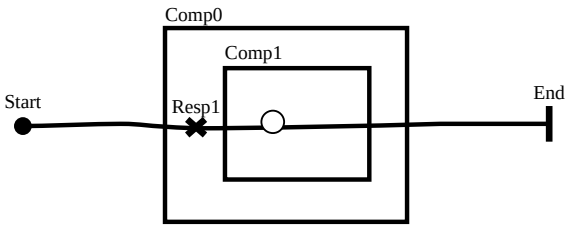


Figure 33 Two components with one step in the parent

The expected testing result is one resource acquire element is shown before Resp1, another resource acquired follows (with a sequence element in between) and two resource release elements (in reverse order) are shown before the End point.
The actual test result: pass.

5.2.9 Test Case 9: Two Components (4)

5.2.9.1 Description

When there are two components along the path with one responsibility inside one component (parent), and one empty point inside another component (child), the incoming and outgoing paths imply a sequence of resource actions for both empty point and Responsibility. The empty point has both component Comp0 and Comp1 as its acquired components, and then Comp1 is released before Responsibility Resp1. In fact, the Responsibility Resp1 cannot acquire component Comp0 (because there is no component difference between empty point and Resp1).

5.2.9.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 34.

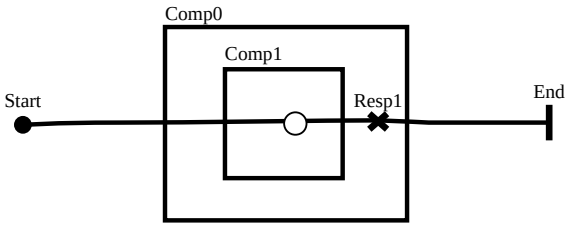


Figure 34 Two components with one step

The expected testing result is two resource acquire element are shown before the empty point and after Start point, and one resource release element (Comp1) is shown before Resp1, and another one (Comp0) is shown after Resp1.

The actual test result: pass.

5.2.10 Test Case 10: Two Components (5)

5.2.10.1 Description

When there are two components along the path with two responsibilities inside each of the components and the two components have the parent-child relationship, the incoming and outgoing paths imply a sequence of resource actions for the steps. Since component Comp0 has responsibility Resp0 inside, the Comp0 will be acquired by Resp0. The same thing happens for Resp1. It acquires component Comp1. But component Comp0 should not be re-acquired between Resp0 and Resp1. Component Comp1 will be released first, and then Comp0 is released.

5.2.10.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 35.

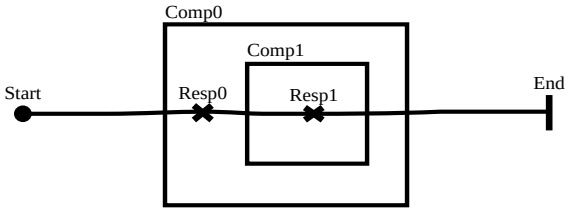


Figure 35 Two components with two steps

The expected testing result is one resource acquire element with Comp0 is shown before Resp0, one resource acquire element with Comp1 is shown after Resp0 and before Resp1;

one resource release element is shown after Resp1 with Comp1 as its released component, and another resource release element follows with Comp0 as its released component.
The actual test result: pass.

5.2.11 Test Case 11: Two Components (6)

5.2.11.1 Description

When there are two components along the path with two responsibilities inside each of the components and the two components have the parent-child relationship, the incoming and outgoing paths imply a sequence of resource actions for the steps. Since component Comp0 has responsibility Resp0 inside, the comp0 will be acquired before Resp0. The same responsibility Resp0 also resides in component Comp1, so component Comp1 is acquired as well. But Comp0 should not be re-acquired between Resp0 and Resp1. Component Comp1 will be released first (after Resp0), and then Comp0 is released (after Resp1).

5.2.11.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 36.

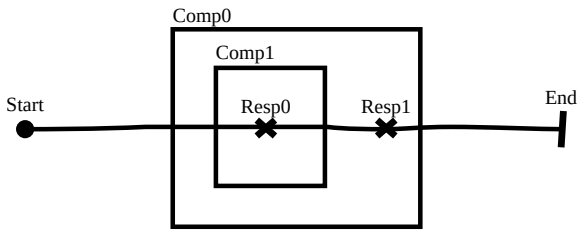


Figure 36 Two components with two steps (2)

The expected testing result is one resource acquire element with Comp0 and one resource acquire element with Comp1 are shown before Resp0; one resource release element, with Comp1 as its released component, is shown after Resp0 and before Resp1. And another resource release element, with Comp0 as its released component, follows the step Resp1.
The actual test result: pass.

5.2.12 Test Case 12: Two Components (7)

5.2.12.1 Description

When there are two components along the path with three responsibilities inside and the two components have the parent-child relationship, the incoming and outgoing paths imply a sequence of resource actions for the steps. Since component Comp0 has responsibility Resp0 inside, the comp0 will be acquired by Resp0. Since Resp1 resides in component Comp1, it acquires component Comp1 as well. But Comp0 should not be re-acquired between Resp0 and Resp1. The same thing happens at Resp2. It can only release Component Comp1 first, and then Comp0 is released after Resp2.

5.2.12.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 37.

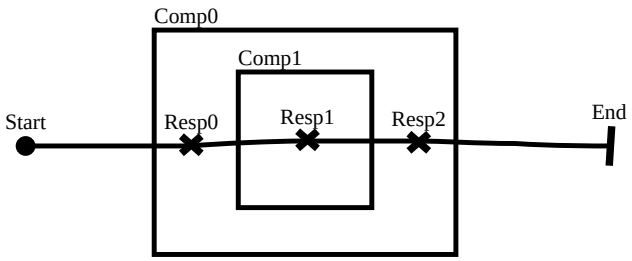


Figure 37 Two components with three steps

The expected testing result is one resource acquire element with Comp0 is shown before Resp0, and one resource acquire element with Comp1 is shown before Resp1 and after Resp0; one resource release element, with Comp1 as its released component, is shown after Resp1 and before Resp2. And another resource release element, with Comp0 as its released component, follows the step Resp2.

The actual test result: pass.

5.2.13 Test Case 13: Three Components (1)

5.2.13.1 Description

When there are three components along the path with three responsibilities inside each of the components, the incoming and outgoing paths imply a sequence of resource actions. Each component will be acquired and released in sequence.

5.2.13.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 38.

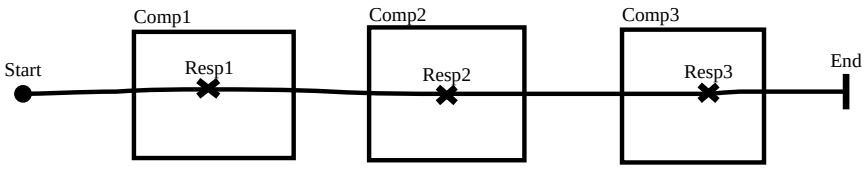


Figure 38 Three Components

The expected testing result is one resource acquire element with Comp1 is shown before the Reps1 and one resource release element with Comp1 is shown after Resp1. The same sequences happen at the Comp2 and Comp3.

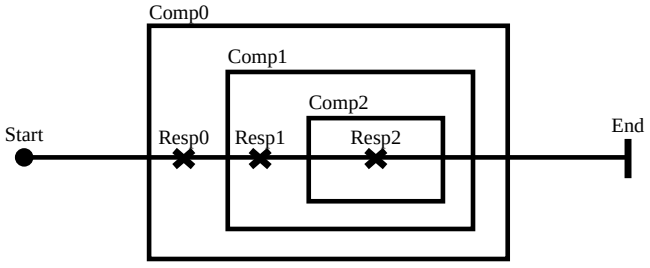
The actual test result: pass.

5.2.14 Test Case 14: Three Components (2)

5.2.14.1 Description

When there are three components along the path with three responsibilities inside each of the components and the three components have the parent-child relationships, the incoming and outgoing paths imply a sequence of resource actions for the steps. Since component Comp0 has responsibility Resp0 inside, it will be acquired before Resp0.The same thing happens for both the Comp1 and the Comp2. Three components are released in the reverse sequence.

5.2.14.2 Test Procedure



Same as the test case 1. The testing input is shown in Figure 39.

Figure 39 Three components with three steps

The expected testing result is one resource acquire element with Comp0 is shown before Resp0; one resource acquire element with Comp1 is shown before Resp1 and after Resp0; one resource acquire element with Comp2 is shown before Resp2 and after Resp1. One resource release element, with Comp2 as its released component, is shown after Resp2. Another resource release element, with Comp1 as its released component, follows. Finally, the last resource release element is for Comp0.

The actual test result: pass.

5.2.15 Test Case 15: Three Components (3)

5.2.15.1 Description

When there are three components along the path with two responsibilities inside two of the components and the three components have the parent-child relationships, the incoming and outgoing paths imply a sequence of resource actions for the steps. Since component Comp0 is the parent of component Comp1 and Comp2, it will be acquired at first by responsibility Resp1. Resp1 acquires Comp1 as well. Comp0 will be kept until all its children finish their operations.

5.2.15.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 40, and the output is presented in Annex B.

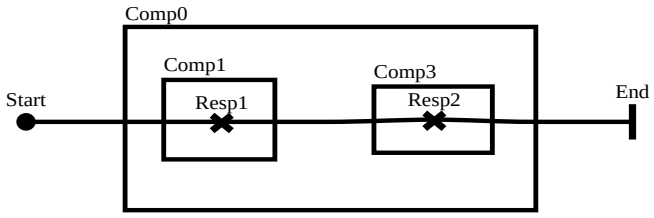


Figure 40 Three components with two responsibilities

The expected testing result is two resource acquire elements with Comp0 and Comp1 are shown between Start and Resp1; one resource releasing element with Comp1 is shown after Resp1; one resource acquire element with Comp3 is shown before Resp2 and after Resp1. One resource release element, with Comp3 as its released component, is shown after Resp2. Finally, another resource release element, with Comp0 as its released component, follows.

The actual test result: pass.

5.2.16 Test Case 16: Three Components (4)

5.2.16.1 Description

When a hyperedge needs to acquire more than two embedded components (e.g., Resp0 in), it needs to do so from the outermost component to the innermost one, while the release is in the reverse order. This checks that our *FindParentsRA* and *FindParentsRR* algorithms work properly.

5.2.16.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 41.

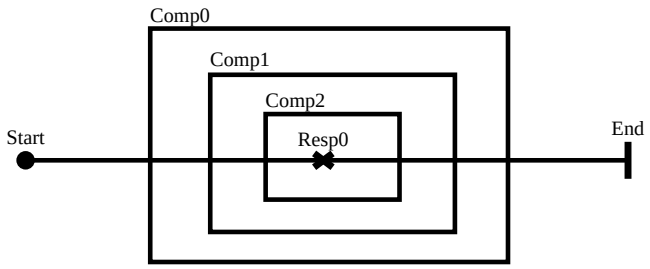


Figure 41 Three components all at once

The expected testing result is three resource acquire elements in sequence (with Comp0, Comp1, and Comp2) before Resp0, and three resource release elements in sequence (with Comp2, Comp1, and Comp0) after Resp0.
The actual test result: pass.

5.2.17 Test Case 17: Multiple Paths across One Component

5.2.17.1 Description

When there is one path crossing the same component twice, there will be multiple RA/RR actions happening in sequence. The same component can be acquired, released, acquired again, and finally released.

5.2.17.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 42.

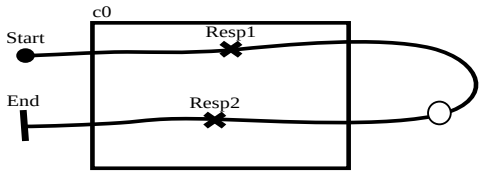


Figure 42 One path across one component twice

The expected testing result is that a resource acquire element (let us name it ra1) is shown between Start point and responsibility Resp1; a resource release element (rr1) is before Resp2 and after Resp1; another resource acquire element (ra2) is after rr1 and before Resp2. Finally, the last resource release element (rr2) is before End point and after Resp2. The actual test result: pass.

5.2.18 Test Case 18: WIN Example

5.2.18.1 Description

When there are multiple paths crossing the same component, there will be multiple RA/RR actions happening in sequence. The same component can be acquired, released, acquired again, and finally released. When start points and end points are inside components, those components can be acquired by start points and be released by end points (according to our assumptions).

5.2.18.2 Test Procedure

Same as the test case 1. The testing input is shown in Figure 43.

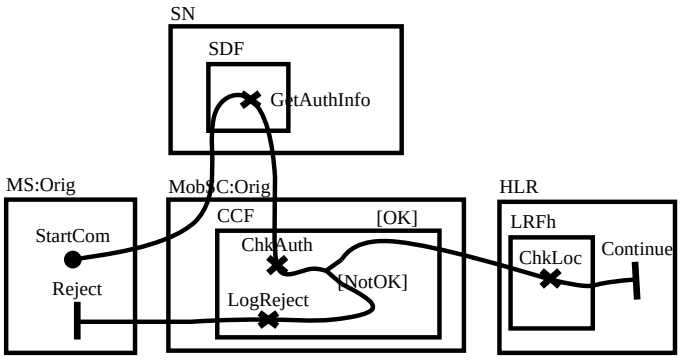


Figure 43 Win Example [2]

The expected testing result is that along the path, start point StartCom acquired the component MS:orig and then releases it; the component MobSC:Orig is acquired and released; components SN and SDF are acquired and released; components MobSC:Orig and CCF

are acquired and then are released twice; Finally, components HLR and LRFH are acquired and released by the end point Continue, and component MS:orig is acquired and released by the end point Reject.

The actual test result: pass.

5.3 Test Matrix

In this section we present a test matrix to help determine whether the transformation program has been adequately tested (Table 19).

Table 19 Test Matrix

	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8	TC9	TC10	TC11	TC12	TC13	TC14	TC15	TC16	TC17	TC18
CSM	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Scenario	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Step	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Start	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
ClosedWorkload			X	X	X													X
OpenWorkload						X	X	X										
End	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Sequence	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Branch	X																	X
Merge	X																	
Fork	X																	
Join	X																	
Component			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
ProcessRes			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
RA			X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
RR			X	X		X	X	X	X	X	X	X	X	X	X	X	X	X
Refinement	X																	
InBinding	X																	
OutBinding	X																	
Scenario/subs	X																	
source/target	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
succ/predec	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
sub_in/sub_out	X																	
parent/sub							X	X	X	X	X	X	X	X	X			X
multiple paths																X	X	X

Since the focus of the transformation program is to automatically transform from UCMs to CSMs, we simply ignore the unexpected and irrelevant elements. The exception handling part is not covered by our test cases. The results from transformation process were validated against the XML Schema and inspected manually.

Test case 1 covers all the classes, associations, and required attributes in the direct mappings from Test case 1 has no component involved. The rest of our test cases are used to test the different combinations of resource acquire and release. Our test objective

is to increase our confidence in the proper functioning of our transformation algorithms. From this point of view, we achieved our goal by using these test cases.

More experiments need to be done in our future work, especially on the end-to-end transformation process, i.e., from UCM models to the CSM, and then from CSM to performance models (most likely LQN). The reason why we do not do this kind of experiment at this time is that CSM-based transformation tools are not yet available.

5.4 Interoperability Testing

The output from our transformation tool has been validated by another transformation tool. As an interoperability testing, Maqbool in his thesis [20] uses our CSM output as an input to his own transformation tool and he successfully generates Petri nets. This demonstrates that the transformation process from UCM to CSM and then to Petri net is feasible, and that our transformation result is compatible with CSM-based tools.

5.5 Chapter Summary

This chapter discussed the various test cases used to validate our UCM-to-CSM transformation tool. We also present a test matrix showing that the important concepts involved in the transformation were covered by the test suite. Interoperability was also exercised by using the output of our transformation tool as input to another tool that generates Petri Nets from CSM models.

6. Conclusions and Future Work

6.1 Conclusions

We have reviewed several approaches in the early software performance analysis. These approaches were chosen due to their success in the automatic generation of performance models from scenario-based design specifications. As with most approaches that define innovative automation methodologies, there are arguments about what concepts are covered and what concepts should be added. Each early performance analysis approach has its own limitation, and there is a growing interest in finding a unified approach that would integrate them.

This thesis has focused on a unified process for early performance analysis based on the Core Scenario Model (CSM), as part of the PUMA research project [27]. The CSM meta-model was defined as a unified interface that hides differences and discrepancies between scenario-based modeling languages used at the design level and that simplifies transformations to performance models. This simplifies the creation of transformation tools as well as their maintenance as scenario languages and performance languages evolve.

We have implemented one application of CSM, i.e., the automatic transformation from Use Case Maps to the CSM representation. The direct and indirect mappings between UCM and CSM are handled automatically by our transformation algorithms. A transformation tool was developed and integrated as part of the UCM Navigator framework. We have validated our transformation tool by transforming an example from the wireless communication domain. More detailed work has been carried out to validate resource acquire and release algorithms.

6.2 Future Work

Several problems remain to be studied and solved. The most important one is how to integrate the performance analysis feedback back in UCM design specifications, with tool support. Although performance results can be stored back in a CSM model with traceability links to the original UCM model, several concepts in CSM, such as resource acquire and release elements, have no explicit mappings in the UCM model, and that may cause trouble when integrating design feedback and design decision with the design

specifications. How best to show the results and emphasize the hot spots in the original model is also a challenge.

Another improvement concerns the implementation of resource acquire and release algorithms. Although our transformation produces reasonable results at the moment, several optimizations when multiple paths cross the borders of a component (e.g., with forks and joins) could be envisioned. When and who will acquire and release this component, without acquiring or releasing it multiple times, needs further exploration.

Finally, other transformations need to be defined to support and validate various kinds of analysis, e.g., UML to CSM, CSM to QN, CSM to LQN, CSM to Stochastic Petri Nets, etc. Case studies and test suites need to be developed to further verify the suitability of CSM in different contexts, and to measure the general benefits promised by this unified approach.

References

[1] D. Amyot and G. Mussbacher, "On the Extension of UML with Use Case Maps Concepts". In: *<<UML 2000>>, 3rd International Conference on Unified Modeling Language*, York, UK, October 2000. LNCS 1939, 13-61.

[2] D. Amyot, "Introduction to the User Requirements Notation: Learning by Example". *Computer Networks*, 42(3), 285-301, 21 June 2003.

[3] D. Amyot, D.Y. Cho, X. He, and Y. He, "Generating Scenarios from Use Case Map Specifications". In: *Third International Conference on Quality Software (QSIC'03)*, Dallas, November 2003, pp. 108-115.

[4] D. Amyot, A. Echihabi, and Y. He, "UCMExporter: Supporting Scenario Transformations from Use Case Maps". In: *Nouvelles TEchnologies de la RÉpartition (NOTERE'04)*, Saidia, Morocco, June 2004.

[5] Apache Software Foundation. The Apache XML Project. See http://xml.apache.org/.

[6] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, "Model-based performance prediction in software development: a survey". *IEEE Transactions on Software Engineering*, Vol 30, No.5, May 2004, 295-310.

[7] R.J.A Buhr, and R.S. Casselman, Use *Case Maps for Object-Oriented Systems*, Prentice-Hall, USA, 1995.

[8] D. Carlson, Modeling XML Applications with UML. Addison-Wesley, 2001. See also http://www.xmlmodeling.com/hyperModel/.

[9] V. Cortellesa, "Deriving a Queueing Network Based Performance Model from UML Diagrams" *in Proc. Second Int. Workshop on Software and Performance (WOSP2000)*, Ottawa, Canada, 2000, pp. 58-70

[10] W. Fokkink, *Introduction to Process Algebra*, Springer, 2000.

[11] G. Franks, S. Majumdar, J. Neilson, D.C. Petriu, J. Rolia, and M. Woodside, "Performance Analysis of Distributed Server Systems", *Proc. 6th Int. Conf. on Software Quality (6ICSQ)*, Ottawa, Ontario, 1996, pp. 15-26.

[12] International Telecommunications Union, Recommendation Z.150 (02/03), User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.

[13] R. Jain, *the Art of Computer Systems Performance Analysis*, John Wiley & Sons, Inc., 1992.

[14] P. Kahkipuro, "UML-Based Performance Modeling Framework for Component-Based Systems", in *Performance Engineering*, R. Dumke, C. Rautenstrauch, A. Schmietendorf, and A. Scholz, Eds. Berlin: Springer, 2001.

[15] Layered Queueing Web Page, http://www.layeredqueues.org/

[16] J.P. López-Grao, J. Merseguer, and J. Campos, "From UML Activity Diagrams To Stochastic Petri Nets: Application To Software Performance Engineering" in *Fourth Int. Workshop on Software and Performance (WOSP 2004)*, Redwood City, CA, Jan. 2004, pp. 25-36.

References

[17] Object Management Group, *UML Profile for Schedulability, Performance, and Time Specification*, OMG Adopted Specification ptc/02-03-02, July 1, 2002

[18] Object Management Group, *Meta Object Facility (MOF) 2.0 Core Specification*, OMG Adopted Specification ptc/03-10-04, Oct. 2003.

[19] Object Management Group, *UML 2.0 Superstructure Specification*, OMG Adopted Specification, April 30, 2004.

[20] S. Maqbool, *Transformation of Core Scenario Model and Activity Diagrams to Petri Nets*, M.C.S. thesis proposal, SITE, University of Ottawa, 2005.

[21] A. Miga, *Application of Use Case Maps to System Design with Tool Support*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa. October 1998. http://www.UseCaseMaps.org/tools/ucmnav/

[22] D.B. Petriu and M. Woodside, "Software Performance Models from System Scenarios in Use Case Maps", in *Proc. 12 Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation (Performance TOOLS 2002)*, London, April 2002.

[23] D.B. Petriu, *Layered Software Performance Models Constructed from Use Case Map Specifications*. M.Eng. thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa. May 2001.

[24] D.B. Petriu, D. Amyot, and M. Woodside, "Scenario-Based Performance Engineering with UCMNav". *11th SDL Forum (SDL'03)*, Stuttgart, Germany, July 2003. LNCS 2708, 18-35.

[25] D.B. Petriu, D. Amyot, M. Woodside, and B. Jiang, "Traceability and Evaluation in Scenario Analysis by Use Case Maps". To appear in: S. Leue and T. Systä (Eds.) *Scenarios: Models, Algorithms and Tools*, LNCS 3466, Springer.

[26] D.B. Petriu, M. Woodside: "A Metamodel for Generating Performance Models from UML Designs". *<<UML 2004>>, 7th International Conference on Unified Modeling Language*, Lisbon, Portugal, 2004, pp. 41-53.

[27] D.C. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-based derivation of LQN models from UML specifications", in *Proc. 12th Int. Conf. on Modeling Tools and Techniques for Computer and Communication System Performance Evaluation*, London, England, 2002.

[28] D.C. Petriu and C.M. Woodside, "Performance Analysis with UML", in *UML for Real*, ed. B. Selic, L. Lavagno, and G. Martin, Kluwer, 2003, pp. 221-240.

[29] Puma Project. November 2002. See http://www.sce.carleton.ca/rads/puma/.

[30] C.U. Smith and L.G. Williams, *Performance Solutions.* Addison-Wesley, 2002.

[31] UCM User Group, *UCMNav XML Document Type Definition*, version 0.23, November 2001, http://www.usecasemaps.org/xml/index.shtml.

[32] X. Wu and M. Woodside, "Performance Modeling from Software Components", *in Proc. 4th Int. Workshop on Software and Performance (WOSP 2004)*, Redwood Shores, CA, USA, Jan. 2004, pp. 290-301.

[33] J. Xu, M. Woodside, and D.B. Petriu "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time", *Proc. 13th Int.*

Conf. on Computer Performance Evaluation, Modeling Techniques and Tools (TOOLS 2003), Urbana, Illinois, USA, Sept 2003, pp 291-310.

[34] Word Wide Web Consortium. XML Schema Part 0: Primer, W3C Recommendation, 2 May 2001—see http://www.w3.org/TR/xmlschema-0/

Annex A: CSM Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
  Core Scenario Model (CSM)
  schema version 0.26
  based on 0.25
  2005-May-08
-->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!--
  ================================================================================
    CSM  (root element)
  ================================================================================
    contained elements: CSMElement
    optional attributes: name
                         description
                         author
                         created
                         version
                         traceability_link
  ================================================================================
  -->
  <xsd:element name="CSM" type="CSMType"/>
  <xsd:complexType name="CSMType">
    <xsd:sequence>
      <xsd:element ref="CSMElement" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="author" type="xsd:string"/>
    <xsd:attribute name="created" type="xsd:dateTime"/>
    <xsd:attribute name="version" type="xsd:string"/>
    <xsd:attribute name="traceability_link" type="xsd:string"/>
  </xsd:complexType>
  <!--
  ================================================================================
    CSMElement
  ================================================================================
    subclasses: Scenario
                GeneralResource
                PerfMeasure
    required attributes: id
                         name
    optional attributes: description
                         traceability_link
 ================================================================================
  -->
  <xsd:element name="CSMElement" type="CSMElementType" abstract="true"/>
  <xsd:complexType name="CSMElementType">
    <xsd:attribute name="id" type="xsd:ID" use="required"/>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="traceability_link" type="xsd:string"/>
  </xsd:complexType>
```

```
<!--
 ================================================================================
   Scenario
 ================================================================================
   parent class: CSMElement
   contained elements: ScenarioElement
   optional attributes: probability
                        transaction
  optional associations: refinement (Refinement IDs)
 ================================================================================
 -->
 <xsd:element name="Scenario" type="ScenarioType"
              substitutionGroup="CSMElement"/>
 <xsd:complexType name="ScenarioType">
   <xsd:complexContent>
     <xsd:extension base="CSMElementType">
       <xsd:sequence>
         <xsd:element ref="ScenarioElement" minOccurs="0"
                                            maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="probability" type="xsd:double"/>
       <xsd:attribute name="transaction" type="xsd:boolean"/>
       <xsd:attribute name="refinement" type="xsd:IDREFS"/>
     </xsd:extension>
   </xsd:complexContent>
 </xsd:complexType>
 <!--
 ================================================================================
   ScenarioElement
 ================================================================================
   subclasses: Step
               PathConnection
               Classifier
   required attributes: id
   optional attributes: description
                        traceability_link
 ================================================================================
 -->
 <xsd:element name="ScenarioElement" type="ScenarioElementType"
              abstract="true"/>
 <xsd:complexType name="ScenarioElementType">
   <xsd:attribute name="id" type="xsd:ID" use="required"/>
   <xsd:attribute name="description" type="xsd:string"/>
   <xsd:attribute name="traceability_link" type="xsd:string"/>
 </xsd:complexType>
 <!--
 ================================================================================
   Step
 ================================================================================
   parent class: ScenarioElement
   subclasses: ResourceAcquire
               ResourceRelease
   contained elements: PreCondition
                       PostCondition
                       InputSet
                       OutputSet
                       ResourceAcquire
                       ResourceRelease
                       Refinement
                       ExternalDemand
   required attributes: name
   optional attributes: host_demand
                        probability
                        rep_count
   required associations: predecessor (PathConnection IDs)
                          successor (PathConnection IDs)
   optional associations: component (Component ID)
                          parent (Scenario ID)
```

```
                          perfMeasureTrigger (PerfMeasure IDs)
                          perfMeasureEnd (PerfMeasure IDs)
  ========================================================================
  -->
  <xsd:element name="Step" type="StepType" substitution-
Group="ScenarioElement"/>
  <xsd:complexType name="StepType">
    <xsd:complexContent>
      <xsd:extension base="ScenarioElementType">
        <xsd:sequence>
          <xsd:element ref="PreCondition" minOccurs="0"/>
          <xsd:element ref="PostCondition" minOccurs="0"/>
          <xsd:element ref="InputSet" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="OutputSet" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="ResourceAcquire" minOccurs="0"
                       maxOccurs="unbounded"/>
          <xsd:element ref="ResourceRelease" minOccurs="0"
                       maxOccurs="unbounded"/>
          <xsd:element ref="Refinement" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="ExternalDemand" minOccurs="0"
                       maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="host_demand" type="xsd:double"/>
        <xsd:attribute name="probability" type="xsd:double"/>
        <xsd:attribute name="rep_count" type="xsd:double"/>
        <xsd:attribute name="predecessor" type="xsd:IDREFS" use="required"/>
        <xsd:attribute name="successor" type="xsd:IDREFS" use="required"/>
        <xsd:attribute name="component" type="xsd:IDREF"/>
        <xsd:attribute name="perfMeasureTrigger" type="xsd:IDREFS"/>
        <xsd:attribute name="perfMeasureEnd" type="xsd:IDREFS"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ========================================================================
    ResourceAcquire
  ========================================================================
    parent class: Step
    optional attributes: r_units
                         priority
    required associations: acquire (GeneralResource ID)
  ========================================================================
  -->
  <xsd:element name="ResourceAcquire" type="ResourceAcquireType"
               substitutionGroup="Step"/>
  <xsd:complexType name="ResourceAcquireType">
    <xsd:complexContent>
      <xsd:extension base="StepType">
        <xsd:attribute name="r_units" type="xsd:double"/>
        <xsd:attribute name="priority" type="xsd:string"/>
        <xsd:attribute name="acquire" type="xsd:IDREF" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ========================================================================
    ResourceRelease
  ========================================================================
    parent class: Step
    optional attributes: r_units
    required associations: release (GeneralResource ID)
  ========================================================================
  -->
  <xsd:element name="ResourceRelease" type="ResourceReleaseType"
               substitutionGroup="Step"/>
  <xsd:complexType name="ResourceReleaseType">
    <xsd:complexContent>
```

```
      <xsd:extension base="StepType">
        <xsd:attribute name="r_units" type="xsd:double"/>
        <xsd:attribute name="release" type="xsd:IDREF" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
  PathConnection
 ================================================================================
    subclasses: Sequence
                Branch
                Merge
                Fork
                Join
                Start
                End
    required attributes: id
    optional attributes: traceability_link
    optional associations: source (Step IDs)
                           target (Step IDs)
                           classifier (Classifier IDs)
                           subIn (InBinding IDs)
                           subOut (OutBinding IDs)
  ================================================================================
  -->
  <xsd:element name="PathConnection" type="PathConnectionType" abstract="true"
               substitutionGroup="ScenarioElement"/>
  <xsd:complexType name="PathConnectionType">
    <xsd:complexContent>
      <xsd:extension base="ScenarioElementType">
        <xsd:attribute name="source" type="xsd:IDREFS"/>
        <xsd:attribute name="target" type="xsd:IDREFS"/>
        <xsd:attribute name="classifier" type="xsd:IDREFS"/>
        <xsd:attribute name="subIn" type="xsd:IDREFS"/>
        <xsd:attribute name="subOut" type="xsd:IDREFS"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    Sequence
  ================================================================================
    parent class: PathConnection
  ================================================================================
  -->
  <xsd:element name="Sequence" type="SequenceType"
               substitutionGroup="PathConnection"/>
  <xsd:complexType name="SequenceType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    Branch
  ================================================================================
    parent class: PathConnection
  ================================================================================
  -->
  <xsd:element name="Branch" type="BranchType"
               substitutionGroup="PathConnection"/>
  <xsd:complexType name="BranchType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
```

```
  ============================================================================
    Merge
  ============================================================================
    parent class: PathConnection
  ============================================================================
  -->
  <xsd:element name="Merge" type="MergeType"
               substitutionGroup="PathConnection"/>
 <xsd:complexType name="MergeType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ============================================================================
    Fork
  ============================================================================
    parent class: PathConnection
  ============================================================================
  -->
  <xsd:element name="Fork" type="ForkType" substitutionGroup="PathConnection"/>
  <xsd:complexType name="ForkType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ============================================================================
    Join
  ============================================================================
    parent class: PathConnection
  ============================================================================
  -->
  <xsd:element name="Join" type="JoinType" substitutionGroup="PathConnection"/>
  <xsd:complexType name="JoinType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ============================================================================
    Start
  ============================================================================
    parent class: PathConnection
    contained elements: Workload
    optional associations: inBinding (InBinding IDs)
  ============================================================================
  -->
  <xsd:element name="Start" type="StartType"
               substitutionGroup="PathConnection"/>
  <xsd:complexType name="StartType">
    <xsd:complexContent>
      <xsd:extension base="PathConnectionType">
        <xsd:sequence>
          <!-- contained elements -->
          <xsd:element ref="Workload" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="inBinding" type="xsd:IDREFS"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ============================================================================
    End
  ============================================================================
    parent class: PathConnection
    optional associations: outBinding (OutBinding IDs)
  ============================================================================
```

```
-->
<xsd:element name="End" type="EndType" substitutionGroup="PathConnection"/>
<xsd:complexType name="EndType">
  <xsd:complexContent>
    <xsd:extension base="PathConnectionType">
      <xsd:attribute name="outBinding" type="xsd:IDREFS"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  InputSet
=============================================================================
  contained elements: PreCondition
  required associations: predecessorSubset (PathConnection IDs)
  constraint: the predecessors are a subset of the parent step's predecessors
=============================================================================
-->
<xsd:element name="InputSet" type="InputSetType"/>
<xsd:complexType name="InputSetType">
  <xsd:sequence>
    <xsd:element ref="PreCondition" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="predecessorSubset" type="xsd:IDREFS" use="required"/>
</xsd:complexType>
<!--
=============================================================================
  OutputSet
=============================================================================
  contained elements: PostCondition
  required associations: successorSubset (PathConnection IDs)
  constraint: the successors are a subset of the parent step's successors
=============================================================================
-->
<xsd:element name="OutputSet" type="OutputSetType"/>
<xsd:complexType name="OutputSetType">
  <xsd:sequence>
    <xsd:element ref="PostCondition" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="successorSubset" type="xsd:IDREFS" use="required"/>
</xsd:complexType>
<!--
=============================================================================
  Constraint
=============================================================================
  subclasses: PreCondition
              PostCondition
  required attributes: expression
=============================================================================
-->
<xsd:element name="Constraint" type="ConstraintType" abstract="true"/>
<xsd:complexType name="ConstraintType">
  <xsd:attribute name="expression" type="xsd:string" use="required"/>
</xsd:complexType>
<!--
=============================================================================
  PreCondition
=============================================================================
  parent class: Constraint
=============================================================================
-->
<xsd:element name="PreCondition" type="PreConditionType"
             substitutionGroup="Constraint"/>
<xsd:complexType name="PreConditionType">
  <xsd:complexContent>
    <xsd:extension base="ConstraintType"/>
  </xsd:complexContent>
</xsd:complexType>
```

```
  <!--
  ================================================================================
    PostCondition
  ================================================================================
    parent class: Constraint
  ================================================================================
  -->
  <xsd:element name="PostCondition" type="PostConditionType"
               substitutionGroup="Constraint"/>
  <xsd:complexType name="PostConditionType">
    <xsd:complexContent>
      <xsd:extension base="ConstraintType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    Classifier
  ================================================================================
    parent class: ScenarioElement
    subclasses: Message
    optional attributes: name
  ================================================================================
  -->
  <xsd:element name="Classifier" type="ClassifierType"
                substitutionGroup="ScenarioElement"/>
  <xsd:complexType name="ClassifierType">
    <xsd:complexContent>
      <xsd:extension base="ScenarioElementType">
        <xsd:attribute name="name" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    Message
  ================================================================================
    parent class: Classifier
    required attributes: kind
    optional attributes: size
                         multi
  ================================================================================
  -->
  <xsd:element name="Message" type="MessageType" substitution-
Group="Classifier"/>
  <xsd:complexType name="MessageType">
    <xsd:complexContent>
      <xsd:extension base="ClassifierType">
        <xsd:attribute name="kind" type="MsgKind" default="async"/>
        <xsd:attribute name="size" type="xsd:double"/>
        <xsd:attribute name="multi" type="xsd:double"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    <<enumeration>> MsgKind
  ================================================================================
    values: async | sync | reply
  ================================================================================
  -->
  <xsd:simpleType name="MsgKind">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="async"/>
      <xsd:enumeration value="sync"/>
      <xsd:enumeration value="reply"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!--
```

```
==========================================================================
  Refinement
==========================================================================
  contained elements: InBinding
                      OutBinding
  optional attributes: selection_cond
  required associations: parent (Step ID)
                         sub (Scenario ID)
==========================================================================
-->
<xsd:element name="Refinement" type="RefinementType"/>
<xsd:complexType name="RefinementType">
  <xsd:sequence>
    <xsd:element ref="InBinding" maxOccurs="unbounded"/>
    <xsd:element ref="OutBinding" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="selection_cond" type="xsd:string"/>
  <xsd:attribute name="parent" type="xsd:IDREF" use="required"/>
  <xsd:attribute name="sub" type="xsd:IDREF" use="required"/>
</xsd:complexType>
<!--
==========================================================================
  InBinding
==========================================================================
  required attributes: id
  required associations: in (PathConnection ID)
                         start (Start ID)
==========================================================================
-->
<xsd:element name="InBinding" type="InBindingType"/>
<xsd:complexType name="InBindingType">
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="in" type="xsd:IDREF" use="required"/>
  <xsd:attribute name="start" type="xsd:IDREF" use="required"/>
</xsd:complexType>
<!--
==========================================================================
  OutBinding
==========================================================================
  required attributes: id
  required associations: end (End ID)
                         out (PathConnection ID)
==========================================================================
-->
<xsd:element name="OutBinding" type="OutBindingType"/>
<xsd:complexType name="OutBindingType">
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="end" type="xsd:IDREF" use="required"/>
  <xsd:attribute name="out" type="xsd:IDREF" use="required"/>
</xsd:complexType>
<!--
==========================================================================
  ExternalDemand
==========================================================================
  required attributes: demand
  required associations: extop (ExternalOperation ID)
==========================================================================
-->
<xsd:element name="ExternalDemand" type="ExternalDemandType"/>
<xsd:complexType name="ExternalDemandType">
  <xsd:attribute name="demand" type="xsd:double"/>
  <xsd:attribute name="extop" type="xsd:IDREF"/>
</xsd:complexType>
<!--
==========================================================================
  GeneralResource
==========================================================================
  parent class: CSMElement
```

```
  subclasses: PassiveResource
              ActiveResource
  optional attributes: multiplicity
                       sched_policy
  optional associations: perfMeasure (PerfMeasure IDs)
=============================================================================
-->
<xsd:element name="GeneralResource" type="GeneralResourceType"
             abstract="true" substitutionGroup="CSMElement"/>
<xsd:complexType name="GeneralResourceType">
  <xsd:complexContent>
    <xsd:extension base="CSMElementType">
      <xsd:attribute name="multiplicity" type="xsd:int"/>
      <xsd:attribute name="sched_policy" type="xsd:string"/>
      <xsd:attribute name="perfMeasure" type="xsd:IDREFS"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  PassiveResource
=============================================================================
  parent class: GeneralResource
  subclasses: Component
=============================================================================
-->
<xsd:element name="PassiveResource" type="PassiveResourceType"
             substitutionGroup="GeneralResource"/>
<xsd:complexType name="PassiveResourceType">
  <xsd:complexContent>
    <xsd:extension base="GeneralResourceType"/>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  Component
=============================================================================
  parent class: PassiveResource
  optional attributes: is_active
  optional associations: host (ProcessingResource ID)
                         parent (Component ID)
                         sub (Component IDs)
=============================================================================
-->
<xsd:element name="Component" type="ComponentType"
             substitutionGroup="PassiveResource"/>
<xsd:complexType name="ComponentType">
  <xsd:complexContent>
    <xsd:extension base="PassiveResourceType">
      <xsd:attribute name="is_active_process" type="xsd:boolean"/>
      <xsd:attribute name="host" type="xsd:IDREF" use="required"/>
      <xsd:attribute name="parent" type="xsd:IDREF"/>
      <xsd:attribute name="sub" type="xsd:IDREFS"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  ActiveResource
=============================================================================
  parent class: GeneralResource
  subclasses: ProcessingResource
              ExternalOperation
  optional attributes: operation_time
=============================================================================
-->
<xsd:element name="ActiveResource" type="ActiveResourceType"
```

```
              substitutionGroup="GeneralResource"/>
<xsd:complexType name="ActiveResourceType">
  <xsd:complexContent>
    <xsd:extension base="GeneralResourceType">
      <xsd:attribute name="op_time" type="xsd:double"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  ProcessingResource
=============================================================================
  parent class: ActiveResource
=============================================================================
-->
<xsd:element name="ProcessingResource" type="ProcessingResourceType"
             substitutionGroup="ActiveResource"/>
<xsd:complexType name="ProcessingResourceType">
  <xsd:complexContent>
    <xsd:extension base="ActiveResourceType"/>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  ExternalOperation
=============================================================================
  parent class: ActiveResource
=============================================================================
-->
<xsd:element name="ExternalOperation" type="ExternalOperationType"
             substitutionGroup="ActiveResource"/>
<xsd:complexType name="ExternalOperationType">
  <xsd:complexContent>
    <xsd:extension base="ActiveResourceType">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  Workload
=============================================================================
  subclasses: ClosedWorkload
              OpenWorkload
  optional attributes: arrival_pattern
                       arrival_param1
                       arrival_param2
                       external_delay
                       value
                       coeff_var_sq
                       description
                       traceability_link
  optional associations: response_time (PerfMeasure IDs)
=============================================================================
-->
<xsd:element name="Workload" type="WorkloadType" abstract="true"/>
<xsd:complexType name="WorkloadType">
  <xsd:attribute name="id" type="xsd:ID" use="required"/>
  <xsd:attribute name="arrival_pattern" type="ArrivalProcess"
                 default="poissonPDF"/>
  <xsd:attribute name="arrival_param1" type="xsd:double"/>
  <xsd:attribute name="arrival_param2" type="xsd:double"/>
  <xsd:attribute name="external_delay" type="xsd:double"/>
  <xsd:attribute name="value" type="xsd:double"/>
  <xsd:attribute name="coeff_var_sq" type="xsd:double"/>
  <xsd:attribute name="description" type="xsd:string"/>
  <xsd:attribute name="traceability_link" type="xsd:string"/>
  <xsd:attribute name="responseTime" type="xsd:IDREFS"/>
</xsd:complexType>
```

```
  <!--
  ================================================================================
    ClosedWorkload
  ================================================================================
    parent class: Workload
    required attributes: population
  ================================================================================
  -->
  <xsd:element name="ClosedWorkload" type="ClosedWorkloadType"
                substitutionGroup="Workload"/>
  <xsd:complexType name="ClosedWorkloadType">
    <xsd:complexContent>
      <xsd:extension base="WorkloadType">
        <xsd:attribute name="population" type="xsd:int" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    OpenWorkload
  ================================================================================
    parent class: Workload
  ================================================================================
  -->
  <xsd:element name="OpenWorkload" type="OpenWorkloadType"
              substitutionGroup="Workload"/>
  <xsd:complexType name="OpenWorkloadType">
    <xsd:complexContent>
      <xsd:extension base="WorkloadType"/>
    </xsd:complexContent>
  </xsd:complexType>
  <!--
  ================================================================================
    <<enumeration>> ArrivalProcess
  ================================================================================
    values: poissonPDF | periodic | uniform | phase_type
    (not included: bounded, bursty, bernoulli, binomial)
  ================================================================================
  -->
  <xsd:simpleType name="ArrivalProcess">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="poissonPDF"/>
      <xsd:enumeration value="periodic"/>
      <xsd:enumeration value="uniform"/>
      <xsd:enumeration value="phase_type"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!--
  ================================================================================
    PerfMeasure
  ================================================================================
    contained elements: PerfValue
    required attributes: measure_type
    optional associations: trigger (Step ID)
                           end (Step ID)
                           duration (Workload ID)
                           resource (GeneralResource ID)
    constraints: one of trigger, duration, or resource association is required
                 end association can only appear if trigger association is pre-
sent
  ================================================================================
  -->
  <xsd:element name="PerfMeasure" type="PerfMeasureType"
              substitutionGroup="CSMElement"/>
  <xsd:complexType name="PerfMeasureType">
    <xsd:complexContent>
      <xsd:extension base="CSMElementType">
        <xsd:sequence>
```

```
        <xsd:element ref="PerfValue" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="measure" type="PerfAttribute" default="delay"/>
      <xsd:attribute name="trigger" type="xsd:IDREF"/>
      <xsd:attribute name="end" type="xsd:IDREF"/>
      <xsd:attribute name="duration" type="xsd:IDREF"/>
      <xsd:attribute name="resource" type="xsd:IDREF"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--
=============================================================================
  PerfValue
=============================================================================
  required attributes: value
  optional attributes: source
                       kind
                       percentile
                       kth_moment
=============================================================================
-->
<xsd:element name="PerfValue" type="PerfValueType"/>
<xsd:complexType name="PerfValueType">
  <xsd:attribute name="value" type="xsd:string" use="required"/>
  <xsd:attribute name="kind" type="PerfValueKind" default="mean"/>
  <xsd:attribute name="source" type="PerfValueSource" default="required"/>
  <xsd:attribute name="percentile" type="xsd:string"/>
  <xsd:attribute name="kth_moment" type="xsd:string"/>
</xsd:complexType>
<!--
=============================================================================
  <<enumeration>> PerfValueKind
=============================================================================
  values: mean | variance | percentile | moment | min | max | distribution
=============================================================================
-->
<xsd:simpleType name="PerfValueKind">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="mean"/>
    <xsd:enumeration value="variance"/>
    <xsd:enumeration value="percentile"/>
    <xsd:enumeration value="moment"/>
    <xsd:enumeration value="min"/>
    <xsd:enumeration value="max"/>
    <xsd:enumeration value="distribution"/>
  </xsd:restriction>
</xsd:simpleType>
<!--
=============================================================================
  <<enumeration>> PerfValueSource
=============================================================================
  values: required | assumed | predicted | measured
=============================================================================
-->
<xsd:simpleType name="PerfValueSource">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="required"/>
    <xsd:enumeration value="assumed"/>
    <xsd:enumeration value="predicted"/>
    <xsd:enumeration value="measured"/>
  </xsd:restriction>
</xsd:simpleType>
<!--
=============================================================================
  <<enumeration>> PerfAttribute
=============================================================================
  values: delay | throughput | utilization | interval | wait
=============================================================================
```

```
  -->
  <xsd:simpleType name="PerfAttribute">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="delay"/>
      <xsd:enumeration value="throughput"/>
      <xsd:enumeration value="utilization"/>
      <xsd:enumeration value="interval"/>
      <xsd:enumeration value="wait"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Annex B: Sample Output XML for Test Case 15

The following sample was produced by the export tool for test case 15 (Figure 40).

```
<?xml version="1.0" encoding="utf-8"?>
<CSM name="root" description="Test Case 15" >
  <Component id="c0" name="Comp0" host="" sub="c1 c2 " />
  <Component id="c1" name="Comp1" host="" parent="c0" />
  <Component id="c2" name="Comp3" host="" parent="c0" />
  <Scenario id="m0" name="root" description="" >
   <Start id="h0" name="Start" target= "h9" >
       <OpenWorkload id="w0" />
   </Start>
   <End id="h2" name="End" source= "h13" />
   <Step id="h3" name="Resp1" description=" " component="c1" host_demand="0"
         predecessor= "h15" successor= "h16" />
   <Step id="h6" name="Resp2" description=" " component="c2" host_demand="0"
         predecessor= "h18" successor= "h17" />
   <Sequence id="h7" name=" " source= "h12" target= "h11" />
   <ResourceAcquire id="h9" acquire="c0" predecessor= "h0" successor= "h14"/>
   <ResourceAcquire id="h10" acquire="c1" predecessor= "h14" successor= "h15"/>
   <ResourceAcquire id="h11" acquire="c2" predecessor= "h7" successor= "h18"/>
   <ResourceRelease id="h12" release="c1" predecessor= "h16" successor= "h7"/>
   <ResourceRelease id="h13" release="c2" predecessor= "h17" successor= "h19"/>
   <ResourceRelease id="h20" release="c0" predecessor= "h19" successor= "h2"/>
   <Sequence id="h14" name=" " source="h9" target="h10" />
   <Sequence id="h15" name=" " source="h10" target="h3" />
   <Sequence id="h16" name=" " source="h3" target="h12" />
   <Sequence id="h17" name=" " source="h6" target="h13" />
   <Sequence id="h18" name=" " source="h11" target="h6" />
   <Sequence id="h19" name=" " source="h13" target="h20" />
   </Scenario>
</CSM>
```

Annex C: Output for Explicit Mapping Example

The following sample was produced by the export tool for the explicit mapping example (Figure 15).

```
<?xml version="1.0" encoding="utf-8"?>
<CSM name="root" description="" >
   <Scenario id="m0" name="root" description="" >
      <Start id="h0" name="s" target= "h12" >
         <OpenWorkload id="w0" />
      </Start>
      <End id="h2" name="e" source= "h13" />
      <Sequence id="h4" name=" " source= "h12" target= "h3" />
      <Sequence id="h5" name=" " source= "h3" target= "h13" />
      <Step id="h12" name="Dummy Step" source= "h0" target= "h4" />
      <Step id="h13" name="Dummy Step" source= "h5" target= "h2" />
      <Step id="h3" name="stub" source= "h4" target="h5"/>
      <Refinement parent="h3" sub="m1">
         <InBinding id="si2" start="h6" in="h4"/>
         <OutBinding id="so3" end="h8" out="h5"/>
      </Refinement>
   </Scenario>
   <Scenario id="m1" name="sub" description="" >
      <Start id="h6" name="start" target= "h9" >
         <OpenWorkload id="w6" />
      </Start>
      <End id="h8" name="end" source= "h9" />
      <Step id="h9" name="resp1" description=" " host_demand="0" prede-
cessor= "h6" successor= "h8" />
   </Scenario>
</CSM>
```