Layered Software Performance Models Constructed from Use Case Map Specifications

by

Dorin B. Petriu, B.Eng.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements of the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, K1S 5B6
Canada

May 9th, 2001

The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the thesis

Layered Software Performance Models Constructed from Use Case Map Specifications

submitted by Dorin B. Petriu, B.Eng. in partial fulfillment of
the requirements of the degree of Master of Engineering

Dr. John W. Chinneck, Chair
Department of Systems and Computer Engineering

Dr. C. Murray Woodside, Thesis Supervisor

Carleton University
May 9th, 2001

Abstract

This thesis addresses the problem of analyzing the performance of an early design specification of a concurrent software system. The design is specified using Use Case Maps (UCMs), a scenario notation which describes paths with responsibilities overlayed on components. The resulting performance model is created using Layered Queueing Networks (LQNs). UCM2LQN, an automated conversion tool that generates LQNs from UCMs, and the algorithm behind it are introduced as a solution for bridging the gap between design and performance analysis. UCM2LQN has been integrated into the UCM Navigator editing tool and generates LQNs suitable for use as input into two existing LQN solvers. The UCM2LQN converter is validated against a set of in-house UCM designs for a Plain Old Telephone System, a Ticket Reservation System, and a Group Communication Server. It is also tested against two industrial designs for a Call Delivery and Set-up for Wireless Intelligent Networks and a Distributed Hand-off Protocol.

Acknowledgments

I would like to thank my supervisor, Murray Woodside, for his understanding, guidance, support, and feedback. He has helped me emerge from the muddle that is graduate studies with hope and enthusiasm for the future. Thank you Murray!

I never thank my parents enough for their love and support. It has been a sometimes windy road but I've made it through and I couldn't have done it without you. Thank you for knowing when to push and when not to. I love you and sis very much and I hope you're as proud of me as I am of you.

A big thank you to all my friends (you know who you are). You've helped keep me sane when I needed it and I'm grateful for that. Life would not be life without stopping to smell the roses...

Thank you to Daniel Amyot, Gunther Mussbacher, and Khalid Siddiqui for helping me with material to test my program on. I would also like to extend my appreciation to NSERC and Carleton University for the scholarship support I received.

And mom, thanks for all the lunches!

Table of Contents

Chapter 1 - Introduction

This thesis presents the UCM2LQN converter, an automated tool that converts annotated Use Case Map (UCM) design models into Layered Queueing Network (LQN) performance models. The UCM2LQN converter works as a link between the existing UCM Navigator (UCMNav) UCM editing tool and two existing LQN analysis tools, LQNS and ParaSRVN.

The UCM2LQN program is an add-on to the UCMNav and uses the UCMNav internal data structure for UCMs in order to create an equivalent LQN model. The LQN model is saved to file in the format used by the LQNS and ParaSRVN tools. Users can thus use UCM2LQN and either of the LQN analysis tools to generate performance data for high-level designs.

1.1. Motivation

Software design and performance analysis are two vital, yet poorly coordinated aspects of the development of software systems. All too often in industry design takes the front stage and is viewed as the key to reducing time-to-market. Performance analysis is viewed as too cumbersome and time-consuming, and when it is done at all is usually as a validation step after the design has been finalized. Thus traditionally software designers design the system and only afterwards do performance analysts get to see it. In this approach performance evaluation is seen as being a part of integration testing at the end of the design cycle.

Since software designers are not performance analysts there is a lot of overhead in going from design to analysis in this traditional paradigm. In order to get a performance analysis done on a design the designers need to meet with the performance analysts, provide them with documentation and a thorough explanation of the system, wait as the performance analysts come up with their own performance model of the system, and wait to finally get the results back. By this time it is likely that key design decisions and commitments have already been made and the design has evolved to the point where it no longer reflects the same system as the one that was analysed. Therefore if the performance analysis uncovers any weaknesses in the original design, addressing them might require a fair bit of re-engineering, or even worse, the performance analysis results might be overlooked altogether since they are out of date.

This particular paradigm of separate fiefdoms for designers and performance analysts is sadly enough pretty much today's norm in industry. Time-to-market pressures and the ever-accelerating speed at which systems must be developed makes it unlikely that this will change as long as the integration of design and performance analysis remains hampered by the high overhead of going from one phase to the other.

Software Performance Engineering (SPE) was an early attempt at evolving the software development paradigm into something that includes performance analysis at an early stage in the design. Championed by Connie Smith, SPE came up in the late 1980's as a research idea with the first publications appearing in the early 1990's[43]. A considerable body of research and literature on software design and performance has grown since then, an overview of which can be obtained from the proceedings of the two international Workshops on Software and Performance (WOSP'98 [52] and WOSP2000 [53]).

The SPE software development model proposes developing performance models at the same time as the design. The results from performance analysis performed early on in the development cycle can thus be integrated back into the design at a point at which they can make an effective contribution. SPE has proven to be appealing in concept but rarely adopted in practice. This is due to the fact that SPE is a methodology for going between the design and performance analysis realms and as such it still requires knowledgeable and trained people to implement it. Acquiring this knowledge and training takes time and given the nature of the industry this time is usually not available.

A possible solution appears to be the automation of the transition from design document to performance model. As the increasing adoption of CASE tools shows, tool use has the benefit of creating a portable record of the design for a given system. When using CASE tools, designers can quickly exchange models and information in a manner that remains consistent from one designer to another. Augmenting a CASE tool used for design with automatic performance model generation capabilities would be an effective way to incorporate performance analysis into the early stages of the software development lifecycle.

1.2. The Converter Tool

The UCM2LQN converter is just such an automated tool that bridges the gap between a CASE tool for design in the form of the UCMNav and CASE tools for performance analysis in the form of LQNS and ParaSRVN. Designers who use UCMs for their high-level design can enter their models in the UCMNav and generate LQNs suitable for use with either the LQNS analytical solver or the ParaSRVN simulator. The performance results from LQNS and ParaSRVN can then be incorporated back into the design. The automation of the conversion of the design into a performance model maintains a consistency between the two models that is hard to achieve by traditional manual means. This approach not only has the advantage of improving the final product by allowing it to be designed with performance in mind, but it also means that there can be less of a distinction between designers and performance analysts.

Using the UCM2LQN converter in conjunction with LQNS and ParaSRVN means that a software designer does not need to be a performance analyst in order to get a performance analysis of a design. There is still a requirement to specify the appropriate performance data - such as service demands by responsibilities, arrival rates at start points, branching probabilities, loop repetitions, and device speed factors - in the UCM in order to get meaningful results. However, some of these values like service demands can be approximated by using a budgeting approach and supplying values based on an estimate of much time operations have to complete [42] [45]. The results from the performance analysis can then be used to confirm the time budget or show where the system the time budgets can be met. The designer can then fine tune the budgeting values or modify the design of the system based on those results. All without requiring an in-depth grounding in performance analysis.

1.3. Contributions Of This Thesis

This thesis describes the relationship between design scenarios as defined by UCMs and performance models in the form of LQNs. It also develops a conversion algorithm to generate the latter from the former. The thesis makes the following contributions:

- identification of corresponding constructs and patterns of interaction between the UCM and LQN notations (see Chapter 3)

- algorithm for traversing the internal UCM model of the UCMNav (see Section 5.1.)
- algorithm for detecting component boundary crossings in UCMs (see Section 5.3.)
- algorithm for interpreting the nature of messaging between UCM components (see Section 5.4.)
- algorithm for creating LQN objects based on UCM constructs and directing the traversal of the UCM accordingly (see Section 5.5.)
- validation of the UCM2LQN converter using “in-house” UCM models (see Chapter 6)
- testing of the UCM2LQN converter using UCM models originating from industry (see Chapter 7)

1.4. Thesis Organization

This thesis is organized in the following manner. Chapter 1 is a contains a basic description of the UCM2LQN tool and describes the motivation behind this research. Chapter 1 also provides a list of contributions made by the thesis. Chapter 2 describes the UCM and LQN notations in more detail, using a common example to for illustration purposes. It also describes the UCMNav tool used for editing UCMs and the LQNS and ParaSRVN tools used for LQN analysis. Chapter 3 introduces the basic corresponding constructs between the UCM and LQN notations, as well as more complex corresponding patterns of interaction that can be modeled using both notations. Chapter 4 describes the strategy used to integrate the UCM2LQN tool into the UCMNav editor. It describes the class inheritance and containment relationships of both tools. Chapter 5 explains the path traversal and LQN object creation algorithms used in UCM2LQN. Chapter 6 shows “in-house” models used to validate the conversion algorithms. Chapter 7 describes two models originating from industry that were used to further test the UCM2LQN converter. Finally, Chapter 8 contains the conclusions.

Chapter 2 - Background

This chapter covers background information on Use Case Maps, Layered Queueing Networks, and building performance models.

2.1. Use Case Map Background

The Use Case Map (UCM) notation results from research instigated by Professor R. J. A. Buhr at Carleton University. UCMs represent scenarios being executed across a system. UCMs work at a level of abstraction high enough to enable the user to grasp the emerging behaviour of the system without getting lost in execution details. Compared to the Unified Modeling Language (UML) notation, UCMs fit in between requirements and UML behavioural diagrams. In UML Class Diagrams are used to describe how a system is constructed, but do not describe how it works, thus the need for a notation such as UCMs that can describe how a system works [15]. Collaboration Diagrams do provide a high-level description of how the system works, but UCMs provide a better visual representation of how scenarios unfold and thus a better understanding of the emerging behaviour of the system [8].

A healthy literature on UCMs is anchored by Buhr and Casselman's book on the notation [21]. An general understanding of how UCMs are used can be obtained from the following papers on: the context and level of abstraction applicable to the UCM notation [15][16]; the application of UCMs to distributed systems [3][5][6]; the application of UCMs to agent systems [18][19][20]; the use of UCMs to detect and avoid feature interactions [2][4] [20]; and the relationship of UCMs with other notations such as MSCs [12] and hierarchical state machines [11][13][14].

UCM usage and acceptance is steadily growing with an active user community anchored by the *www.usecasemaps.org* website. There is also an industry-led effort to make UCMs an ITU-T standard as part of a recognized User Requirements Notation (URN) [29][22].

2.1.1. UCM Notation

A UCM map is a collection of elements that describe one or more scenarios unfolding

throughout a system. This section introduces these elements, using the UCM notation described in [21] and [17], and explains how to use them using the simple example of a banking transaction at an automatic teller machine (ATM).

The basic building block of the UCM notation is the path, which is the visual representation of a scenario. In its most basic form a path is a line with a start point and an end point, represented by a filled circle and a bar respectively. As a scenario is executed one can imagine a token traversing the path from the start to the end. Since UCM is a concurrent notation there is no restriction as to the number of tokens that may traverse a given path or the position of any token on a path relative to any other token. Figure 2-1 shows a simple path corresponding to making a banking transaction at an ATM. The transaction starts with the insertion of a bank card in the ATM and is completed when the card is ejected from the ATM.

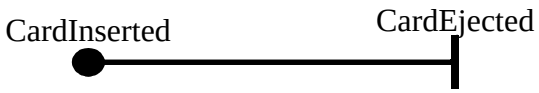


Figure 2-1: Simple UCM path for an ATM banking transaction.

UCM paths can be overlaid 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 in a system, as well as refinements of those resources. Figure 2-2 shows the path representing the ATM banking transaction overlaid on a component representing the ATM. In this case the component represents the ATM in its entirety.

A path can be refined to show more scenario detail with the addition of responsibilities. Responsibilities are denoted with an X shaped mark along the path. They represent functions that need to be accomplished at given points in the execution of the scenario. Figure 2-3 shows the banking transaction path, but refines it with the addition of responsibilities to read the information from the card, get the user's PIN, display a PIN request to the user, collect the PIN from the keypad, perform the banking transaction, and finally eject the card. The ATM is also refined by showing distinct components for the card reader, the keypad, the display, and the cash dispenser. A component representing the ATM customer as the user is also shown.

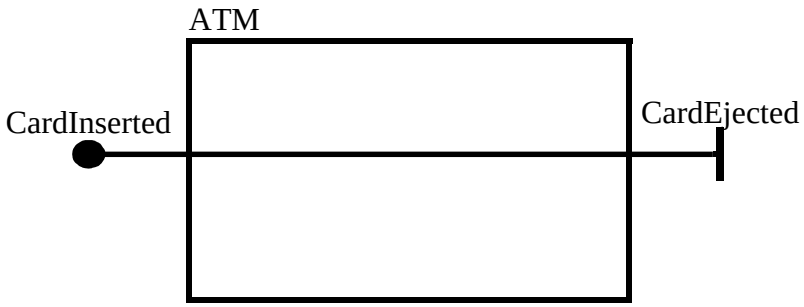


Figure 2-2: Simple UCM path for an ATM banking transaction overlaid on an ATM component.

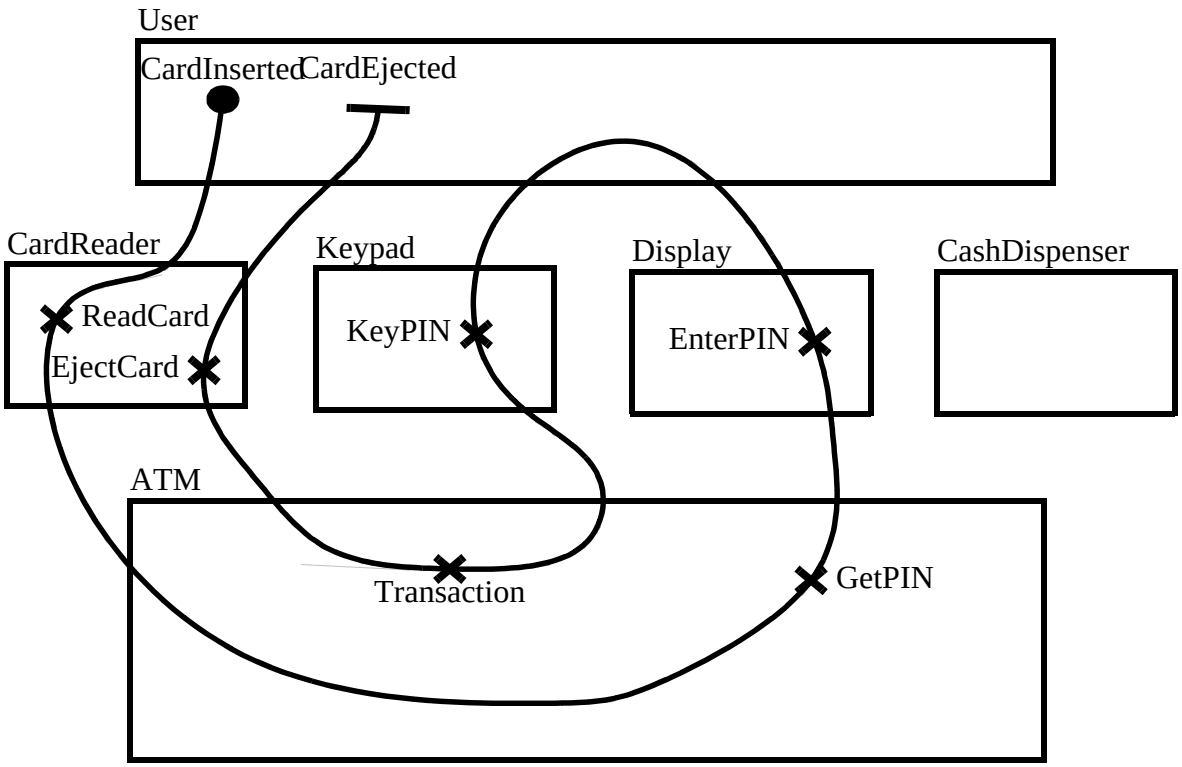


Figure 2-3: UCM with responsibilities and additional components for an ATM banking transaction.

Although responsibilities can be used to represent any kind of function at any level of abstraction, they are normally used to represent simple operations that can be considered atomic. Another construct, called a stub, is available to denote broader functionality that can be described

by another UCM called a plug-in. There may be several alternative plug-ins for any stub. A plug-in is a separately specified map that has further detail describing a given aspect of a scenario. A plug-in may commonly be thought of as a sub-map of the map with its referring stub, the plug-in map can also stand alone as a valid UCM in its own right. There are no restrictions as to the presence of further stubs in the plug-in map, although stub recursion may hinder the user's ability to navigate through the UCM and understand the scenario and system. Figure 2-4 shows the ATM banking example further refined with the substitution of a stub for the responsibility of performing the banking transaction. Figure 2-5 shows a withdrawal plug-in for the banking transaction stub. Other kinds of transactions, such as deposits, account balance inquiries, balance transfers, etc., can be described with other plug-ins of their own.

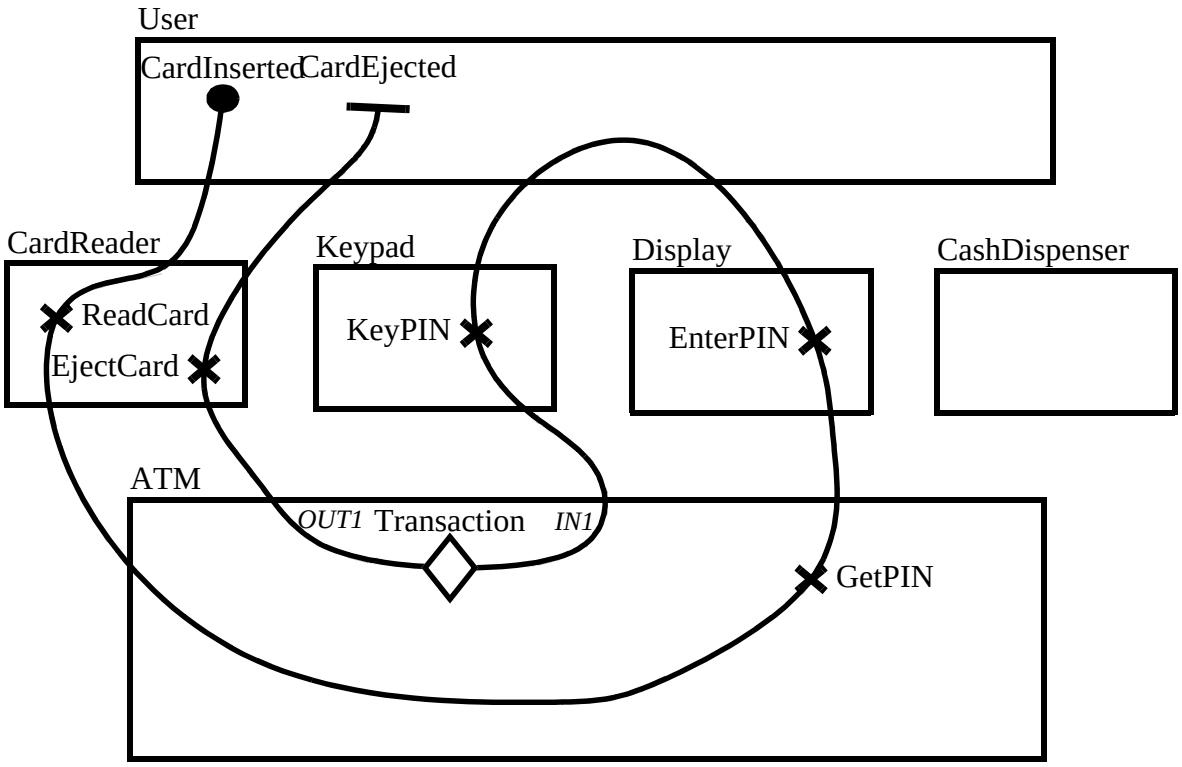


Figure 2-4: UCM for an ATM banking transaction with a stub for the transaction.

The UCM synchronization construct is used to indicate a place where parallel path segments split or gather. In general, synchronizations are used as either logical AND forks (a single path splitting into two or more parallel paths) or logical AND joins (two or more parallel paths

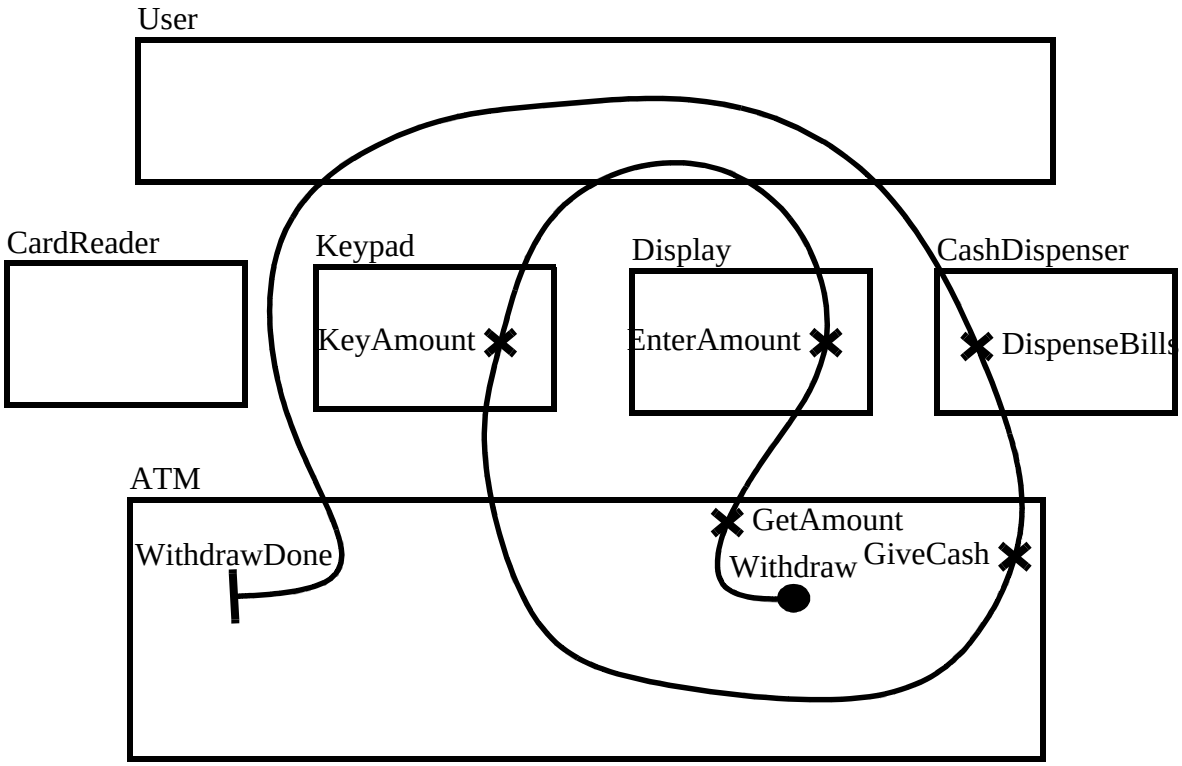


Figure 2-5: Withdrawal UCM used as a plug-in for the Transaction stub in Figure 2-4.

gathering into a single path). There is no restriction as to how many paths must lead in or out of a synchronization. Any synchronization that joins paths requires that tokens travelling along each incoming path must all arrive at the synchronization before path traversal can proceed past the synchronization. Figure 2-6 shows the withdrawal plug-in refined with a path segment that requests account information from a central bank database while in parallel the ATM displays a message asking the user to wait.

Scenario alternatives are shown using OR forks (a single path splitting into two or more alternative paths) and OR joins (two or more alternative paths gathering into a single path). An OR fork indicates that a choice between alternatives is being made and only one of the possible branches may be traversed after the fork. An OR join indicates that at least one of the possible paths leading into it needs to be traversed before proceeding further. Figure 2-7 further expands the ATM example by adding an alternative path that cancels the transaction if the user presses the “cancel” button or is unable to provide a correct PIN.

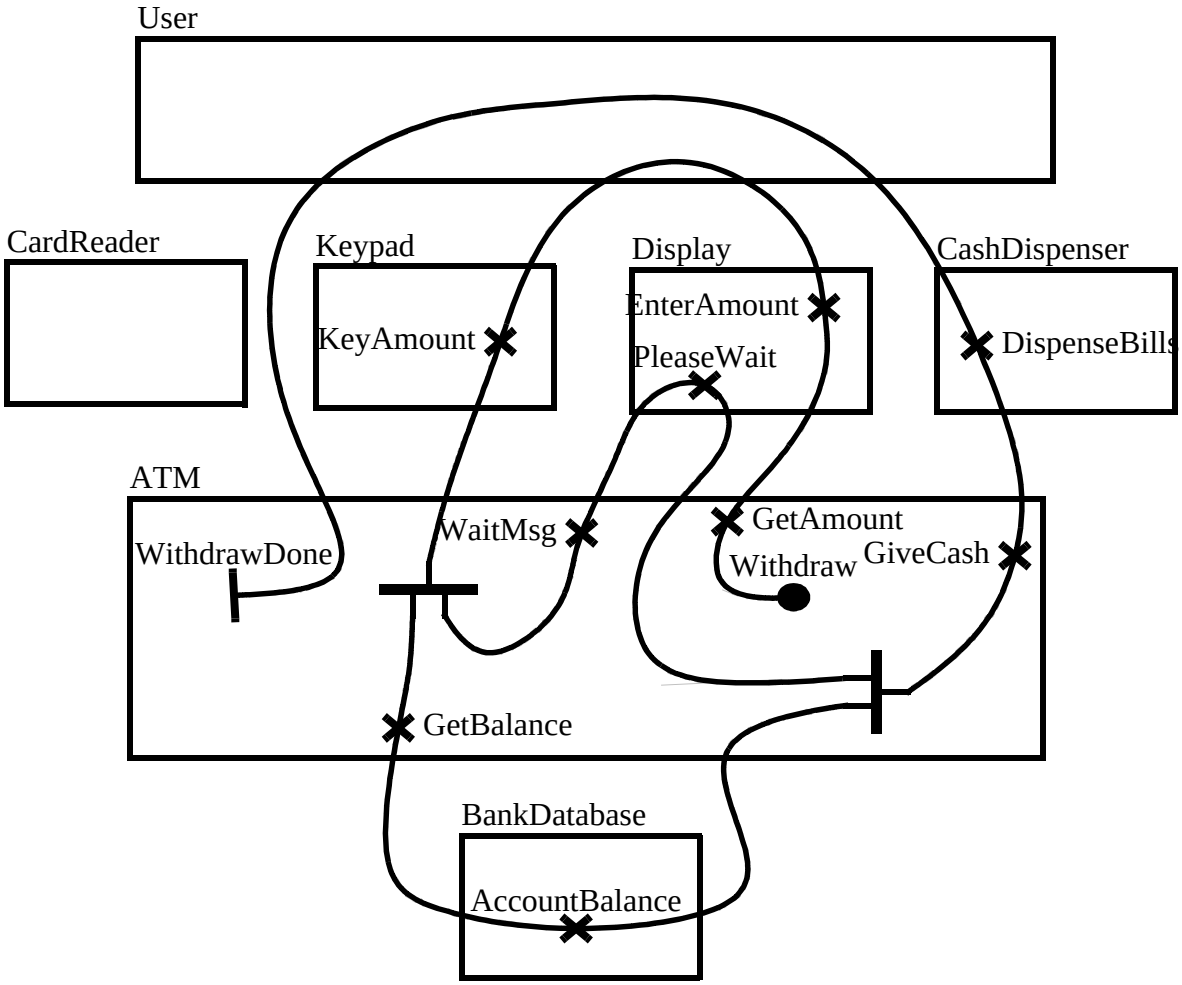


Figure 2-6: Refined withdrawal UCM with a parallel path segment getting account information from the bank database used as a plug-in for the Transaction stub defined in Figure 2-4.

OR joins and forks can be used to create informal looping structures in UCM, but there is a loop construct as well. The loop construct indicates that the body of the loop is traversed a certain number of times before the traversal of the main path resumes. Figure 2-8 shows a loop added to the ATM example in order to show that an incorrect PIN may be re-entered a certain number of times. For the work described in this thesis it is assumed that all UCM loops use the explicit loop construct. Other looping structures are not converted to LQNs.

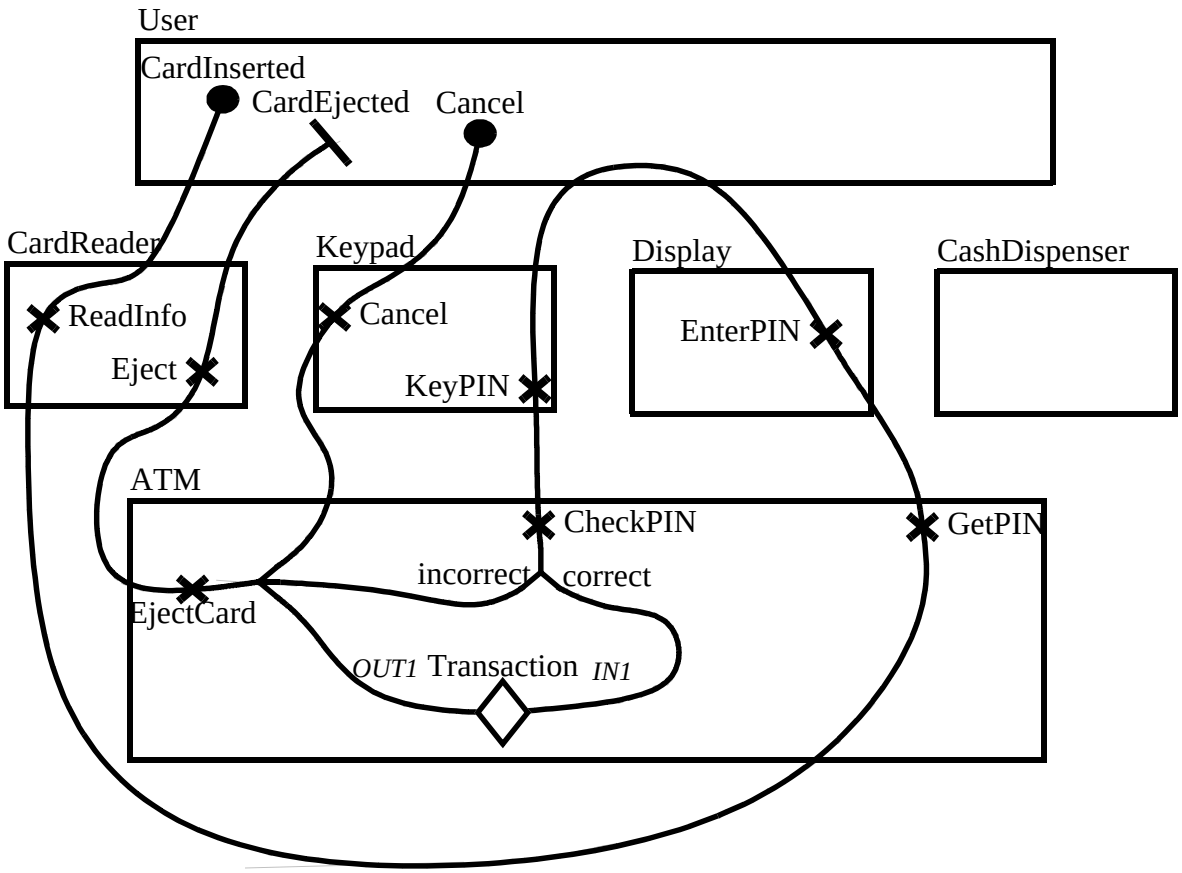


Figure 2-7: UCM for an ATM banking transaction with alternative paths that end the transaction.

2.1.2. The UCM Navigator

The UCM Navigator (UCMNav) is a UCM editing tool developed at Carleton University by Andrew Miga [28]. It is currently used at Carleton University and the University of Ottawa, as well as within Nortel Networks and Mitel Networks. The UCMNav allows the user to draw and modify UCMs, add comments and descriptions for the design and/or individual elements, specify system devices, integrate multiple UCMs into an overall design, and even generate Message Sequence Charts (MSC) from UCMs [12].

Figure 2-9 shows a screen shot of the UCMNav. The top menu bar provides access to the file input and output functions, various editing options and preferences, and advanced options

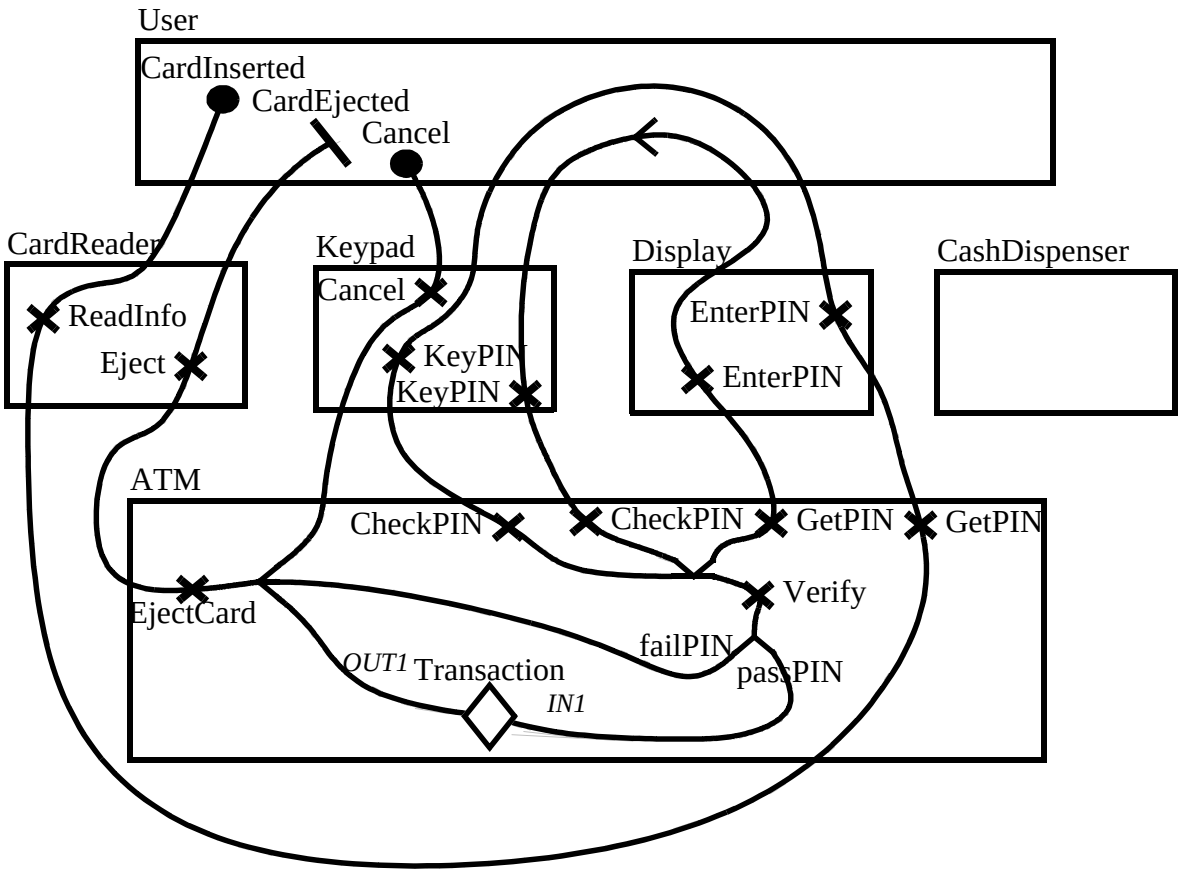


Figure 2-8: UCM path for an ATM banking transaction with a loop to get the correct PIN number.

such as MSC or LQN generation. The UCM drawings are done on the editing canvas which is the large white area in the upper left portion of the UCMNav window. Graphical editing is done using tools from the tool palette right above the canvas and below the menu bar. The smaller gray areas right of the canvas are comment and description boxes. They are used to display additional information about the UCM elements and the overall design.

The UCM designs are represented internally as hypergraphs and saved as XML files. The hypergraph model is explained in further detail in Chapter 4. For details on the UCMNav XML document type definition please refer to [7].

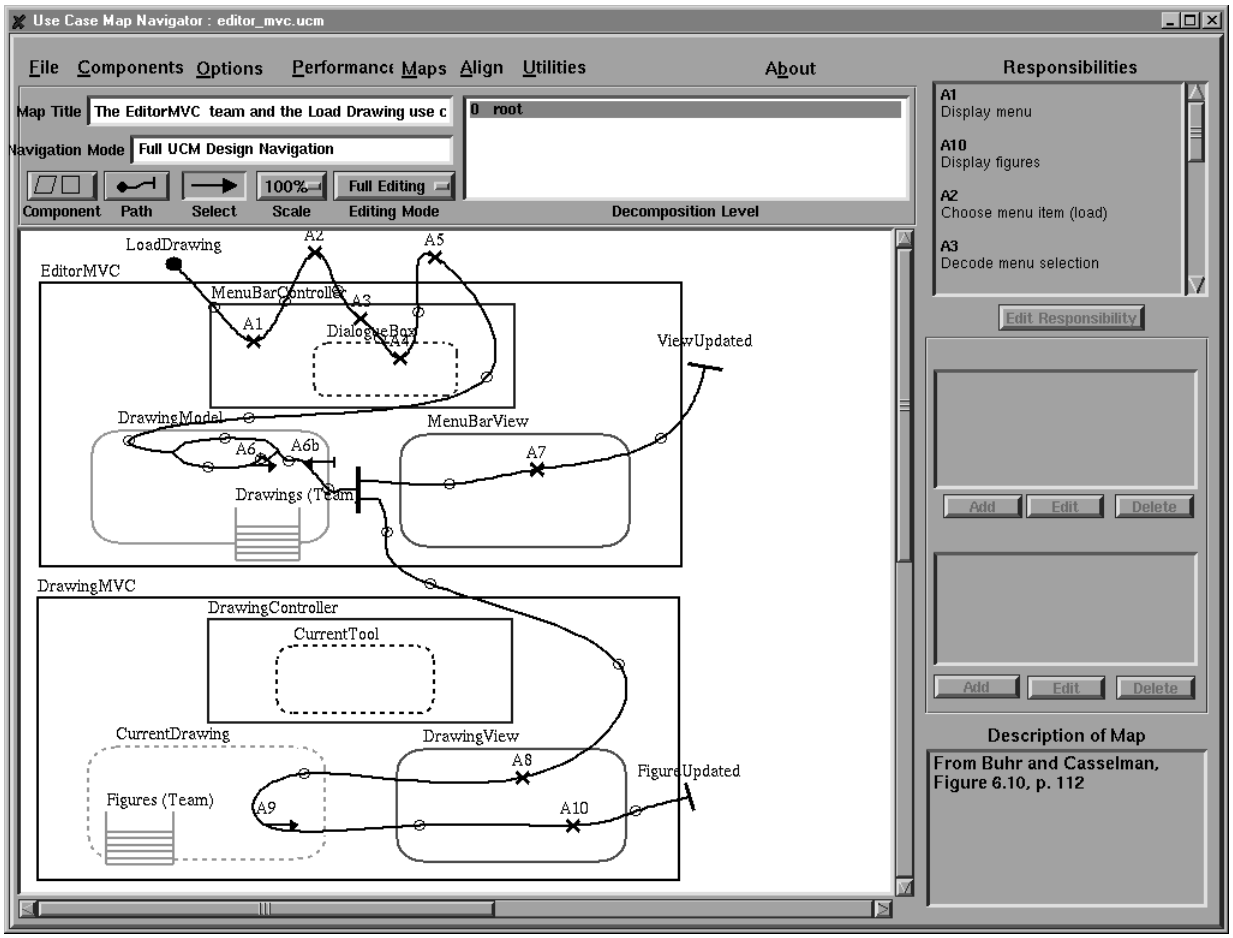


Figure 2-9: Screen shot of the UCMNav.

2.2. Layered Queueing Network Background

Queueing Networks are based on a customer-server paradigm. The customers make service requests of the servers and these request are queued at the server until it can service them. Traditional queueing networks can model only a single set of customer-server relationships. In practice this means that queueing networks can only model hardware resources as pure servers and software tasks as pure customers.

Layered Queueing Networks (LQN), or Stochastic Rendez-Vous Networks (SRVN) as they were previously called, allow for of an arbitrary number of client-server levels [39][48]. An LQN can thus model intermediate software servers and be used to detect software deadlocks and

software as well as hardware performance bottlenecks [34]. The layered aspect of LQNs makes them very suitable for evaluating the performance of distributed systems, such as telecommunication networks [36][50][51].

2.2.1. LQN Notation

LQNs can model both software and hardware resources. The basic software resource is a task. A task is any software object that has its own thread of execution. The basic hardware resource is a device. Typical devices are CPUs and disks. Figure 2-10 shows the visual notation for tasks and devices.

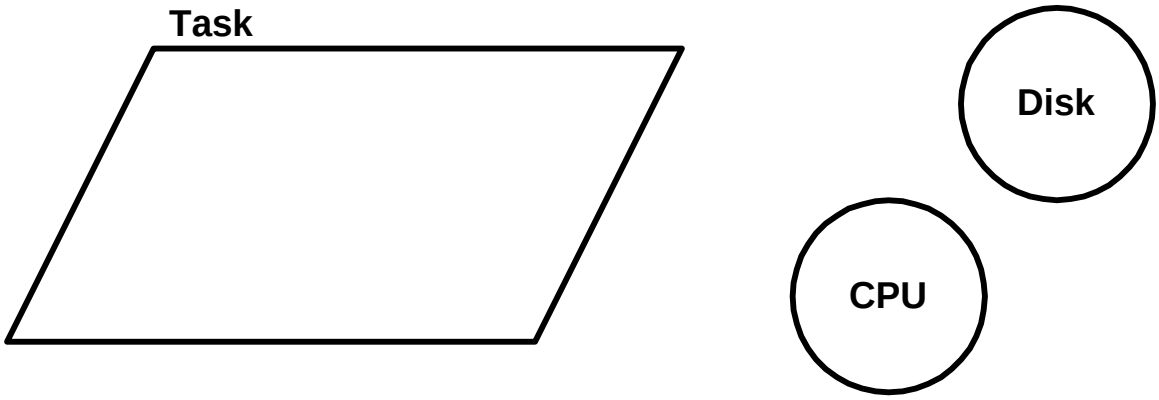


Figure 2-10: LQN task and CPU and disk devices.

Service requests are shown in LQN by messaging arrows. Tasks may both send and receive messages, whereas devices may only be pure servers that receive messages. Whenever tasks model pure clients which only send messages, they are called reference tasks.

There are two types of messaging: asynchronous and synchronous. Asynchronous messages are sent by a task and do not require a reply. The sending task continues executing normally after sending an asynchronous message. Synchronous messages are blocking calls that require a reply. A task sending a synchronous message suspends execution until it receives a reply to that message. Figure 2-11 shows an example of asynchronous and synchronous messaging with Task_A acting as a reference task.

Tasks receive service requests at designated interface points called entries. Entries corre-

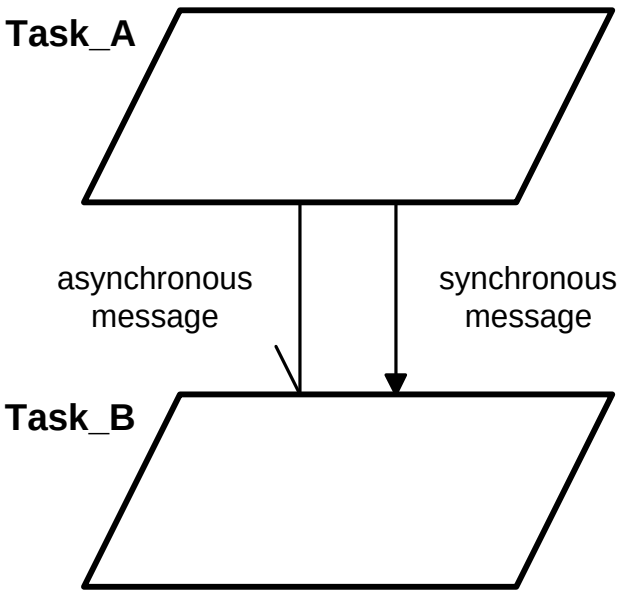


Figure 2-11: LQN synchronous and asynchronous message arrows.

spond to service access point for a task. There is a different entry for every kind of service that a task provides. An entry may be defined atomically, with its own hardware service demands and calls to other tasks. Alternately, an entry may be defined by blocks of smaller computational blocks called activities. Activities have their own hardware service demands and can make calls to entries in other tasks, as shown in Figure 2-12. They can be arranged in sequences, as well as in parallel (AND forks and joins) or alternative (OR forks and joins) configurations. An activity can also make repeated service calls in order to model repetitive behaviour.

The LQN notation supports three types of calls: asynchronous, synchronous, and forwarded calls. An asynchronous call does not involve any kind of blocking on the part of the sending task. A synchronous call means that the sending client task blocks until it receives a reply. In a forwarding call, the sending client task makes a synchronous call and blocks until it receives a reply, the receiving intermediate server task partially processes the call and then forwards it to another server which becomes responsible for sending a reply to the blocked client task. The intermediate server task can continue operation after forwarding the call and there can be any number forwarding levels. Figure 2-13 shows the time semantics used to interpret the types of LQN calls.

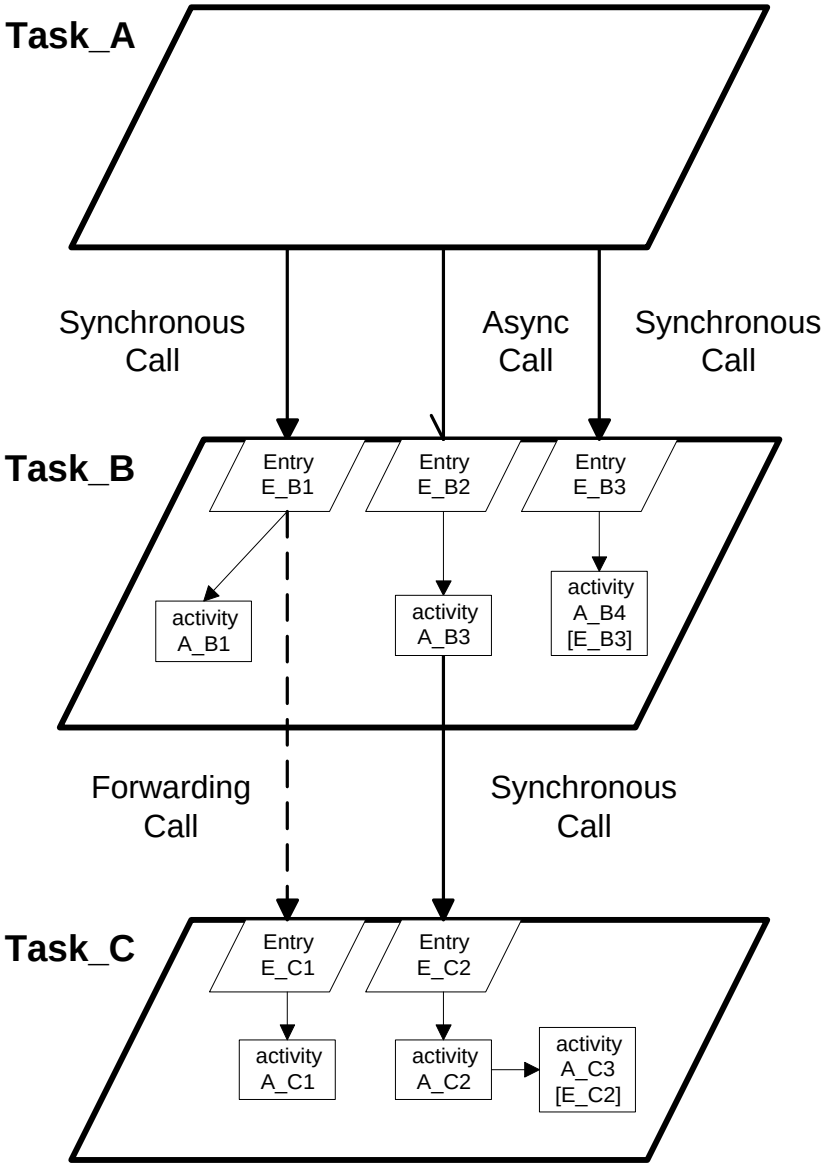


Figure 2-12: LQN with entry and activity detail.

If an entry receives a synchronous service request it is responsible for sending a reply after the request has been completed. Such a reply is implied for entries that are defined atomically. However for entries defined by activities a reply activity needs to be explicitly designated, such as A_C3 in Figure 2-12. The replying activity is usually assumed to be the last activity for the entry if no forks occurred earlier.

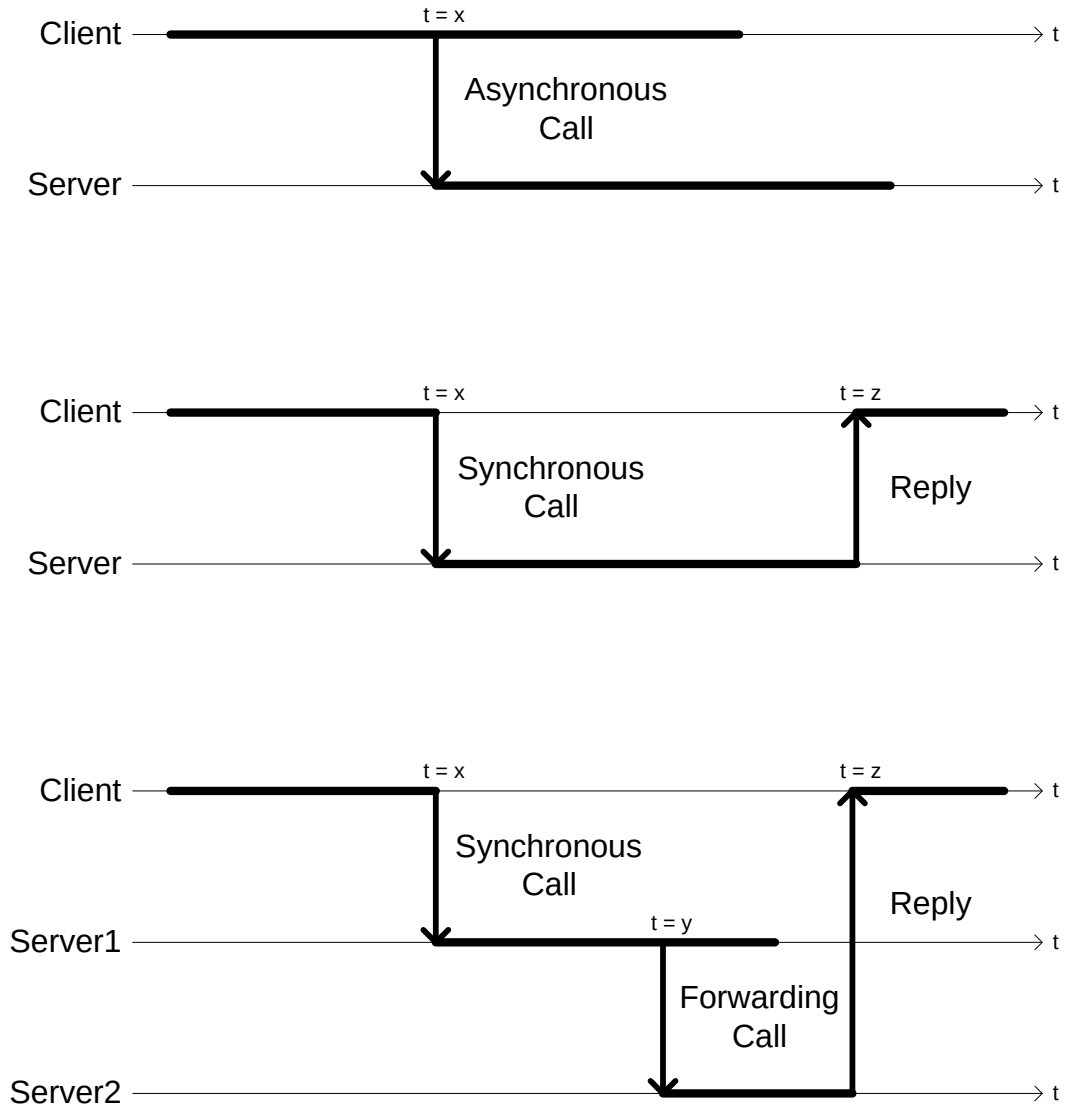


Figure 2-13: Time semantics of LQN asynchronous, synchronous, and forwarded calls.

An entry receiving a synchronous service request may also forward it to other entries which then become responsible for sending the reply to the original caller. In the case of a forwarded call, the original calling task remains blocked until it finally receives the reply at the end of the forwarding chain.

Entries and activities can be used to fully describe a task's functions. Figure 2-14 shows an LQN based on the ATM banking transaction model introduced in Section 2.1. The ATM task is

showing entry and activity detail corresponding to a withdrawal transaction. Please note that OR relationships are shown using a circle with a + sign inside and AND relationships are shown using a circle with an & inside.

For the purposes of this research it is assumed that LQN activities are the basic building blocks of LQN models. An LQN activity is constructed to directly correspond to each UCM responsibility. Further correspondences between LQN and UCM models are introduced in Chapter 3.

2.2.2. Applying LQN

LQN models need to be simulated or solved in order to extract performance metrics from them. This requires more data in addition to the execution and calling structure modeled by the notation described in Section 2.2.1.

Since all software runs on hardware, devices must have a speed factor specified that indicates their response time per operation. Reference tasks also need to have arrival rates specified which indicate the distribution and frequency of their initial service requests. It is also necessary to indicate whether the system supports open or closed arrivals.

If entries are used in conjunction with activities to describe the behaviour, then they are not required to have any hardware demands specified since those are specified by the activities themselves. Each activity needs to have its hardware demands specified, as does each entry that is not described by activities. Alternative OR forks must specify the probability of execution of each branch. Of course, parallel AND forks have an equal 100% probability of each branch being executed. Any entry or activity making a call must also specify the probability of that call being made. If an activity is repeated it also needs to have a repetition count.

This type of information is not necessarily included in UCM designs, but the UCMNav does have facilities to specify it as additional annotations to the path elements involved. If the information is missing, then the UCM2LQN converter assigns a default value for the missing parameters.

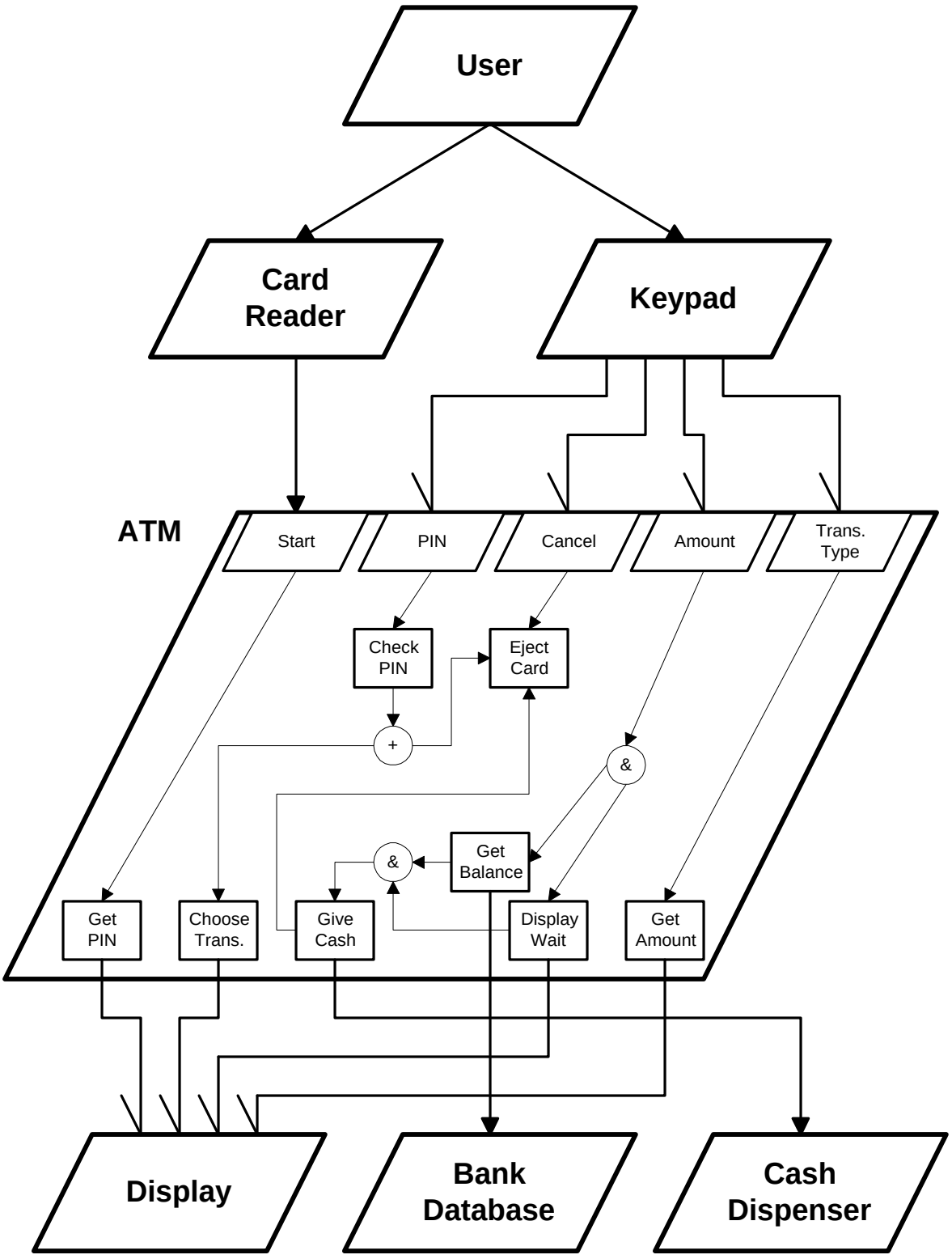


Figure 2-14: LQN for an ATM withdrawal transaction with entry and activity detail shown for the ATM task.

2.2.3. LQN Tools

This section introduces the tools available to support the LQN notation. There are two tools available from Carleton University that can be used to solve LQN models and get performance metrics. The Layered Queueing Network Solver (LQNS) solves LQN models analytically, whereas the ParaSol Stochastic Rendez-Vous Network Simulator (ParaSRVN) simulates LQN models using the ParaSol simulation system. A third tool, the Java Layered Queueing Network Definition Editor (jLqnDef) can be used as an LQN editor and to display a graphical view of the LQN file.

All three tools use the same text file format, part of which is described in Section 2.2.3.3.

2.2.3.1. LQNS

The Layered Queueing Network Solver (LQNS) is a tool developed at Carleton University by Greg Franks as part of his Ph.D. research [25]. LQNS is an analytic solver that breaks the LQN layers down into separate queueing network sub-models. The individual queueing networks can be solved analytically using mean value analysis (MVA). The MVA results for each sub-model are then used to fine-tune the MVA parameters for the other sub-models it is connected to and the MVA is performed anew. This process is repeated either for a maximum number of iterations or until the results converge on a convergence value specified by the user.

LQNS can use different layering techniques for the sub-models. The default is batched layering where the layers are composed of as many servers as possible. The two other layering techniques that are implemented are loose layering, where layers have only a single server, and strict layering, which is based on the Method of Layers [39].

Although LQNS can solve a broad variety of models, it cannot solve models that cannot be hierarchically decomposed. Thus its analytic model cannot not handle corresponding forks and joins that occur in different tasks, and only works well for corresponding forks and joins that occur in the same task.

2.2.3.2. ParaSRVN

The ParaSol SRVN (a.k.a. LQN) simulator (ParaSRVN) uses the ParaSol simulation envi-

ronment. ParaSol can simulate multithreaded systems that support transactions and provides built-in statistics for monitoring simulation objects [32][33].

ParaSrvn simulates LQNs by creating tokens for each call and following those tokens through the system. The performance metrics are arrived at by recording the wait times and other statistics for each token. Since ParaSRVN simulates the execution of the system rather than attempting to generate analytical solutions, it can deal with any type of model. ParaSRVN is more robust than LQNS in this respect and can solve those models that cannot be hierarchically decomposed.

Since ParaSRVN generated results by running the simulation of the LQN and recording waiting times for the tokens, it requires a large number of runs in order to gather data that's statistically meaningful. Thus ParaSRVN is appreciably slower than LQNS and requires very long simulations in order to generate accurate results.

2.2.3.3. LQN File Format

This is a quick overview of the text file format used by the LQN tools. An LQN file is composed of a general information section, a processor information section, a task information section, an entry information section, and an activity information section - in this exact order.

The general information section is denoted by a 'G' followed by a comment on the model, a convergence value, and iteration limit, an optional iteration interval for printing intermediate results and an optional under-relaxation coefficient. The general information section ends with a '-1' as an end-of-list symbol.

The processor information section begins with a 'P' followed by the number of processors in the section and the declaration of each processor. The declaration of each processor begins with a 'p' followed by an integer processor id, a scheduling flag ('f' for first-come, first served scheduling), and an optional quantum value, multiserver flag, replication flag, and processing rate. The processor information section ends with a '-1'.

The task information section begins with a 'T' followed by the number of tasks in the list. Each task is denoted by a 't', a task name, a task scheduling type ('r' for reference task and 'f' for first come first served), a listing of the entries for the task, and a processor id. There are some

other optional fields that are not used by the UCM2LQN converter. The task information section ends with a '-1'.

The entry information section begins with an 'E' followed by the number of entries in the list and the entry declaration for each entry. An entry may be described by more than one declaration. The declaration for an entry that is defined by activities begins with an 'A' followed by the entry's name and the name of its first activity. The declaration for an entry that forwards to another entry begins with an 'F' followed by the entry name, the name of the entry being forwarded to, a forwarding probability, and ends with a '-1'. The declaration for an entry that has a defined arrival rate begins with an 'a' followed by the entry's name and the arrival rate. The entry information section ends with a '-1'.

The activity information section is composed of activity information lists for each task. The activity information list for a task begins with an 'A' followed by the task's name and the activity declarations. Each activity may be described by more than one declaration. The declaration for an activity with a defined service time begins with and 's' followed by the activity's name and the service time. The declaration for an activity with known phases begins with an 'f' followed by the activity name and the phase number. The declaration for an activity that makes a synchronous call begins with a 'y' and is followed by the activity name, the target entry name, and the number of calls per phase. The declaration for an activity that makes an asynchronous call begins with a 'z' and is followed by the activity name, the target entry name, and the number of calls per phase. The activity declarations are followed by a ':' and the activity connection list for the task. Activity connections are described as follows:

- *A1 -> A2* - means that activity A1 is followed in sequence by activity A2
- *A1 -> (0.5) A2 + (0.5) A3* - means that activity A1 is followed by either activity A2 OR activity A3, the numbers in brackets denote the probability of choosing that particular activity
- *A1 -> A2 & A3 & A4* - means that activity A1 is followed by all three of the activities A2 AND A3 AND A4
- *A1 -> 4*A2, A3* - means that activity A1 is followed by a repeated activity A2, the number is the count of how many times activity A2 is executed, A2 is then followed by activity A3
- *A1 + A2 -> A3* - means that activity A1 OR A2 is followed by activity A3
- *A1 & A2 -> A3* - means that activity A1 AND A2 are both followed by activity A3

- *A1 [E1]* - means that activity A1 generates a reply to the call received by entry E1

Activity connections are separated from each other by semi-colons, ';'. The activity information section ends with a '-1'.

2.3. Derivation of Performance Models from Design Specifications

The UCM2LQN converter introduced in this thesis provides an automated way of generating LQN performance models from UCM design specifications. This section provides a brief overview of some of the other non-automated methods for deriving performance models from design specifications as presented in [9].

The Software Performance Engineering (SPE) methodology introduced by Smith in 1990 [43] was the first bonafide attempt at integrating performance analysis into the software development process. SPE uses a software execution model and a system execution model. The software execution model is based on execution graphs. It is used to derive the resource demands of the software. This is a process that is analogous to the activity graph analysis used in [49]. The software demands are then used as an input into the system execution model to get performance metrics. The system execution model is based on queueing networks and reflects the computing platform as a whole, including both software and hardware concerns. Smith and Williams have also developed a tool to support SPE, called SPE-ED, that is used to simulate or solve the queueing networks from the execution model [44]. SPE-ED does not automatically generate queueing networks directly from the software execution model though.

Another approach by Cortellessa and Mirandola [23] integrates information from different UML diagrams to derive a performance model of the system. This approach extends the SPE methodology presented in [43] and [46]. The use case diagrams are augmented with probabilities for the edges linking the user to the system and in conjunction with their corresponding sequence diagrams are used to derive a meta-execution graph for the system. The execution graphs are then built incrementally from individual sequence diagrams by integrating them into the meta-model. The approach only works with sequence diagrams of synchronous systems. The UML deployment diagram is then used to build an execution model and combined with the execution graphs. The combined model is then solved using the SPE approach.

Gomaa and Menasce [26] also use UML diagrams as a starting point for deriving performance models. Starting off with class diagrams, as a static model of the system, and collaboration diagrams, as a dynamic model, Gomaa and Menasce provide additional performance information and translate everything into an XML notation. This allows them to capture both the architecture and performance parameters in a single notation. The resulting model is then matched against patterns of component interconnections for client/server systems. Based on this matching they can then be turned into extended queueing networks that are solved using Markov chain analysis or analytical methods.

Chapter 3 - Correspondences Between UCM and LQN

This chapter deals with the correspondences that were identified between the UCM and LQN models. It covers corresponding constructs between the two notations, corresponding ways to model basic patterns of interaction between components, as well as ways to model more complex patterns of interaction.[35]

3.1. Corresponding Constructs

There are some constructs that correspond directly between the UCM and LQN notations. These constructs are the building blocks for the more complex correspondences described later in this chapter and are listed in Table 3-1.

UCM Construct	LQN Construct
responsibility	activity
component	task
device	device
service	task with a dedicated processor

Table 3-1: Corresponding UCM and LQN constructs.

The UCM notation allows components to be of several different types - team, process, object, interrupt service routine, pool, agent, or other - but for the purposes of generating LQNs every component is treated as if it were a task. There are utilities that can be used on the resulting LQN in order to aggregate superfluous tasks.

3.2. Basic Patterns of Interaction

This section shows the basic correspondences between UCMs and LQNs. We use elementary UCM systems that illustrate one interaction type at a time. The UCMs are shown as outputs from the UCMNav. The LQNs are shown as visual output from the jLqnDef tool and as such their appearance differs slightly from the LQN notation as introduced in Section 2.2.1. Currently jLqn-

Def does not display activity connections graphically so textual annotations are provided to do so. The LQNs shown were saved as LQNS files and are syntactically correct and can be solved with LQNS. Figure 3-1 shows how the graphical output from jLqnDef should be interpreted. Table 3-2 explains how the textual annotations should be read.

Textual Activity Connections	Interpretation
A1 -> A2	activity A1 is followed by activity A2 (sequence)
A1 [E1]	activity A1 sends a reply corresponding to the call received by entry E1
A1 -> A2 & A3	activity A1 is followed by activity A2 and activity A3 (AND fork)
A1 & A2 -> A3	activity A1 and activity A2 are followed by activity A3 (AND join)
A1 -> (x)A2 + (y)A2	activity A1 is followed by activity A2 with a probability of ‘x’ or by activity A3 with a probability of ‘y’ (OR fork)
A1 + A2 -> A3	activity A1 or activity A2 is followed by activity A3 (OR join)
A1 -> z*A2, A3	activity A1 is followed by activity A2 which is repeated ‘z’ times, after activity A2 has been repeated it is followed by activity A3

Table 3-2: Textual activity connections and their interpretations.

The underlying assumption behind the identification of calling patterns between components is that a message is sent every time a path leaves a component and a message is received every time a path enters a component. It it also assumed that a synchronous interaction always occurs in cases where the UCM path returns to a component it has previously passed through. This maximization of the synchronous interpretation of calls is desirable for LQN models because synchronous communication captures the layering aspect of the components, captures the scenario aspect of the messaging, and makes it easy to determine response times. In the LQN context, a synchronous call indicates that the calling thread or task blocks until it receives a reply. An LQN synchronous call does not necessarily dictate that the communication must always be implemented in a synchronous manner in the actual product.

Some UCM models may use other interpretations of messaging where, for example, a path passing through multiple components can denote a pattern of communication between the compo-

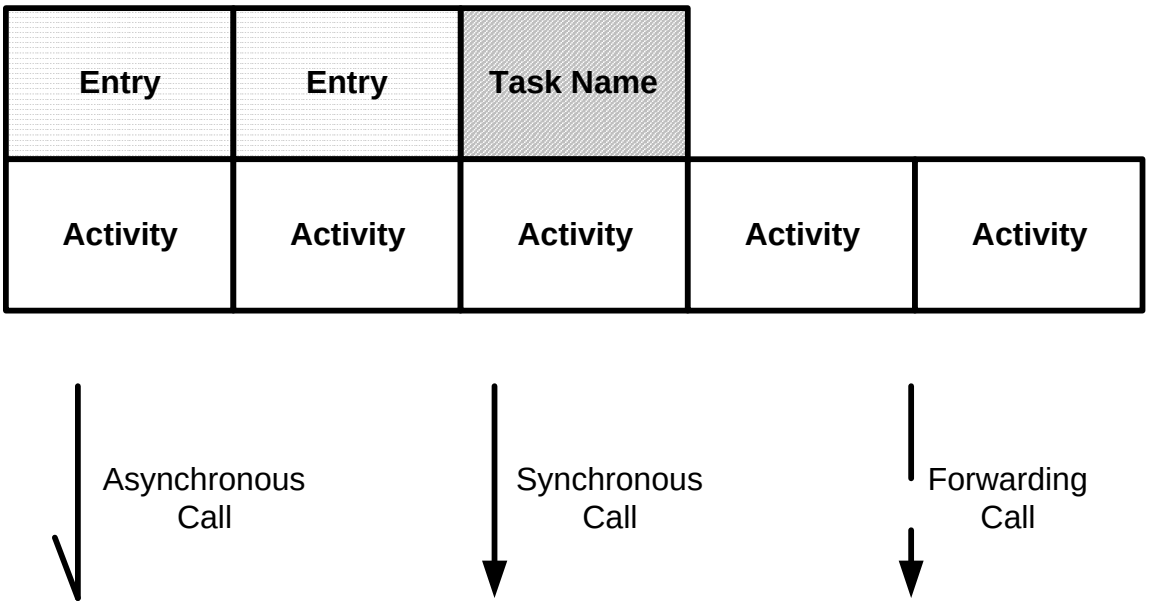


Figure 3-1: Interpretation of the graphical output from jLqnDef for a task and messaging arrows.

nents where multiple messages are exchanged [5]. Such models do not have a set LQN correspondence.

3.2.1. Synchronous Call and Return

A synchronous call is made whenever the UCM path crosses from one component to another and returns back to the original component. In the corresponding LQN model each call corresponds to an entry in the called task. The entry then leads to a succession of activities that correspond to the UCM responsibilities. The last activity in that succession points back to its entry when the call is ready to be returned. The call is shown in the LQN as a line with a filled arrowhead pointing from the activity that makes the call to the entry that is being called. Figure 3-2 shows the corresponding UCM and LQN models for a synchronous call and return.

3.2.1.1. Multiple Calls

Multiple synchronous calls are made whenever the UCM path crosses from one component to another, returns back to the original component, and repeats the same pattern. In the LQN

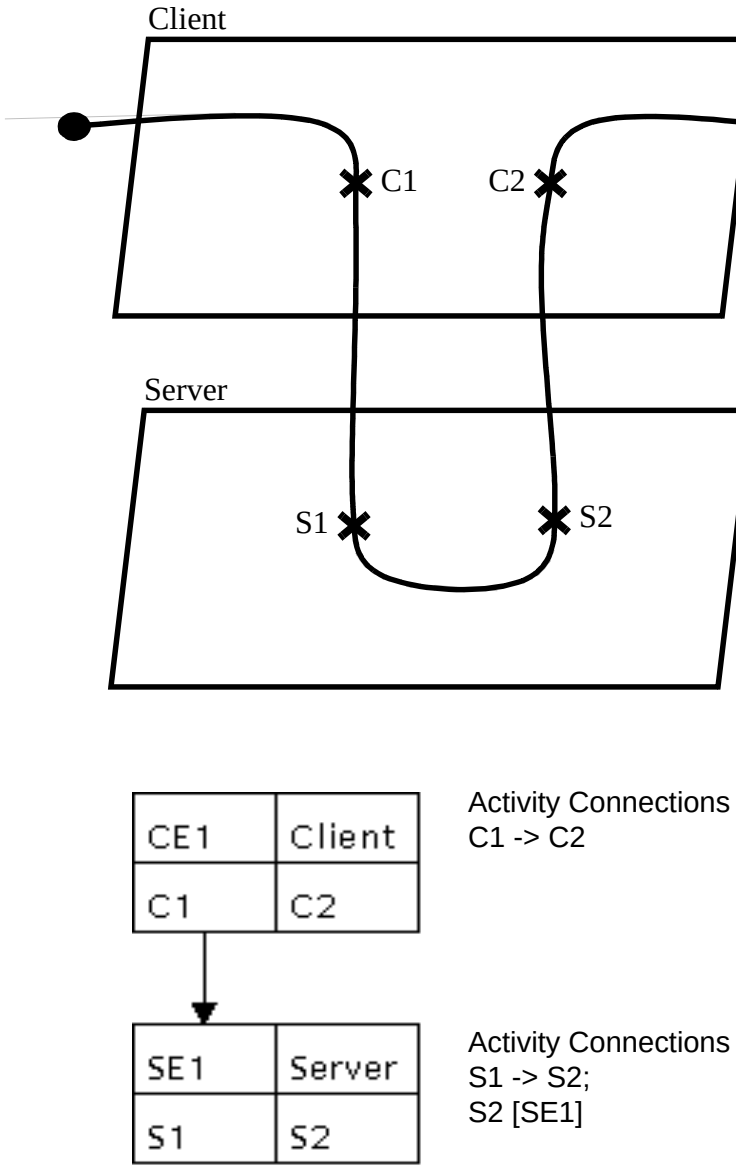


Figure 3-2: Corresponding UCM and LQN models for a simple synchronous call and return.

model each separate call corresponds to a separate entry in the called task. Each entry then leads to a succession of one or more activities that correspond to the UCM responsibilities. Figure 3-3 shows corresponding UCM and LQN models for two successive synchronous calls and returns

3.2.2. Asynchronous Call

An asynchronous call is made whenever the UCM path crosses from one component to

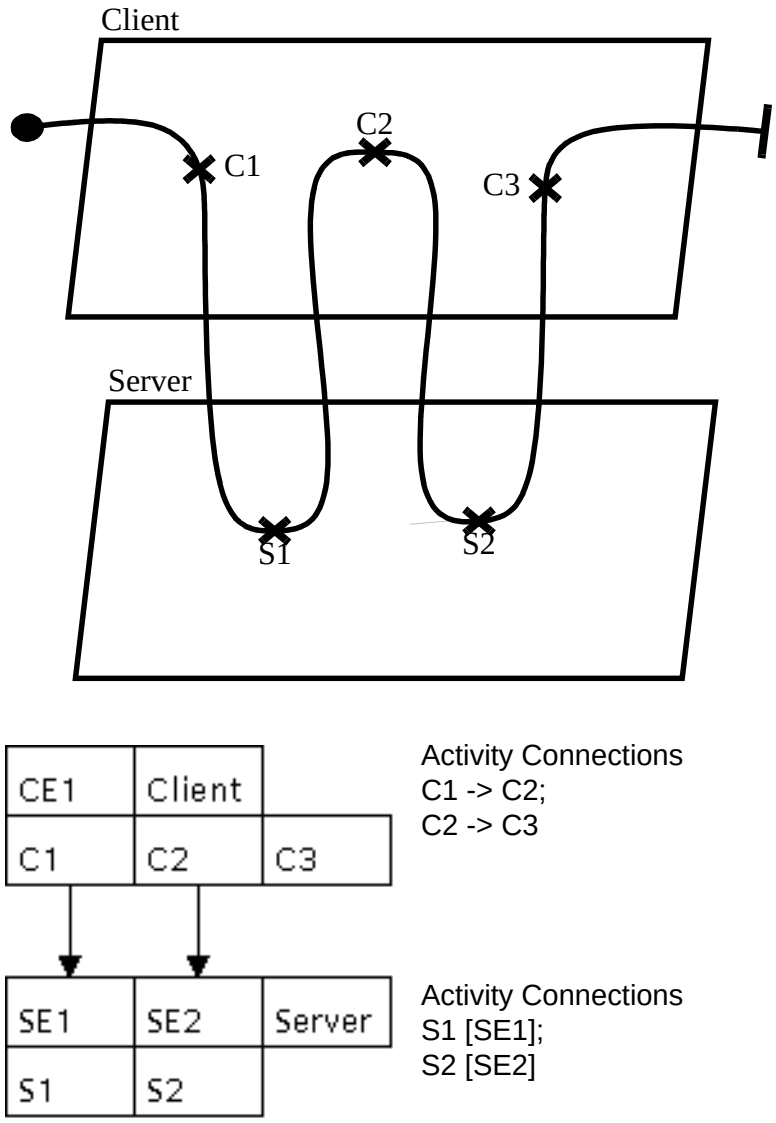


Figure 3-3: Corresponding UCM and LQN models for successive synchronous calls and returns.

another and does not return back to the original component. In the LQN model each asynchronous call corresponds to an entry in the called task. The entry then leads to a succession of activities that correspond to the UCM responsibilities. An asynchronous call is shown in the LQN as a line with an empty arrowhead pointing from the activity that makes the call to the entry that is being called. Figure 3-4 shows the corresponding UCM and LQN models for an asynchronous call.

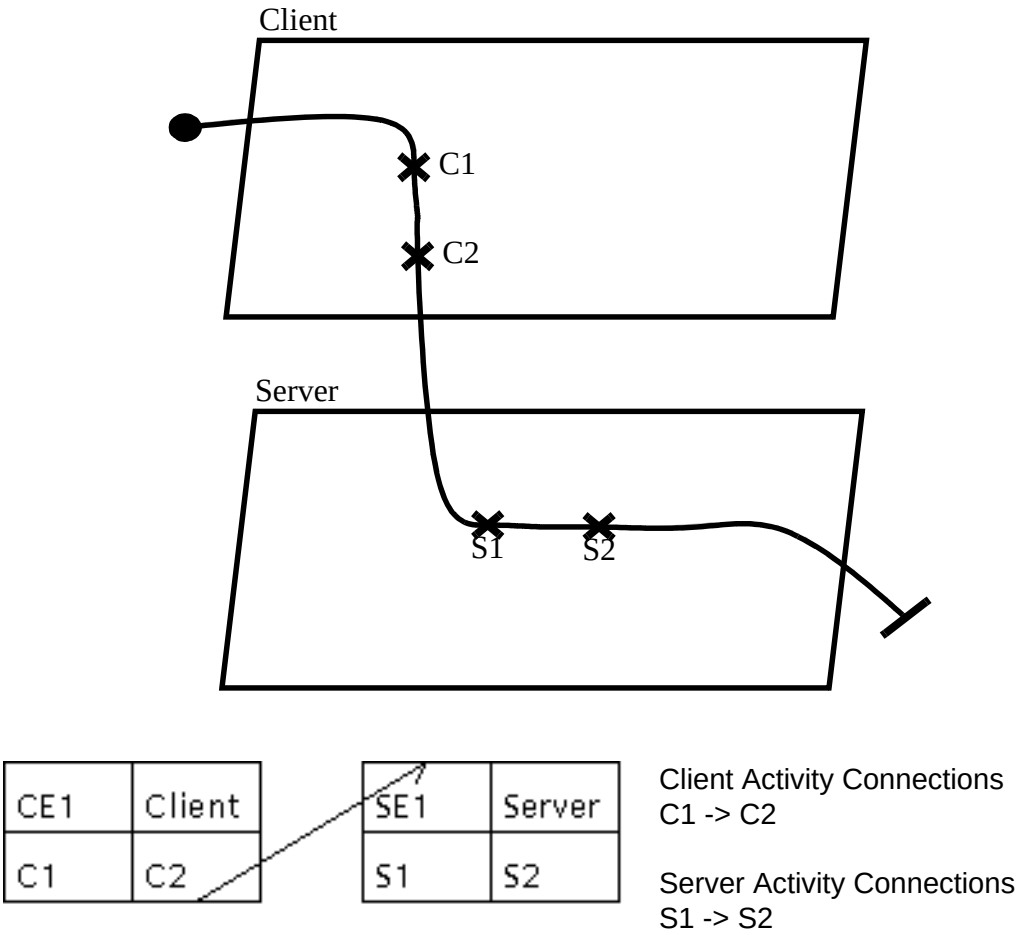


Figure 3-4: Corresponding UCM and LQN models for a simple synchronous call and return.

3.2.3. Forwarding

A call forwarding is made whenever the UCM path crosses from one component to another, and then to several others, before returning back to the original component. The original call is synchronous for the original component, but the forwarding is asynchronous for the other components. In LQNs forwarding is shown by a dashed line with a filled arrowhead that goes from the original entry that does the forwarding to subsequent forwarded-to entries. Figure 3-5 shows UCM and LQN models for a forwarding interaction.

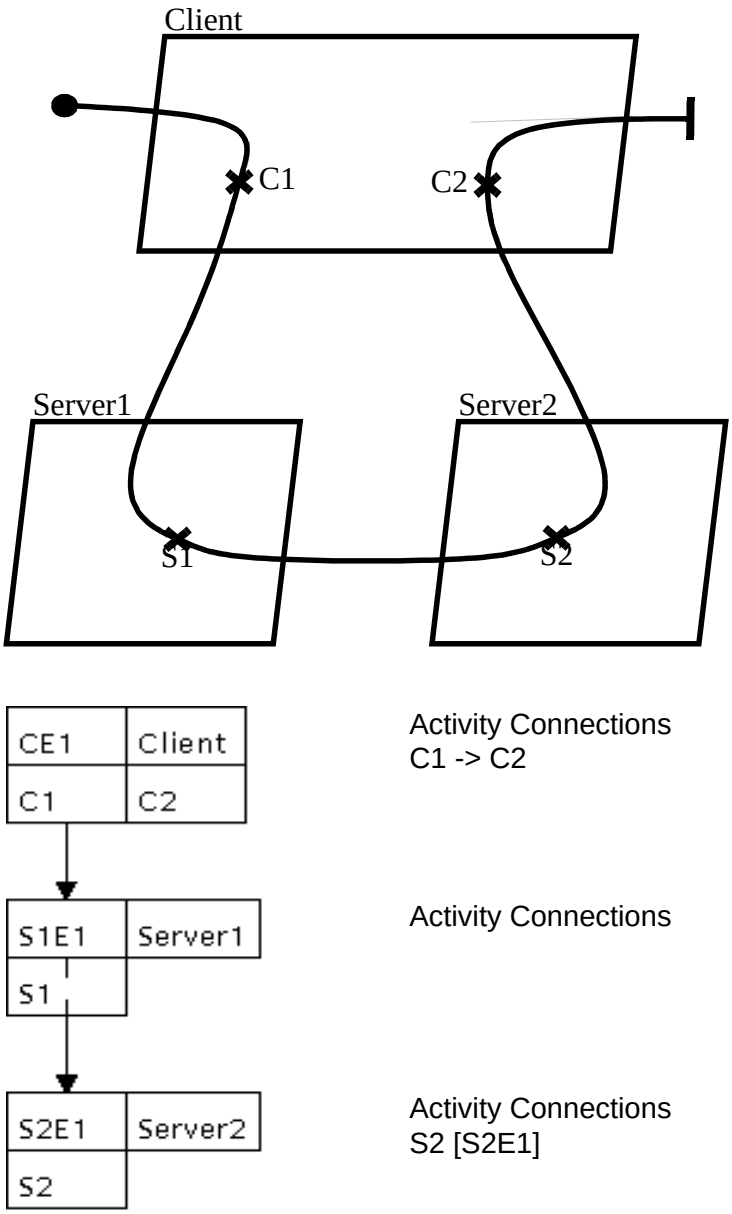


Figure 3-5: Corresponding UCM and LQN models for a forwarded synchronous call and subsequent return.

3.2.4. Parallel Calls

The UCM path has an AND fork and then join in the calling component. By making calls from each branch after the fork, parallel services are requested in the other components. In the

LQN model the AND is indicated by an ‘&’ between activities in the activity connection text boxes. Figure 3-6 shows the corresponding UCM and LQN models for such an instance.

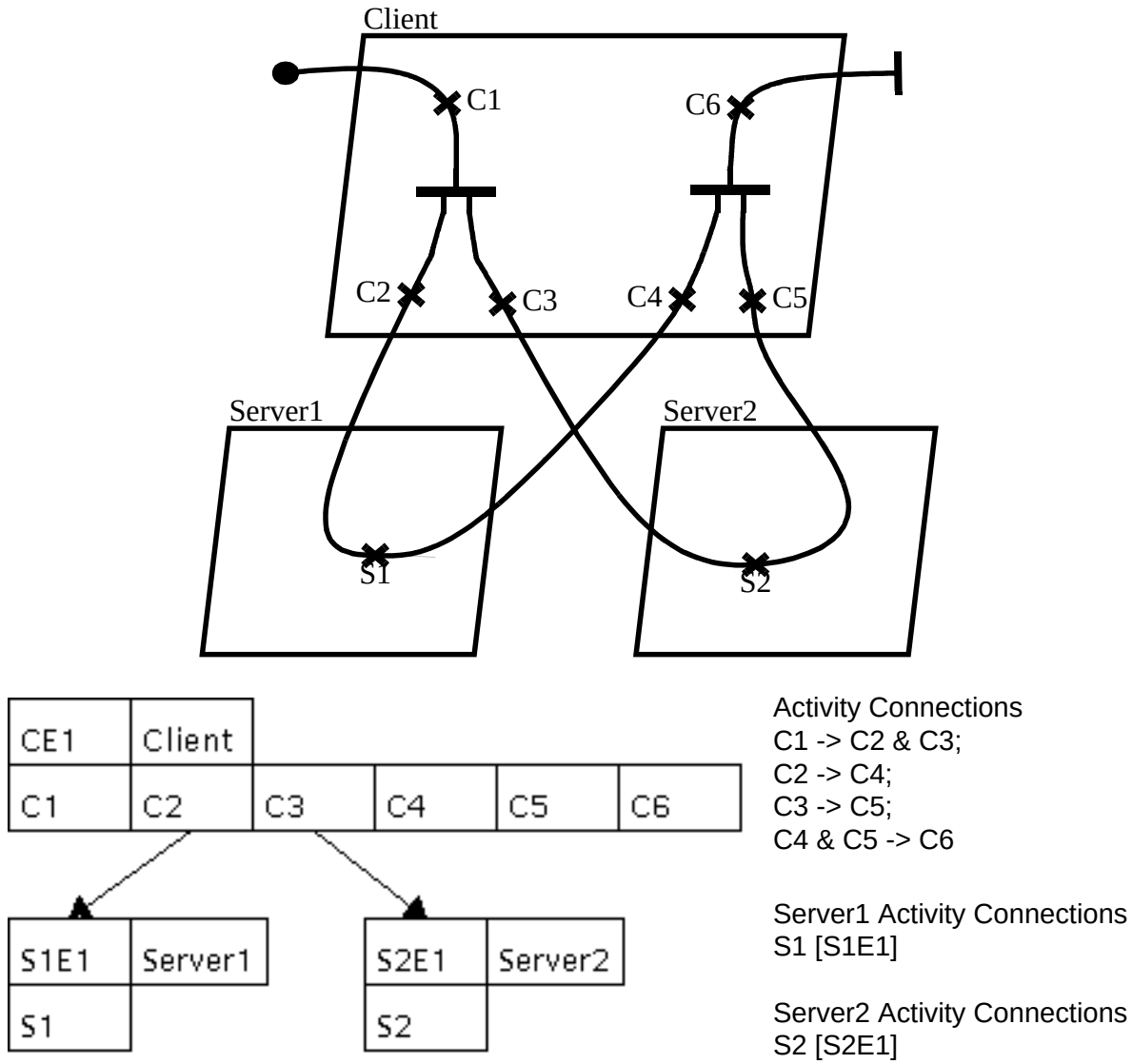


Figure 3-6: Corresponding UCM and LQN models for parallel synchronous calls and returns.

3.2.5. Alternative Calls

Similarly to the parallel case above, the UCM path has an OR fork and join in the calling component. By making calls from each branch after the fork, competing alternate services are

requested in the other components. In the LQN model the AND is indicated by a '+' between activities in the activity connection text boxes. Figure 3-7 shows the corresponding UCM and LQN models for this case.

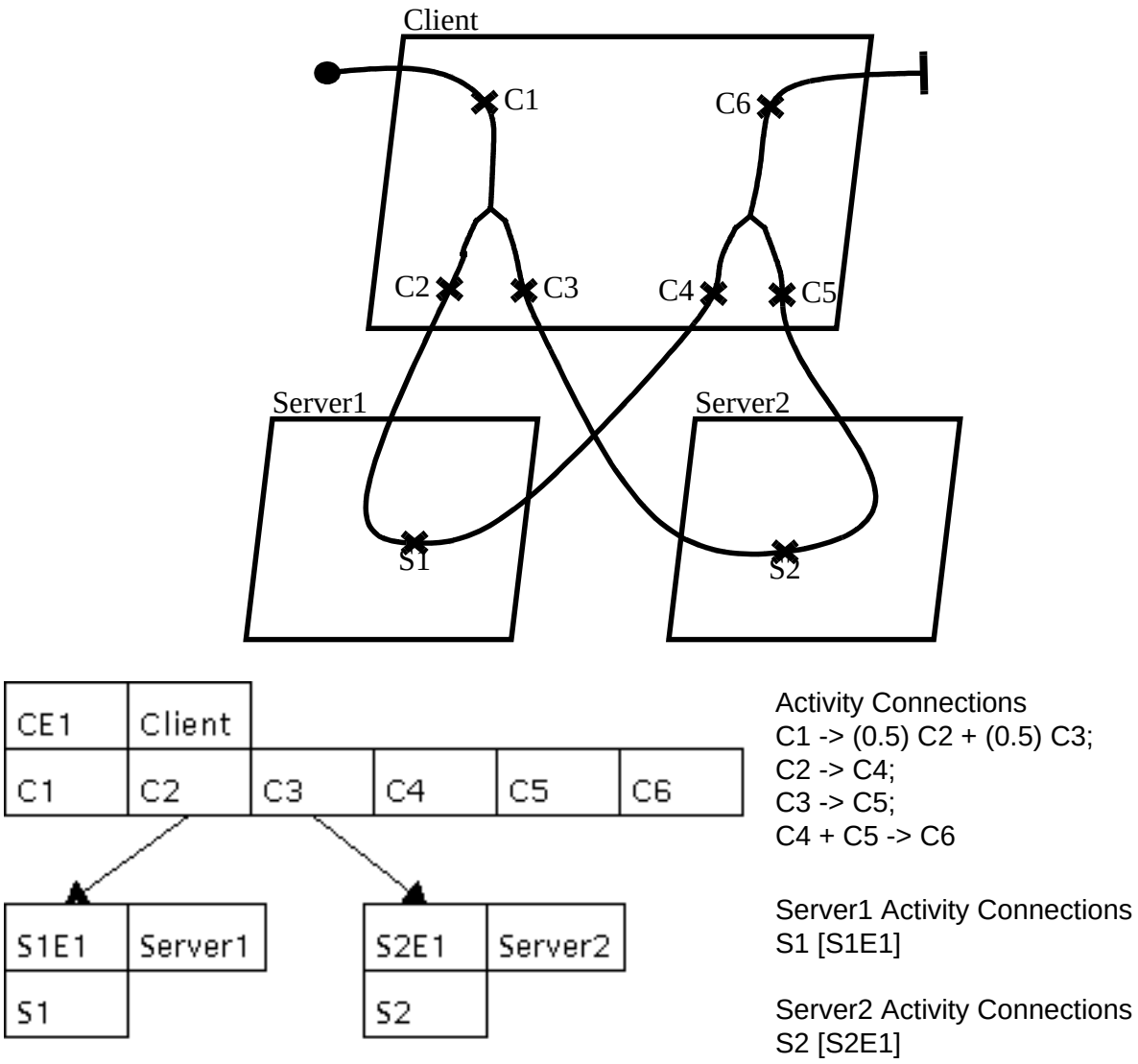


Figure 3-7: Corresponding UCM and LQN models for alternative synchronous calls and returns.

3.2.6. Looping

A loop is indicated by a special UCM loop construct that appears the same as an OR join followed immediately by an OR fork. In the LQN model the loop is indicated by a loop traversal

count multiplying the loop activity ID in the activity connection text boxes. Figure 3-8 shows the corresponding UCM and LQN models for a synchronous interaction with a loop in the server.

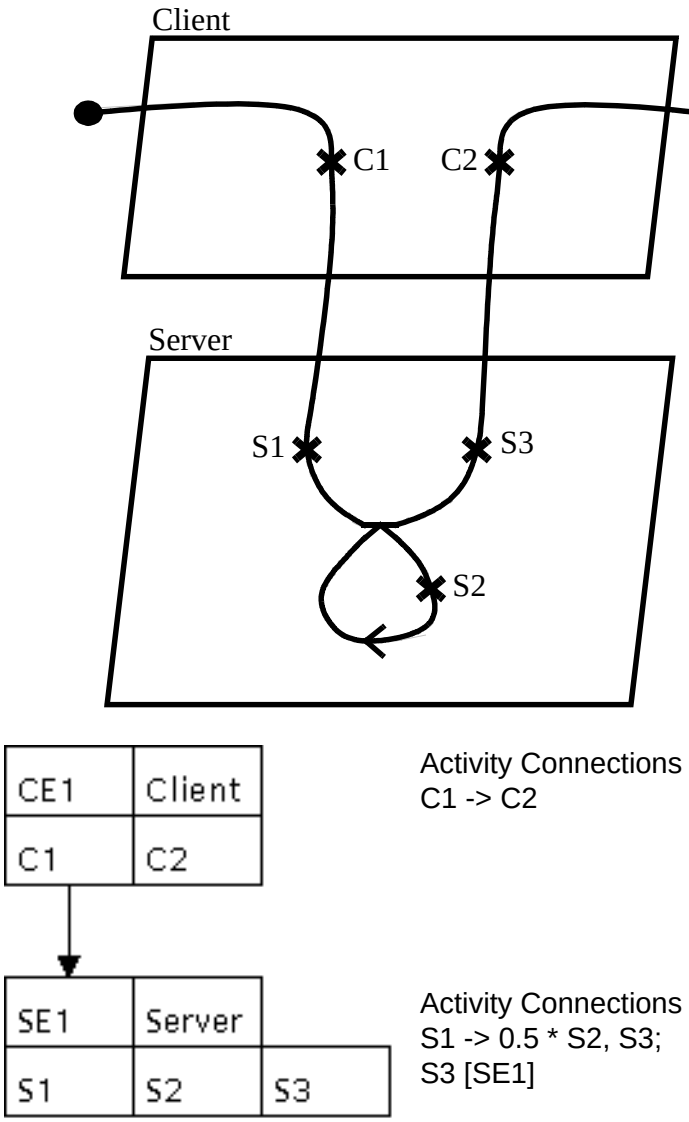


Figure 3-8: Corresponding UCM and LQN models for a loop.

3.3. Complex Patterns of Interaction

There are possible patterns of interaction that can be expressed as UCMs but do not have a straightforward corresponding LQN representation. This section examines two such patterns.

3.3.1. Fork and Join in Separate Components

It is common to have UCM models that represent systems where paths fork in one component and join in another, such as the example shown in Figure 3-9.

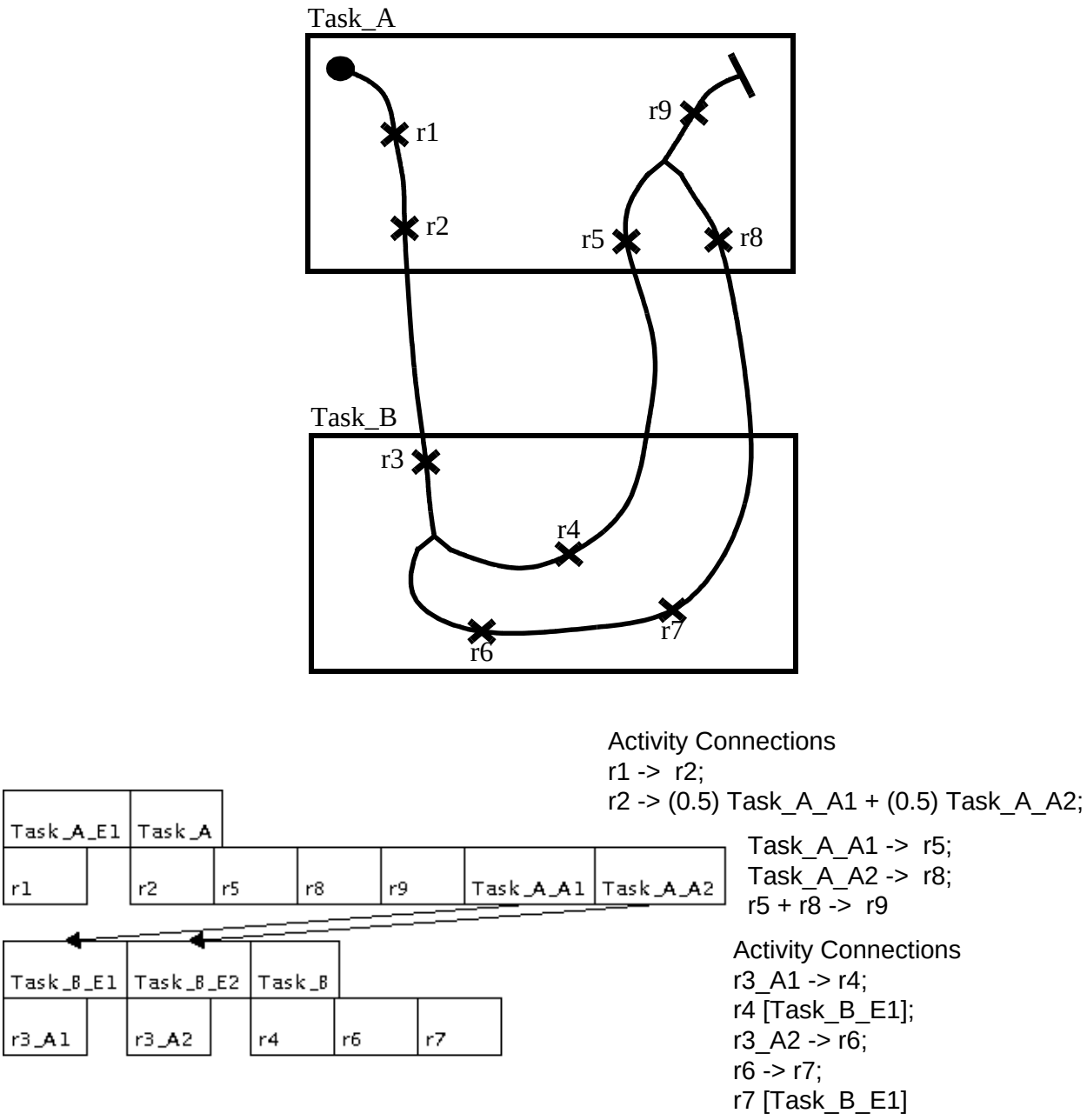


Figure 3-9: UCM and LQN models including an OR fork and join in separate tasks.

While such systems can also be represented using the LQN notation, and they are syntactically correct, semantically they are doubtful at best. The LQNS solver is unable to process the semantics of such a model since it does not break down into a tidy analytical solution. The ParaSRVN simulator can solve models with corresponding forks and joins in different tasks, but only if the activities that send messages are defined to send exactly one message and their workload is deterministic. Such solving restrictions make the use LQN models with forks and corresponding joins in different tasks undesirable. A better solution is to create equivalent LQN models which do not have distributed forks and joins.

The example system shown in Figure 3-9 has a client *Task_A* making a synchronous service request at the server *Task_B*. *Task_B* executes responsibility r3 and then splits into two alternative streams of execution, one which executes responsibility r4 and then sends a reply, or the other one which executes responsibilities r6 and r7 before replying. *Task_A* executes responsibility r4 after receiving the reply from the first alternative stream and responsibility r8 after receiving the reply from the second parallel stream. After either responsibility r4 or r8 have been executed, *Task_A* resumes a single stream of execution. The equivalent LQN model removes the OR fork from *Task_B* and places it in *Task_A*. *Task_B* has two fully independent execution paths and responsibility r3 is duplicated as two identical activities, r3_A1 and r3_A2. The resulting LQN model can now be solved using both LQNS and ParaSRVN without any restrictions on workload specification.

Please note that the same strategy of only results in an approximate, not a fully equivalent, LQN model if the original UCM model has an AND instead of an OR fork and join. In such a case, Task_B would end up executing both r3_A1 and r3_A2, instead of either r3_A1 or r3_A2 in this case.

3.3.2. Loop with Complex Body

LQNs can easily represent loops with a single activity as their body, as described in Section 3.2.6. These loops correspond to repeated activities in the LQN. Representing models with more complex loop bodies can be a problem however, since there is no provision in the LQN notation to repeat sequential blocks of activities. This problem can be overcome by abstracting the loop control activity away from the loop body. Figure 3-10 shows an example system with is a

loop with multiple activities in the loop body.

The UCM in Figure 3-10 shows a client Task_A making a synchronous service request at the server Task_B. Before replying, Task_B must execute responsibility r3, loop twice through the sequence of responsibilities r4, r5, r6 and r7, and then execute responsibility r8 before replying. In order to model the system as an LQN, it was necessary to abstract the loop head from the loop body. The resulting LQN model includes a clone of Task_B named Task_B_clone. This clone task is identical to Task_B in every respect and handles the activities associated with the loop body. The loop is modeled by repeating a loop control activity Task_B_LH1, which in turns makes a synchronous call to entry Task_B_clone_E1 in Task_B_clone. Entry Task_B_clone_E1 is executes the sequence of activities r4, r5, r6, and r7 before replying back to Task_B_LH1.

The resulting LQN model can thus be made to correspond to the same system as in the original UCM, despite the limitations of the LQN notation when it comes to describing repeated blocks of activities.

3.4. Performance Information in UCMs

In order to create meaningful LQN models it is necessary that some performance information be included in the UCM specification. Table 3-3 shows UCM constructs, their associated performance data that is required, whether the UCMNav has facilities to enter the data, and the default values used in the LQN generation if the data is not specified. In order to create meaningful LQN models it is necessary that some performance information be included in the UCM specification. Table 3-3 shows UCM constructs, their associated performance data that is required, whether the UCMNav has facilities to enter the data, and the default values used in the LQN generation if the data is not specified.

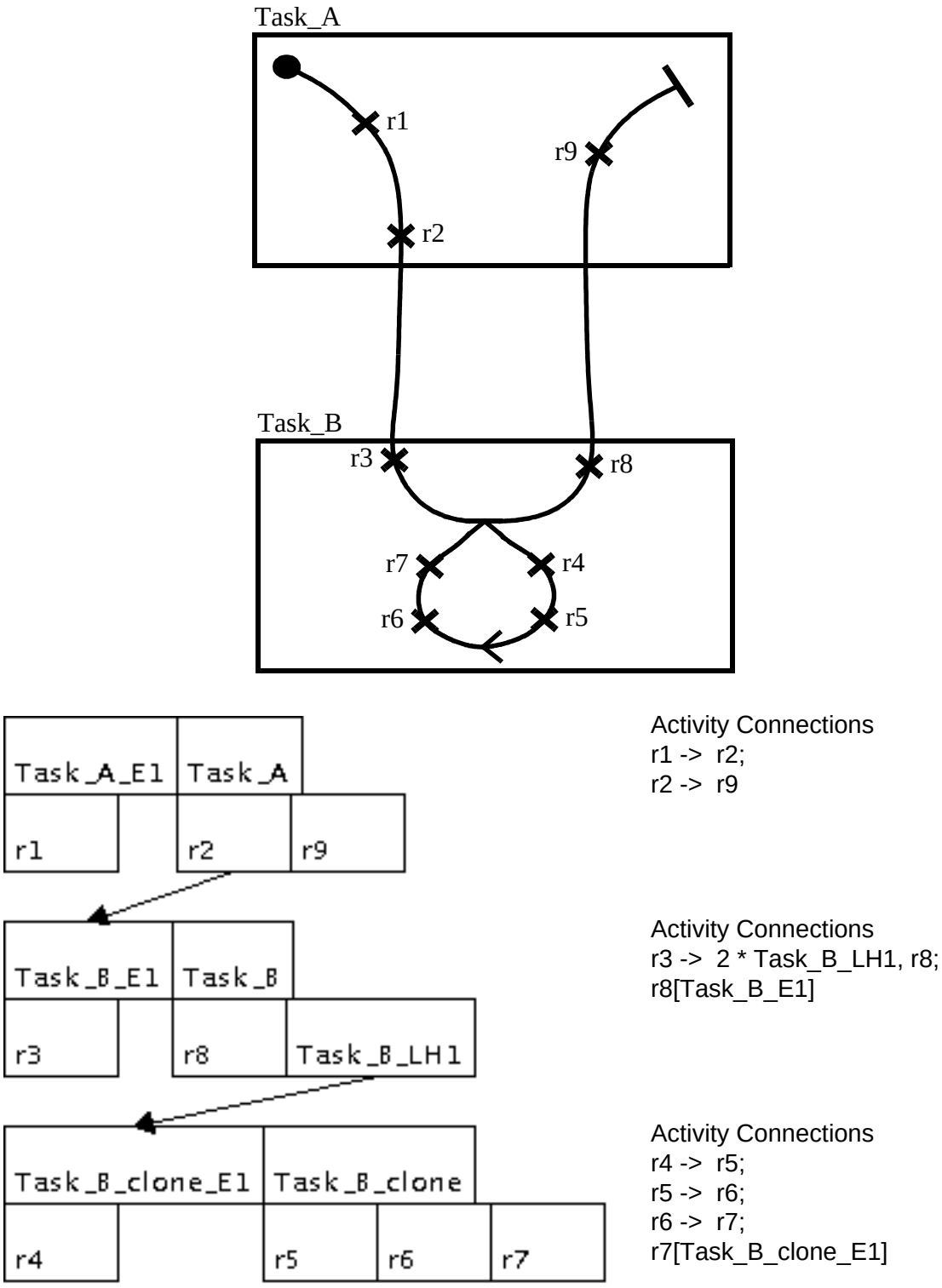


Figure 3-10: UCM and LQN models including a complex loop.

UCM Construct	Performance Data Required	UCMNav Support	Default Values
responsibility	number of calls	yes	1.0 calls
component	associated devices	yes	infinite processor
device	speed-up factor	yes	1.0
OR fork	probability of taking each branch	yes	equal probability for each branch
loop	number of loop iterations	no	3.0 iterations
start point	open system arrival rate and distribution	yes	deterministic distribution, 1.0 arrival rate
	closed system population	no	10

Table 3-3: UCM constructs, the necessary performance data needed to create meaningful LQNs, UCMNav support for entering the data, and default values used if the data is not specified.

Chapter 4 - Transformation Strategy

The UCM2LQN tool transforms UCM models from the UCMNav into LQN models that can be input into the LQNS tool. This chapter describes the strategy behind some of the design decisions that were made, as well as the internal data and class structure of the UCMNav and UCM2LQN converter.

4.1. UCM2LQN Design Choices

A previous attempt at creating a UCM-to-LQN conversion tool was made by Greg Franks as part of his Ph.D. research. Unfortunately the resulting program did not work as well as hoped for, but his work did provide a suitable staring point for the research effort described in this thesis.

Greg Franks' UCM-to-LQN tool used the XML files saved by the UCMNav as its input. This strategy had the seeming advantage of fully decoupling the conversion from the UCMNav and allowing to use only the required UCM path and component information from the file and dispense with the memory requirements of creating objects that are used by the UCMNav but are unnecessary for conversion. It also meant that the conversion tool could be completely decoupled from the UCMNav. This did require Franks to create a new XML loading filter to read in the UCM file. Unfortunately, the XML document-type definition (DTD) for the UCM file format needs to be modified as the UCMNav is refined and additional features are added, and such modifications of the DTD did indeed take place. This meant that the conversion tool was soon unable to read in the latest UCMNav file versions and as such became obsolete.

Reading in the XML file output from the UCMNav also has the additional shortcoming of misinterpreting the proper sequence of points along a path. Each point along a UCM path is saved with an identifier number that is generated when it is instantiated. Franks interpreted this number as indicating the point's position along a UCM path. Unfortunately the identifier number is only an indication of when a given point was created and bears no relationship to its position or sequence along a path. This shortcoming in interpreting the XML file can be overcome by creating UCM paths in strict sequence and never adding any new points after a path has been created, but this is an unenforceable restriction if the tool is to be more widely distributed and makes the

creation of complicated UCMs too inconvenient.

This first attempt at a UCM-to-LQN conversion tool showed that the only reliable and practical way to convert UCMs is to start with the internal UCM model used in the UCMNav instead of reading in the UCMNav file output because the UCMNav XML DTD will evolve as the UCMNav is refined and the UCMNav classes provide methods for following a path that would need to be reinvented if the XML is to be parsed directly. This means that the UCM2LQN conversion tool must communicate directly with an instance of the UCMNav that has the desired UCM and can pass on its internal model. Given this restriction, it was decided that UCM2LQN might as well be fully integrated with the UCMNav as an optional add-on, thus making it more transparent and convenient to the user.

The danger with this approach is in having the add-on become too closely coupled with the UCMNav code. In order to prevent this, a convention was adopted whereby hooks into the UCMNav code are provided, but all the add-on code is kept in separate files and no other changes are made to UCMNav code. Normally, add-on files are not compiled into the UCMNav, but if the add-on is needed its code can be included by defining a special compilation flag in the makefile. This convention allows for the concurrent development of both the UCMNav and any add-on, such as UCM2LQN, in a manner which does not lead to the creation of separate code variants for either tool. It also means that multiple add-ons can be included into the UCMNav and the mix of those can be tailored to suit any preference simply by declaring the appropriate flags at compile time.

Both the UCMNav and the UCM2LQN classes make use of a *Cltn* class to manage sets of multiple pointers or instances of other classes. *Cltn* is a template class that provides methods to treat sets of identical objects in either an ordered or unordered manner. Furthermore, the *Cltn* class dynamically allocates and deallocates memory and as such can be used to manage sets of arbitrary and variable size. Any references to multiple objects in this chapter assume that those objects are organized in *Cltn* collections.

4.2. UCMNav

This section describes the design of the UCMNav, its internal hypergraph model, and the

inheritance and containment relationships of the UCMNav classes that were used as part of the input of the converter.

4.2.1. Design

The UCMNav has two main functions: managing all the logical objects that make up a UCM model and providing a visual interface that displays the model and makes it possible to edit it. As such, the UCMNav classes can be divided into two major categories: logical classes and display classes. The logical classes store all the data associated with the model, while the display classes provide the user interface to access this data.

The UCMNav display is managed by the *DisplayManager* class. The *DisplayManager* controls all the UCM entities that have a visual representation. The *DeviceDirectory* class keeps track of all the devices in the UCM model. When the UCM2LQN converter is invoked, it is passed a pointer to the complete set of UCM maps from the *DisplayManager* and a pointer to the list of devices from *DeviceDirectory.*

There are three main kinds of logical entities that we're concerned with in order to generate LQNs: path elements, components, and devices. Of these three, the path elements and components also have a corresponding UCM visual notation and hence corresponding display classes, while the devices do not have any such corresponding visual notation nor any corresponding display classes. The relationship between path elements is stored as a hypergraph model which is explained in the next section. The UCMNav *Map* class passed as an input to the UCM2LQN converter contains the hypergraph describing its elements as well as the list of components included in the map.

4.2.2. The Hypergraph Model

The UCMNav uses a hypergraph to represent the connections between UCM path elements in a path. The decision to use a hypergraph was made much earlier in the development of the UCMNav, well before the start of this research project. A hypergraph is a graph whose hyperedges connect two or more vertices.

A hypergraph can be thought of as a directed graph-in-reverse. It is composed of edges,

called hyperedges, and vertices, or nodes. A hyperedge connects a set of multiple source nodes with a set of multiple target nodes. A node has a single hyperedge leading into it and a single hyperedge leading from it. This contrasts with a traditional graph where the edges are arcs that connect a single node to another single node, and the nodes are hubs that can have multiple edges leading into and from them.

The hypergraph supports the expected kind of operations on its elements. Hyperedges and nodes can be added to, inserted in, shifted around, and removed from the graph. Since the hypergraph is directed, both hyperedges and nodes support direction by distinguishing between source inputs and target outputs. Although it is possible to create infinite looping structures in the hypergraph, as it is used in the UCMNav there is a requirement that there be at least one start point and an ultimate end point. There is no restriction on how many start or end points there are as long as there is at least one of each. Since a hyperedge can have multiple source and target nodes, there is no need for another type of construct to show forks, joins, or loop-heads. Thus the hyperedge is the only construct needed for a straight connection, a fork, a join, a join-then-fork construct, or a loophead.

The hyperedges in the UCMNav hypergraph thus correspond to points along a UCM path and the nodes correspond to the arcs between those points. All the UCM path constructs with a semantic meaning - such as start and end points, responsibilities, forks and joins, loop heads - are points along a path and as such correspond to hyperedges in the hypergraph. The arcs along a path do not carry special semantic meaning, and as such neither do the nodes in the hypergraph.

Components are not directly part of a path in the UCM notation and thus they are not part of the hypergraph proper. They are represented in the UCMNav internal model by component objects that have a containment relationship with hypergraph elements. The task of generating LQN models from the UCMNav involves dealing almost exclusively with the logical classes, except when it comes to this containment of path elements in components. Technically, the containment status of points along a UCM path is solely a factor of how the component and path figures are drawn and displayed on the screen. As such the UCMNav *Component* class, which defines the logical component objects, does not provide any methods to directly access the hyperedges corresponding to the path elements it may contain. Therefore it is necessary to refer to the *HyperedgeFigure* and *ComponentReference* classes, which handle the display of the *Hyperedge*

and *Component* classes respectively, in order to be able to determine whether a given hyperedge is contained in a component (or vice-versa).

4.2.3. Hypergraph Classes

The hypergraph classes are divided into two general types. The first type are the logical entity classes which represent the UCM constructs, their interconnections, and associated data. The second type are the associated figure classes which deal with the screen placement of those logical objects. This section describes the inheritance hierarchy and the class containment relationships between those classes.

4.2.3.1. Class Inheritance Hierarchy

The base hypergraph class is the *Hyperedge.* It is a virtual class that defines all the methods and data common to every hyperedge. Every class with a single input and output is a direct child of *Hyperedge,* as well as the *Loop* class which has a fixed number of two inputs and two outputs (one of each for the main path and the loop body). The *MultipathEdge* virtual class refines *Hyperedge* with methods to manage a variable number of multiple input or output paths. The *OrFork*, *OrJoin,* and *Synchronization* classes are derived from *MultipathEdge* since they can have a variable number of input and/or output branches. Figure 4-1shows the class hierarchy for the hyperedge classes.

The base class for the display classes is the *Figure* class. It defines the basic methods of positioning and drawing the objects on the screen. The *HyperedgeFigure* class further refines *Figure* with pointers to the corresponding logical hyperedge and containing component reference. The other hyperedge display classes descend from *HyperedgeFigure,* with *PointFigure* handling the display of empty points, start points, end points, waiting places, and timers. All the classes descending from HyperedgeFigure deal solely with the display of the points associated with their logical hyperedge counterparts. The *LoopNullFigure, OrNullFigure,* and *SynchNullFigure* classes do not display hyperedges directly but rather are associated with the display of the branching structures from the *LoopFigure, OrFigure,* and *SynchronizationFigure* respectively. Figure 4-2 shows the inheritance hierarchy for the hyperedge display classes in the UCMNav.

Figure 4-3 shows which figure classes and logical classes correspond with each other. In

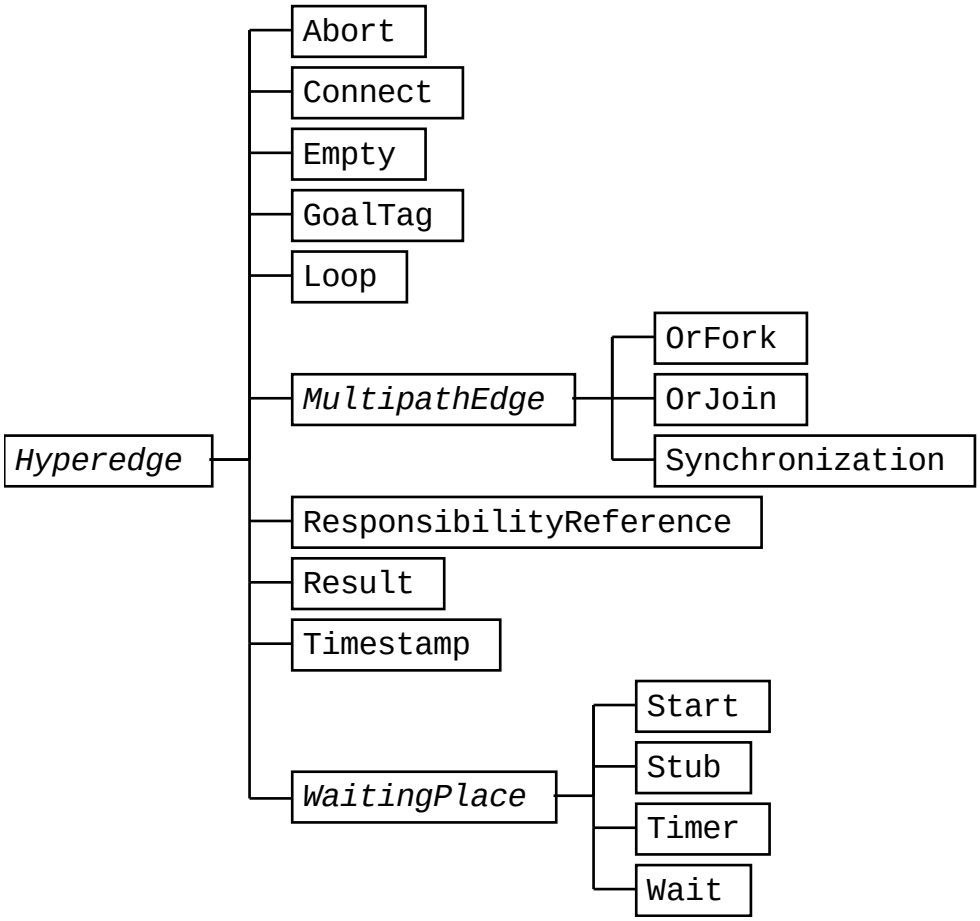


Figure 4-1: Inheritance hierarchy for the classes derived from *Hyperedge* (obtained using the WindRiver SNiFF+ code browser).

some cases the figure class and the logical class will have a direct relationship where one or both of the classes have a pointer to the other class (as is the case with the *HyperedgeFigure* and *Hyperedge* classes) or where either the figure or logical class includes its counterpart explicitly as a friend class. In other cases the relationship between the figure and logical classes is indirectly inherited from direct relationship between the parent *HyperedgeFigure* and *Hyperedge* classes.

4.2.3.2. Class Containment Relationships

The UCM2LQN converter takes as its input the active maps and devices from the UCM-Nav. Figure 4-4 shows a partial class containment diagram for the *Map* and *Device* classes.

A *Map* contains a *Hypergraph*, a collection of *ComponentReferences*, a collection of

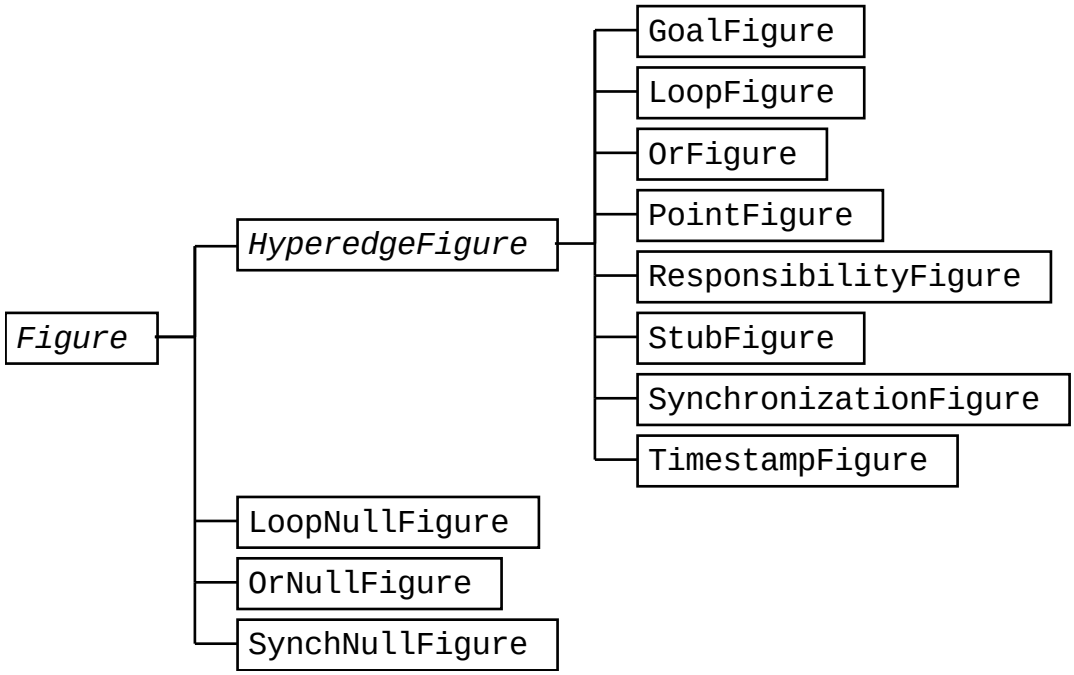


Figure 4-2: Inheritance hierarchy for the classes derived from *Figure* (obtained using the WindRiver SNiFF+ code browser).

Paths, a collection of *ResponsibilityReferences*, a collection of *HyperedgeFigures*, and a pointer to its parent *Stub* it is used as a plug-in anywhere. A *Hypergraph* contains a collection of logical *Hyperedges* and a collection of *Nodes*. *ComponentReferences* all contain a logical *Component*, which in turn contains an integer device id number that can be used to identify the processor it is running on. Each *ComponentReference* also has a collection of pointers to the *HyperedgeFigures* enclosed within its borders. A *Path* contains a collection of pointers to its logical *Hyperedge* elements. Every *ResponsibilityReference* contains a logical *Responsibility*, which in turn contains a collection of *ServiceRequests* each of which contains an integer device id number identifying which device is being requested. *Stubs* contain either a collection of *ServiceRequests* like *Responsibilities* or a collection of sub-*Maps* for their plug-ins. All *HyperdgeFigures* contain a pointer to their enclosing *ComponentReference* and a pointer to their corresponding logical *Hyperedge*. *Hyperedges* contain a pointer to their corresponding *HyperedgeFigure*, a collection of pointers to their input *Nodes*, and another collection of pointers to their output *Nodes*. *Nodes* contain a pointer to their respective input and output *Hyperedges*. Finally, each *Device* contains an integer identifier. Please note that *Devices* are only contained in the *DeviceDirectory* class (which is not

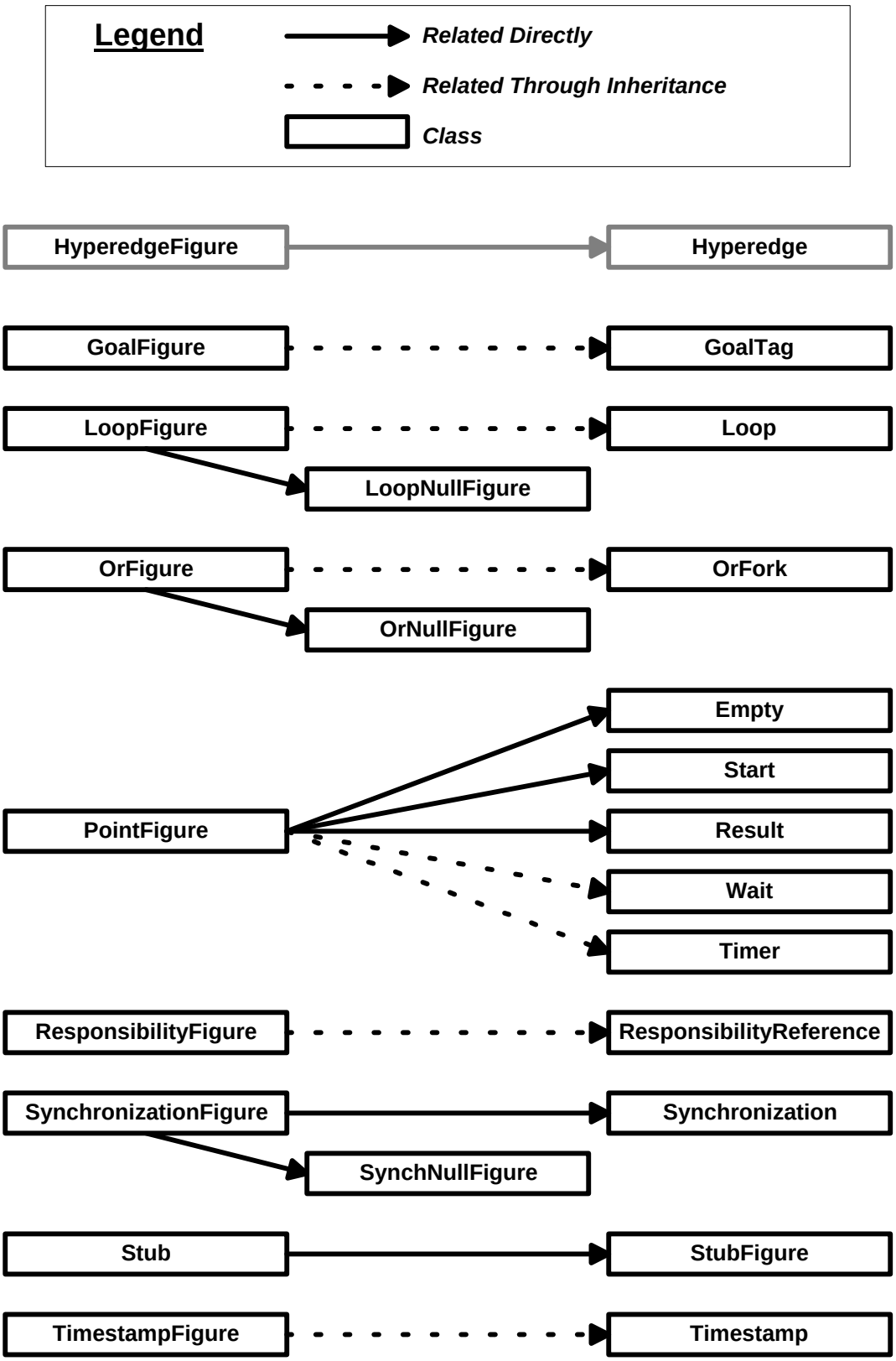


Figure 4-3: Correspondence relationships between the UCMNav hypergraph figure classes and logical classes

shown in Figure 4-4) and only their identifier is used by any other classes.

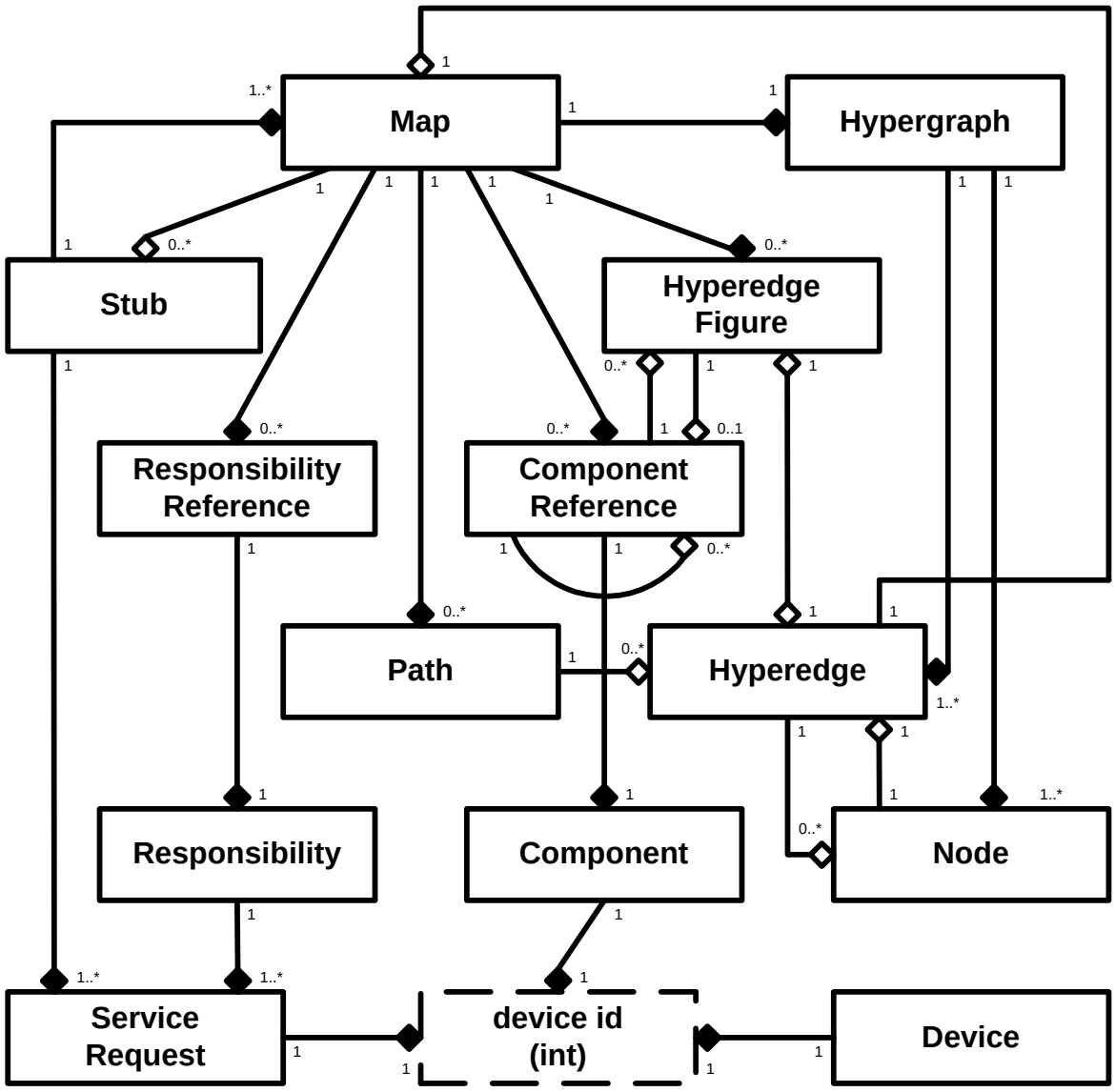


Figure 4-4: Partial class containment diagram for the UCMNav classes passed to the UCM2LQN converter.

4.3. UCM2LQN LQN Model

The UCM2LQN converter takes the set of maps and devices as an input when invoked from the UCMNav and generates a set of LQN objects representing the same model. Those

objects are then saved to file in the LQNS file format described in Section 2.2.3.3. This section describes the LQN classes that were created, along with their inheritance hierarchy and containment relationships.

4.3.1. LQN Classes

There are eight LQN classes as follows:

- *Ucm2Lqn* - wrapper class for the UCM2LQN converter, implements the UCM to LQN conversion algorithm described in Chapter 5
- *Lqn* - container class for the LQN elements
- *LqnActivity* - LQN element class, describes an activity
- *LqnEntry* - LQN element class, describes an entry
- *LqnTask* - LQN element class, describes a task
- *LqnDevice* - LQN element class, describes a device
- *LqnCrs* - Call and Reply Stack class, used to keep track of component boundary crossing when following UCM paths
- *LqnCrsElement* - Call and Reply Stack element class, wrapper for an *LqnActivity* or *LqnEntry*

The inheritance hierarchy for the UCM2LQN classes is shown in Figure 4-5. It is a flat hierarchy with each class being defined independently. None of the classes have shared features that would have made it worthwhile to put them together in related groups.

The Ucm2Lqn class is invoked when the “Create LQN” item is chosen from the Performance menu in the UCMNav. An *Lqn* object and a collection of *LqnCrs*’s are created along with the Ucm2Lqn object. All other objects are created dynamically as the UCM map is parsed. The *Ucm2Lqn* class has the following methods which may be of interest:

```
void Transmorgrify(Cltn<Map*>* maps, Cltn<Device*>* devices)
```

- called from the UCMNav to transform UCMs into LQNs
- checks for the completeness of the UCM and orchestrates its traversal

```
void Start2Lqn(Hyperedge* start_pt)
```

- transforms a UCM start point into the appropriate set of LQN constructs

```
void Edge2Lqn(Hyperedge* edge)
```

- transforms any other UCM hyperedge into the appropriate LQN construct(s)

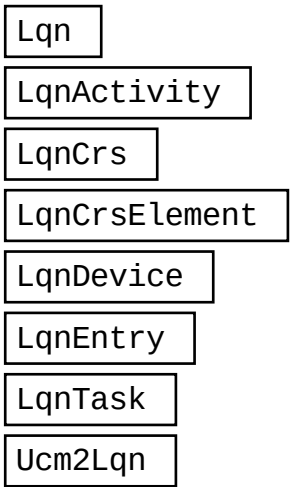


Figure 4-5: Inheritance hierarchy for the UCM2LQN LQN classes (obtained using the WindRiver SNiFF+ code browser).

xing_type Xing(Hyperedge* edge1, Hyperedge* edge2)

- determines if component boundaries have been crossed when going from edge1 to edge2
- if a boundary was crossed then also determines the type of crossing - entering a component, leaving a component, or moving directly from one component into another

The algorithm used to traverse the UCM and create the LQN constructs is described in Chapter 5.

4.3.2. Class Containment Relationships

Figure 4-6 shows the class containment diagram of the UCM2LQN classes. As the wrapper class, *Ucm2Lqn* contains an *Lqn* and a collection of *LqnCrs*'s. The *Lqn* object represents the entire LQN model for the UCM input, including sub-maps that are plugged into any stubs. As such there is no need to have more than one *Lqn* object.

The LQN elements - *LqnTask*, *LqnEntry*, *LqnActivity*, and *LqnDevice* - are contained in such a way as to make it easy to get the correct output file format. The*Lqn* class contains a collection of *LqnDevices* and a collection of *LqnTasks*. Each *LqnTask* points to the *LqnDevice* it runs on. *LqnTask* also includes a collection of *LqnEntries* and a collection of *LqnActivities*. The *LqnEntry* class has a pointer to its parent *LqnTask*, a pointer to its first *LqnActivity*, and a pointer to the *LqnActivity* that called it. The LqnActivity class has a a pointer to its parent *LqnTask*, a

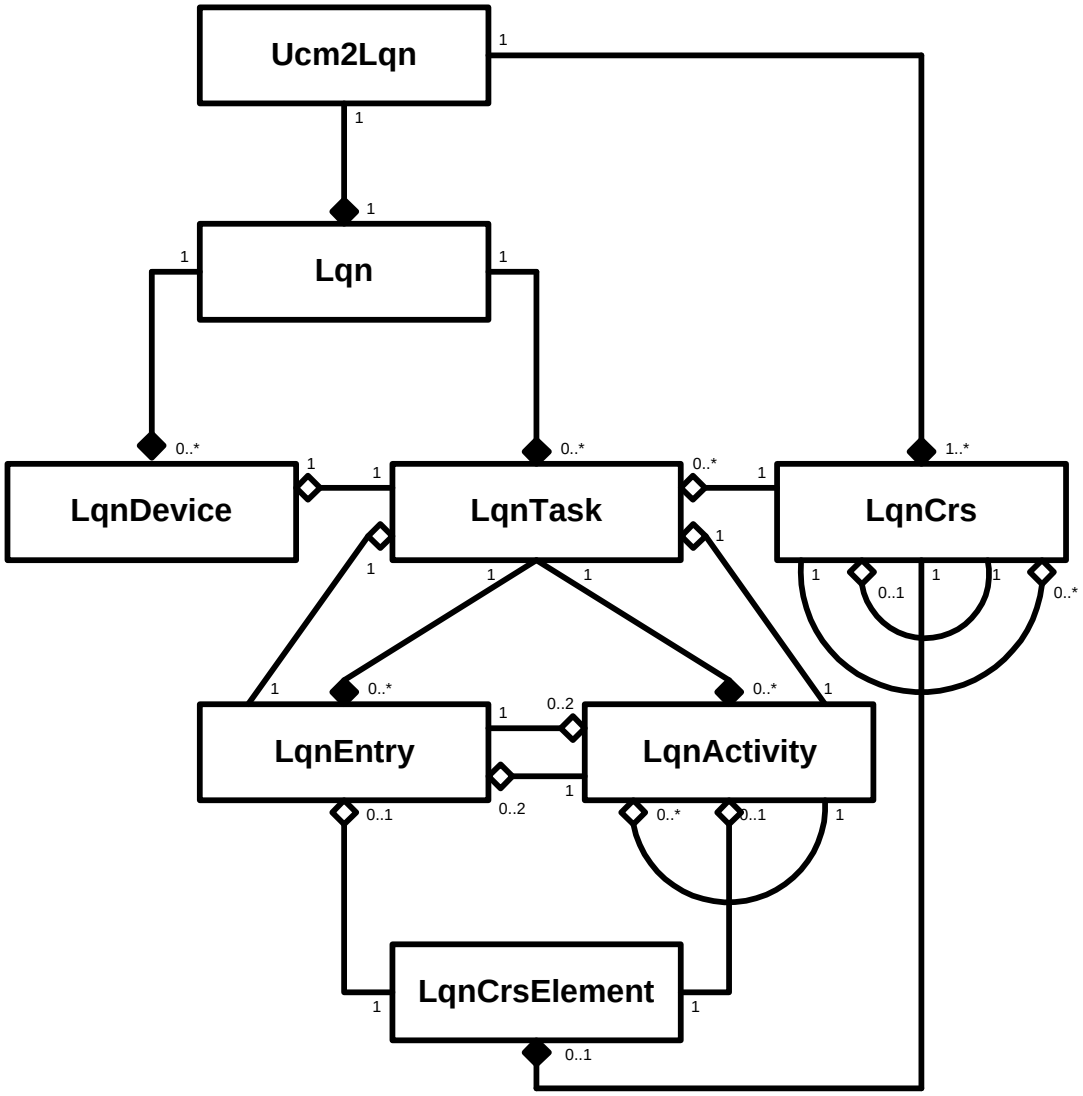


Figure 4-6: Class containment diagram for the UCM2LQN classes.

pointer to the *LqnEntry* through which it began execution, a pointer to any *LqnEntry* it may call on, a collection of pointers to any possible preceding *LqnActivities*, and a collection of pointers to any succeeding *LqnActivities*.

The *LqnCrs* class contains a collection of *LqnCrsElements*, a pointer to the preceding *LqnCrs*, and a collection of pointers to any succeeding *LqnCrs*'s. The *LqnCrsElement* class has a pointer to the *LqnEntry* and a pointer to the *LqnActivity* it may represent.

4.3.3. LQN File Output

The LQN model is output to the file by taking advantage of the containment hierarchy of the LQN objects. Once the LQN model has been created, the *Ucm2Lqn* object opens a "ucm2lqn.lqn" file to write the output to. *Ucm2Lqn* then invokes the FilePrint method for the *Lqn* object and passes it a file pointer to the output file. The *Lqn* prints the general system simulation parameters, the device information, and the task information to the file. The *Lqn* then invokes the FilePrint method for each *LqnTask* in order to output the specific task information. Each LqnTask thus saves its LqnEntry information by invoking the entries' FilePrint method, the LqnActivity information by invoking the activities' FilePrint method, and finally the LqnActivity connections by invoking each activity's FilePrintConnections method. Once all the information has been saved, the *Ucm2Lqn* object closes the "ucm2lqn.lqn" file.

Chapter 5 - UCM2LQN Algorithm

This chapter describes the algorithm that was created to generate an LQN model from a UCM. The algorithm is based upon a hyperedge to hyperedge traversal of the internal UCMNav data model as follows:

- if needed, create an LQN object corresponding to the current hyperedge
- get the next hyperedge on the path
- identify any component boundary crossings
  - determine he calling relationship between components
  - update the Call and Reply Stack (CRS)
- set the next hyperedge as the current hyperedge

If forks are encountered along the path, then outgoing branches are followed one by one until either an end point or a join is reached. If joins are encountered along the path, the traversal proceeds past the joins only after all the incoming branches have been traversed.

The algorithm details are presented in the following sections:

- *accessing the hyperedges sequentially Section 5.1.* - deals with getting the set of next or previous hyperedges for a given hyperedge
- *handling the next hyperedge (Section 5.2.)* - describes the major steps necessary after the next hyperedge on the path has been found
- *identifying component boundary crossings (Section 5.3.)* - identifies whether a component boundary crossing has occurred, and if that is the case then what type of crossing was it
- *determining the calling relationships between the components (Section 5.4.)* - deals with the creation of LQN objects necessary to model the calling relationships between components
- *creating the LQN objects corresponding to the UCM path elements and directing the path traversal (Section 5.5.*) - deals with the creation of LQN objects corresponding to UCM elements along the path, and steers the path traversal based on the type of the UCM element
- *the Call and Reply Stack (Section 5.6.*) - used to keep track of unresolved calls during the path traversal

5.1. Accessing Hyperedges Sequentially

The basic logical UCM path element is the hyperedge, as described in Section 4.2.2. and Section 4.2.3. This section describes the logic necessary to access the next or the previous hyperedges along the path for a given hyperedge.

5.1.1. Getting the Next Hyperedges

There may be multiple next hyperedges for a given hyperedge, so a collection of next hyperedges is always used, even when there is only a single such edge. A collection of next hyperedges *next_edges* for a given hyperedge *edge* is obtained as follows:

```
1. next_edges = new collection of hyperedges
2. get the collection of target_nodes for the given edge
3. for each successive node in target_nodes, starting with the first
   node, until target_nodes is done
   3.1. next_edge = target hyperedge of node
   3.2. add next_edge to next_edges
4. return next_edges
```

5.1.2. Getting the Previous Hyperedges

Similarly, there may be multiple previous hyperedges for a given hyperedge, so a collection of previous hyperedges is always used, even when there is only a single such edge. A collection of previous hyperedges *previous_edges* for a given hyperedge *edge* is obtained as follows:

```
1. previous_edges = new collection of hyperedges
2. get the collection of source_nodes for the given edge
3. for each successive node in source_nodes, starting with the first
   node, until source_nodes is done
   3.1. previous_edge = source hyperedge of node
   3.2. add next_edge to previous_edges
4. return previous_edges
```

5.2. Handling The Next Hyperedge

This section describes the steps needed after the next hyperedge is retrieved. Whenever a new hyperedge is encountered, the following actions are taken:

```
1. find component boundary crossings (see Section 5.3.)
2. switch component boundary crossing
   2.1. case leaving component
```

2.1.1. handle leaving component (see Section 5.4.1.)
2.1.2. **break**
2.2. **case** entering component
2.2.1. handle entering component (see Section 5.4.2.)
2.2.2. **break**
2.3. **case** changing components
2.3.1. handle leaving component (see Section 5.4.1.)
2.3.2. handle entering component (see Section 5.4.2.)
2.3.3. **break**
3. create LQN object (see Section 5.5.)

5.3. Identifying Component Boundary Crossings

This section deals with the determination of the type of communication between components that needs to be captured in the LQN model. The first step in determining what kind of communication occurs between components is to identify component boundary crossings. Thus in addition to traversing the UCM path, it is necessary to identify if the path crosses any component boundaries and the direction of the crossing (entering or leaving the component) when going from one hyperedge to another. The algorithm section to detect and identify the type of component boundary crossings when going from an *edge1* to an *edge2* is as follows:

1. get enclosing component *comp1* for *edge1*
2. get enclosing component *comp2* for *edge2*
3. **if** *comp1* does not exist and *comp2* does not exist **then**
   3.1. neither edge is in a component, therefore the path is not crossing any component boundaries
4. **else if** *comp1* exists and *comp2* does not exist **then**
   4.1. edge1 is in a component and edge2 is not, therefore the path is leaving a component
5. **else if** *comp1* does not exist and *comp2* exists **then**
   5.1. edge1 is not in a component and edge2 is, therefore the path is entering a component
6. **else**
   6.1. both edges are in a component
   6.2. **if** *comp1* is the same as *comp2* **then**
      6.2.1. *edge1* and *edge2* are in the same component, therefore the path is not crossing any component boundaries
   6.3. **else**
      6.3.1. *edge1* and *edge2* are in different components, therefore the path is changing components (leaving and then entering)

5.4. Handling Component Boundary Crossings

This section describes the algorithm details related to the creation of LQN objects neces-

sary to model the calling relationships between components. There are three possible types of component boundary crossings: leaving a component, entering a component, and changing components. This section shows the algorithms for handling leaving a component and entering a component. Changing components is simply a matter of leaving the first component and immediately entering the next component.

The CRS is used to keep track of unresolved messages along the path. Activities sending unresolved messages and entries receiving unresolved messages are pushed on the CRS as they are encountered. Whenever a message is resolved as being either synchronous or asynchronous, the associated activity and entry are popped off the CRS.

5.4.1. Handling Leaving a Component

The following algorithm fragment applies when leaving a component. Leaving a component always corresponds with sending a message. The default activity that is created corresponds to sending that message, although whether that message is a call or a reply can only be determined when the path enters another component. It is assumed that the hyperedge involved is the last edge encountered still inside the component being left.

```
1. get the enclosing component from the edge
2. get the LQN task corresponding to the enclosing component
3. if currently processing a loop body then
   3.1. task = clone of task for the loop body
4. message_activity = new default activity in task
5. push message_activity on the CRS
```

5.4.2. Handling Entering a Component

The following algorithm fragment applies when entering a component. Entering a component always corresponds with receiving a message and this part of the algorithm is applied to determine whether that message is a call or a reply. It is assumed that the hyperedge involved is the first edge encountered inside the component being entered.

```
1. get the enclosing component from the next_edge
2. next_task = task corresponding to the enclosing component
3. if currently processing a loop body then
   3.1. next_task = clone of next_task for the loop body
// check if the message is from a task that has already been visited
4. if next_task can be found on the CRS then
```

4.1. the message is a reply
4.2. *previous_task* = task before the last task on the CRS
// check if the reply comes from the last task called
4.3. **if** *previous_task* == *next_task* **then**
4.3.1. the reply is a direct synchronous reply
4.3.2. *reply_activity* = pop item off the CRS
4.3.3. *reply_entry* = pop item off the CRS
4.3.4. update *reply_activity* as making a reply to *reply_entry*
4.3.5. *call_activity* = pop item off the CRS
4.3.6. update *call_activity* as making a synchronous call
4.3.7. *handle_reply_activity* = new default activity in *next_task*
4.3.8. connect *handle_reply_activity* as the next activity after *call_activity*
// the reply does not come from the last task called
4.4. **else**
4.4.1. the reply is a forwarded reply
4.4.2. *reply_activity* = pop item off the CRS
4.4.3. *reply_entry* = pop item off the CRS
4.4.4. update *reply_activity* as making a reply to *reply_entry*
4.4.5. *previous_task* = task before the last task on the CRS
// keep going until the last task called
4.4.6. **while** *previous_task* != *next_task*
4.4.6.1. *forward_entry* = *reply_entry*
4.4.6.2. *reply_activity* = pop item off the CRS
4.4.6.3. update *reply_activity* as making a synchronous call
4.4.6.4. *reply_entry* = pop item off the CRS
4.4.6.5. update *reply_activity* as making a reply to *reply_entry*
4.4.6.6. update *reply_entry* as forwarding its call to *forward_entry*
4.4.6.7. *previous_task* = task before the last task on the CRS
4.4.7. *forward_entry* = *reply_entry*
4.4.8. *reply_activity* = pop item off the CRS
4.4.9. update *reply_activity* as making a synchronous call
4.4.10. *reply_entry* = pop item off the CRS
4.4.11. update *reply_activity* as making a reply to *reply_entry*
4.4.12. update *reply_entry* as forwarding its call to *forward_entry*
4.4.13. *call_activity* = pop item off the CRS
4.4.14. update *call_activity* as making a synchronous call
4.4.15. *handle_reply_activity* = new default activity in *next_task*
4.4.16. connect *handle_reply_activity* as the next activity after *call_activity*
// the message is not from a task that has already been visited
5. **else**
5.1. the message being received is a call
5.2. *next_entry* = new entry for *next_task*
5.3. *call_activity* = pop item off the CRS
5.4. update *call_activity* as making a call to *next_entry*
5.5. push *next_entry* on the CRS
5.6. *next_activity* = new default activity in *next_task*
5.7. connect *next_activity* as the first activity of *next_entry*

5.5. LQN Object Creation Algorithm

This section presents the part of the algorithm that deals with the creation of LQN objects that correspond to UCM path elements, as well as any special constraints these elements might place on the path traversal. This is the most complex part of the UCM2LQN algorithm since it must not only deal with the creation of the LQN objects, but also set up the interconnections between them, and implement the logic to choose which hyperedge to follow next. In the case of outgoing branches from forks, the path traversal is directed to follow each outgoing branch until either an end point or a join is encountered. When encountering joins, the path traversal is allowed to proceed past the join only after all the incoming branches have been traversed.

Reference tasks are assigned arrival rates and distributions as specified by the UCM start points. LQN activities are assigned workload demands as specified for the corresponding UCM responsibility. OR branches are assigned the probabilities set in the UCM. If any performance data is missing from the UCM, default values are assigned as noted in Table 3-3 in Section 3.4.

The algorithm for LQN object creation reads as follows:

1. get the hyperedge type of the *current_edge*
2. **switch** hyperedge type
   - 2.1. **case** *start_point*
     - 2.1.1. mark *start_point* as visited
     - // assign the *start_point* to a separate reference task running on
     - // an infinite processor as follows:
     - 2.1.2. *reference_task* = new reference task
     - 2.1.3. *reference_entry* = new entry for *reference_task*
     - 2.1.4. *reference_activity* = new default activity in*reference_task* for *start_point*
     - 2.1.5. connect *reference_activity* as the first activity of *reference_entry*
     - 2.1.6. push *reference_activity* on CRS
     - // if the start point was in a component then make a call to that
     - // component
     - 2.1.7. **if** *start_point* is contained in a component **then**
       - 2.1.7.1. *first_task* = task corresponding to the component
       - 2.1.7.2. *first_entry* = new entry for *first_task*
       - 2.1.7.3. update *reference_activity* as making a call to *first_entry*
       - 2.1.7.4. push *first_entry* on CRS
       - 2.1.7.5. *first_activity* = new default activity in *first_task*
       - 2.1.7.6. connect *first_activity* as the first activity of *first_entry*
     - 2.1.8. continue path traversal (see Section 5.2.)
     - 2.1.9. **break**
   - 2.2. **case** *responsibility*

2.2.1. **if** *responsibility* has not been visited **then**
// create and connect an activity, if necessary create a new task
// and entry for the activity
2.2.1.1. mark *responsibility* as visited
// check if the responsibility is in a component
2.2.1.2. **if** *responsibility* is contained in a component **then**
2.2.1.2.1. *responsibility_task* = task corresponding to the component
2.2.1.2.2. **if** currently processing a loop body **then**
2.2.1.2.2.1. *responsibility_task* = clone of *responsibility_task* for the loop body
2.2.1.2.3. *responsibility_activity* = new responsibility activity in *responsibility_task* for *responsibility*
2.2.1.2.4. **if** responsibility has service demands specified **then**
2.2.1.2.4.1. assign service demands to *responsibility_activity*
2.2.1.2.5. **else**
2.2.1.2.5.1. *responsibility_activity* has default service demands
2.2.1.2.6. connect *responsibility_activity* as the next activity after the last activity added to *responsibility_task*
2.2.1.2.7. continue path traversal (see Section 5.2.)
// the responsibility is not in a component
2.2.1.3. **else**
2.2.1.3.1. *responsibility_task* = new default task
2.2.1.3.2. *responsibility_entry* = new entry for *responsibility_task*
2.2.1.3.3. update the last activity on the CRS as making a call to *responsibility_entry*
2.2.1.3.4. push *responsibility_entry* on CRS
2.2.1.3.5. *responsibility_activity* = new responsibility activity in *responsibility_task* for *responsibility*
2.2.1.3.6. **if** responsibility has service demands specified **then**
2.2.1.3.6.1. assign service demands to *responsibility_activity*
2.2.1.3.7. **else**
2.2.1.3.7.1. *responsibility_activity* has default service demands
2.2.1.3.8. connect *responsibility_activity* as the first activity of *responsibility_entry*
2.2.1.3.9. *message_activity* = new default activity in *responsibility_task*
2.2.1.3.10. connect *message_activity* as the next activity after *responsibility_activity*
2.2.1.3.11. push *message_activity* on the CRS
2.2.1.3.12. continue path traversal (see Section 5.2.)
2.2.2. **break**
2.3. **case** *or_fork*
2.3.1. **if** *or_fork* has not been visited **then**
// create and connect fork activities, if necessary create a new
// task and entry for the activities
2.3.1.1. mark *or_fork* as visited
// check if the or fork is in a component
2.3.1.2. **if** *or_fork* is contained in a component **then**
2.3.1.2.1. *or_fork_task* = task corresponding to the component

2.3.1.2.2. **if** currently processing a loop body **then**
2.3.1.2.2.1. *or_fork_task* = clone of *or_fork_task* for the loop body
2.3.1.2.3. *or_fork_activity* = new default activity in *or_fork_task* for *or_fork*
2.3.1.2.4. connect *or_fork_activity* as the next activity after the last activity added to *or_fork_task*
2.3.1.2.5. get *next_edges* for *or_fork* (see Section 5.1.1.)
2.3.1.2.6. skip over the first item in *next_edges*
2.3.1.2.7. *fork_crs* = CRS
// deal with the secondary branches
2.3.1.2.8. **while** *next_edges* is not done
2.3.1.2.8.1. *branch_crs* = new CRS
2.3.1.2.8.2. set *branch_crs* as being on a branch path
2.3.1.2.8.3. connect *branch_crs* as the next CRS after *fork_crs*
2.3.1.2.8.4. CRS = *branch_crs*
2.3.1.2.8.5. *branch_activity* = new default activity in *or_fork_task*
2.3.1.2.8.6. connect *branch_activity* as the next activity after *or_fork_activity*
2.3.1.2.8.7. continue traversal of branch path using the current item in *next_edges* (see Section 5.2.)
2.3.1.2.8.8. continue to next item in *next_edges*
// deal with the main path
2.3.1.2.9. *branch_crs* = new CRS
2.3.1.2.10. set *branch_crs* as being on the main path
2.3.1.2.11. connect *branch_crs* as the next CRS after *fork_crs*
2.3.1.2.12. CRS = *branch_crs*
2.3.1.2.13. *branch_activity* = new default activity in *or_fork_task*
2.3.1.2.14. connect *branch_activity* as the next activity after *or_fork_activity*
2.3.1.2.15. continue traversal of main path using the first item in *next_edges* (see Section 5.2.)
// the or fork is not in a component
2.3.1.3. **else**
2.3.1.3.1. *or_fork_task* = new default task
2.3.1.3.2. *or_fork_entry* = new entry for *or_fork_task*
2.3.1.3.3. update the last activity on the CRS as making a call to *or_fork_entry*
2.3.1.3.4. push *or_fork_entry* on CRS
2.3.1.3.5. *or_fork_activity* = new default activity in *or_fork_task* for *or_fork*
2.3.1.3.6. connect *or_fork_activity* as the first activity of *or_fork_entry*
2.3.1.3.7. get *next_edges* for *or_fork* (see Section 5.1.1.)
2.3.1.3.8. skip over the first item in *next_edges*
2.3.1.3.9. *fork_crs* = CRS
// deal with the secondary branches
2.3.1.3.10. **while** *next_edges* is not done
2.3.1.3.10.1. *branch_crs* = new CRS

2.3.1.3.10.2. set *branch_crs* as being on a branch path
2.3.1.3.10.3. connect *branch_crs* as the next CRS after *fork_crs*
2.3.1.3.10.4. CRS = *branch_crs*
2.3.1.3.10.5. *branch_activity* = new default activity in *or_fork_task*
2.3.1.3.10.6. connect *branch_activity* as the next branch activity after *or_fork_activity*
2.3.1.3.10.7. push *branch_activity* on the CRS
2.3.1.3.10.8. continue traversal of branch path using the current item in *next_edges* (see Section 5.2.)
2.3.1.3.10.9. continue to next item in *next_edges*

// deal with the main path
2.3.1.3.11. *branch_crs* = new CRS
2.3.1.3.12. set *branch_crs* as being on the main path
2.3.1.3.13. connect *branch_crs* as the next CRS after *fork_crs*
2.3.1.3.14. CRS = *branch_crs*
2.3.1.3.15. *branch_activity* = new default activity in *or_fork_task*
2.3.1.3.16. connect *branch_activity* as the next branch activity after *or_fork_activity*
2.3.1.3.17. push *branch_activity* on the CRS
2.3.1.3.18. continue traversal of main path using the first item in *next_edges* (see Section 5.2.)

2.3.2. **break**

2.4. **case** *or_join*

2.4.1. **if** *or_join* has not been visited **then**

// deal with a secondary branch for the first time
// create and connect branch and join activities, if necessary
// create a new task and entry for the activities
2.4.1.1. mark *or_join* as visited
// check if the or join is in a component
2.4.1.2. **if** *or_join* is contained in a component **then**
2.4.1.2.1. *or_join_task* = task corresponding to the component
2.4.1.2.2. **if** currently processing a loop body **then**
2.4.1.2.2.1. *or_join_task* = clone of *or_join_task* for the loop body
2.4.1.2.3. *branch_activity* = new default activity in *or_join_task*
2.4.1.2.4. connect *branch_activity* as the next activity after the activity last added to *or_join_task*
2.4.1.2.5. *or_join_activity* = new default activity for *or_join* not added to *or_join_task*
2.4.1.2.6. connect *or_join_activity* as next join activity after *branch_activity*

// the or join is not in a component
2.4.1.3. **else**
2.4.1.3.1. *or_join_task* = new default task
2.4.1.3.2. *or_join_branch_entry* = new entry for *or_join_task*
2.4.1.3.3. update the last activity on the CRS as making a call to *or_join_branch_entry*
2.4.1.3.4. push *or_join_entry* on CRS
2.4.1.3.5. *branch_activity* = new default activity in *or_join_task*

2.4.1.3.6. connect *branch_activity* as the first activity of *or_join_branch_entry*
2.4.1.3.7. *or_join_activity* = new default activity for *or_join* not added to *or_join_task*
2.4.1.3.8. connect *or_join_activity* as next join activity after *branch_activity*
2.4.2. **else**
// create and connect branch and join activities, if necessary
// create a new task and entry for the activities
// check if the or join is in a component
2.4.2.1. **if** *or_join* is contained in a component **then**
2.4.2.1.1. get *or_join_task* corresponding to the component
2.4.2.1.2. **if** currently processing a loop body **then**
2.4.2.1.2.1. *or_join_task* = clone of *or_join_task* for the loop body
2.4.2.1.3. *branch_activity* = new default activity in *or_join_task*
2.4.2.1.4. connect *branch_activity* as the next activity after the activity last added to *or_join_task*
2.4.2.1.5. get *or_join_activity* corresponding to *or_join*
2.4.2.1.6. connect *or_join_activity* as next join activity after *branch_activity*
// the or join is not in a component
2.4.2.2. **else**
2.4.2.2.1. get *or_join_task* corresponding to the *or_join*
2.4.2.2.2. *or_join_branch_entry* = new entry for *or_join_task*
2.4.2.2.3. update the last activity on the CRS as making a call to *or_join_branch_entry*
2.4.2.2.4. push *or_join_entry* on CRS
2.4.2.2.5. *branch_activity* = new default activity in *or_join_task*
2.4.2.2.6. connect *branch_activity* as the first activity of *or_join_branch_entry*
2.4.2.2.7. get *or_join_activity* corresponding to *or_join*
2.4.2.2.8. connect *or_join_activity* as next join activity after *branch_activity*
// path traversal logic
2.4.2.3. **if** CRS on the main path **then**
2.4.2.3.1. add *or_join_activity* to the *or_join_task*
2.4.2.3.2. continue normal path traversal (see Section 5.2.)
2.4.3. **break**
2.5. **case** *synchronization*
2.5.1. get *previous_edges* for *synchronization* (see Section 5.1.1.)
2.5.2. get *next_edges* for *synchronization* (see Section 5.1.1.)
2.5.3. **if** a single *previous_edge* and multiple *next_edges* **then**
2.5.3.1. this is an *and_fork*
2.5.3.2. **if** *and_fork* has not been visited **then**
// create and connect fork activities, if necessary create a
// new task and entry for the activities
2.5.3.2.1. mark *and_fork* as visited
// check if the and fork is in a component
2.5.3.2.2. **if** *and_fork* is contained in a component **then**

2.5.3.2.2.1. *and_fork_task* = task corresponding to the component
2.5.3.2.2.2. **if** currently processing a loop body **then**
2.5.3.2.2.2.1. *and_fork_task* = clone of *and_fork_task* for the loop body
2.5.3.2.2.3. *and_fork_activity* = new default activity in *and_fork_task*
2.5.3.2.2.4. connect *and_fork_activity* as the next activity after the activity last added to *and_fork_task*
2.5.3.2.2.5. get *next_edges* for *and_fork* (see Section 5.1.1.)
2.5.3.2.2.6. skip over the first item in *next_edges*
2.5.3.2.2.7. *fork_crs* = CRS
// deal with the secondary branches
2.5.3.2.2.8. **while** *next_edges* is not done
2.5.3.2.2.8.1. *branch_crs* = new CRS
2.5.3.2.2.8.2. set *branch_crs* as being on a branch path
2.5.3.2.2.8.3. connect *branch_crs* as the next CRS after *fork_crs*
2.5.3.2.2.8.4. CRS = *branch_crs*
2.5.3.2.2.8.5. *branch_activity* = new default activity in *and_fork_task*
2.5.3.2.2.8.6. connect *branch_activity* as the next activity after *and_fork_activity*
2.5.3.2.2.8.7. continue traversal of branch path using the current item in *next_edges* (see Section 5.2.)
2.5.3.2.2.8.8. continue to next item in *next_edges*
// deal with the main path
2.5.3.2.2.9. *branch_crs* = new CRS
2.5.3.2.2.10. set *branch_crs* as being on the main path
2.5.3.2.2.11. connect *branch_crs* as the next CRS after *fork_crs*
2.5.3.2.2.12. CRS = *branch_crs*
2.5.3.2.2.13. *branch_activity* = new default activity in *and_fork_task*
2.5.3.2.2.14. connect *branch_activity* as the next activity after *and_fork_activity*
2.5.3.2.2.15. continue traversal of main path using the first item in *next_edges* (see Section 5.2.)
// the and fork is not in a component
2.5.3.2.3. **else**
2.5.3.2.3.1. *and_fork_task* = new default task
2.5.3.2.3.2. *and_fork_entry* = new entry for *and_fork_task*
2.5.3.2.3.3. update the last activity on the CRS as making a call to *and_fork_entry*
2.5.3.2.3.4. push *and_fork_entry* on CRS
2.5.3.2.3.5. *and_fork_activity* = new default activity in *and_fork_task*
2.5.3.2.3.6. connect *and_fork_activity* as the first activity of *and_fork_entry*
2.5.3.2.3.7. get *next_edges* for *and_fork* (see Section 5.1.1.)
2.5.3.2.3.8. skip over the first item in *next_edges*
2.5.3.2.3.9. *fork_crs* = CRS
// deal with the secondary branches

2.5.3.2.3.10. **while** *next_edges* is not done
2.5.3.2.3.10.1. *branch_crs* = new CRS
2.5.3.2.3.10.2. set *branch_crs* as being on a branch path
2.5.3.2.3.10.3. connect *branch_crs* as the next CRS after *fork_crs*
2.5.3.2.3.10.4. CRS = *branch_crs*
2.5.3.2.3.10.5. *branch_activity* = new default activity in *and_fork_task*
2.5.3.2.3.10.6. connect *branch_activity* as the next branch activity after *and_fork_activity*
2.5.3.2.3.10.7. push *branch_activity* on the CRS
2.5.3.2.3.10.8. continue traversal of branch path using the current item in *next_edges* (see Section 5.2.)
2.5.3.2.3.10.9. continue to next item in *next_edges*
// deal with the main path
2.5.3.2.3.11. *branch_crs* = new CRS
2.5.3.2.3.12. set *branch_crs* as being on the main path
2.5.3.2.3.13. connect *branch_crs* as the next CRS after *fork_crs*
2.5.3.2.3.14. CRS = *branch_crs*
2.5.3.2.3.15. *branch_activity* = new default activity in *and_fork_task*
2.5.3.2.3.16. connect *branch_activity* as the next branch activity after *and_fork_activity*
2.5.3.2.3.17. push *branch_activity* on the CRS
2.5.3.2.3.18. continue traversal of main path using the first item in *next_edges* (see Section 5.2.)
2.5.4. **else if** multiple *previous_edges* and a single *next_edge* **then**
2.5.4.1. this is an *and_join*
2.5.4.2. **if** *and_join* has not been visited **then**
// deal with a secondary branch for the first time
// create and connect branch and join activities, if necessary
// create a new task and entry for the activities
2.5.4.2.1. mark *and_join* as visited
// check if the and join is in a component
2.5.4.2.2. **if** *and_join* is contained in a component **then**
2.5.4.2.2.1. *and_join_task* = task corresponding to the component
2.5.4.2.2.2. **if** currently processing a loop body **then**
2.5.4.2.2.2.1. *and_join_task* = clone of *and_join_task* for the loop body
2.5.4.2.2.3. *branch_activity* = new default activity in *and_join_task*
2.5.4.2.2.4. connect *branch_activity* as the next activity after the activity last added to *and_join_task*
2.5.4.2.2.5. *and_join_activity* = new default activity for*and_join* not added to *and_join_task*
2.5.4.2.2.6. connect *and_join_activity* as next join activity after *branch_activity*
// join is not in component
2.5.4.2.3. **else**
2.5.4.2.3.1. *and_join_task* = new default task
2.5.4.2.3.2. *and_join_branch_entry* = new entry for *and_join_task*

2.5.4.2.3.3. update the last activity on the CRS as making a call to *and_join_branch_entry*
2.5.4.2.3.4. push *and_join_entry* on CRS
2.5.4.2.3.5. *branch_activity* = new default activity in *and_join_task*
2.5.4.2.3.6. connect *branch_activity* as the first activity of *and_join_branch_entry*
2.5.4.2.3.7. *and_join_activity* = new default activity for*and_join* not added to *and_join_task*
2.5.4.2.3.8. connect *and_join_activity* as next join activity after *branch_activity*

2.5.4.3. **else**
// create and connect branch and join activities, if necessary
// create a new task and entry for the activities
// check if the and join is in a component
2.5.4.3.1. **if** *and_join* is contained in a component **then**
2.5.4.3.1.1. get *and_join_task* corresponding to the component
2.5.4.3.1.2. **if** currently processing a loop body **then**
2.5.4.3.1.2.1. *and_join_task* = clone of *and_join_task* for the loop body
2.5.4.3.1.3. *branch_activity* = new default activity in *and_join_task*
2.5.4.3.1.4. connect *branch_activity* as the next activity after the activity last added to *and_join_task*
2.5.4.3.1.5. get *and_join_activity* corresponding to *and_join*
2.5.4.3.1.6. connect *and_join_activity* as next join activity after *branch_activity*
// the and join is not in a component
2.5.4.3.2. **else**
2.5.4.3.2.1. get *and_join_task* corresponding to the *and_join*
2.5.4.3.2.2. *and_join_branch_entry* = new entry for *and_join_task*
2.5.4.3.2.3. update the last activity on the CRS as making a call to *and_join_branch_entry*
2.5.4.3.2.4. push *and_join_entry* on CRS
2.5.4.3.2.5. *branch_activity* = new default activity in *and_join_task*
2.5.4.3.2.6. connect *branch_activity* as the first activity of *and_join_branch_entry*
2.5.4.3.2.7. get *and_join_activity* corresponding to *and_join*
2.5.4.3.2.8. connect *and_join_activity* as next join activity after *branch_activity*
// path traversal logic
2.5.4.3.3. **if** CRS on the main path **then**
2.5.4.3.3.1. add *and_join_activity* to the *and_join_task*
2.5.4.3.3.2. continue normal path traversal (see Section 5.2.)

2.5.5. **else**
2.5.5.1. this is an *and_synchronization* (an *and_join* followed by an *and_fork*)
2.5.5.2. treat as an *and_join*
2.5.5.2.1. **goto** step 2.5.4.1. above

2.5.5.2.2. **stop** after step 2.5.4.3.3.1.
2.5.5.3. treat as an *and_fork*
2.5.5.3.1. **goto** step 2.5.3.1. above
2.5.6. **break**
2.6. **case** *loop_head*
2.6.1. **if** *loop_head* has not been visited **then**
2.6.1.1. mark *loop_head* as visited
2.6.1.2. *old_crs* = CRS
2.6.1.3. *loop_crs* = new CRS
2.6.1.4. CRS = *loop_crs*
2.6.1.5. get *next_edges* for *loop_head* (see Section 5.1.1.)
// check if loop head is contained in a component
2.6.1.6. **if** *loop_head* is contained in a component **then**
2.6.1.6.1. *loop_head_task* = task corresponding to the component
// handle possible loop within a loop
2.6.1.6.2. **if** currently processing a loop body **then**
2.6.1.6.2.1. *loop_head_task* = clone of *loop_head_task* for the loop body
// create loop activity and pseudo-task for loop body
2.6.1.6.3. *loop_head_activity* = new default activity in *loop_head_task* for *loop_head*
2.6.1.6.4. connect *loop_head_activity* as the next activity after the activity last added to *loop_head_task*
2.6.1.6.5. *loop_body_task* = new clone of *loop_head_task*
2.6.1.6.6. *loop_body_entry* = new entry for *loop_body_task*
2.6.1.6.7. update *loop_head_activity* as making a synchronous call to *loop_body_entry*
2.6.1.6.8. push *loop_body_entry* on CRS
2.6.1.6.9. *loop_body_activity* = new default activity in *loop_body_task*
2.6.1.6.10. connect *loop_body_activity* as first activity of *loop_body_entry*
2.6.1.6.11. continue traversal of the loop body path using the last item in *next_edges* (see Section 5.2.)
// prepare to handle reply from loop body
2.6.1.6.12. *loop_body_reply_activity* = new default activity in *loop_body_task*
2.6.1.6.13. update *loop_body_reply_activity* as making a reply to *loop_body_entry*
2.6.1.6.14. *handle_reply_activity* = new default activity in *loop_head_task*
2.6.1.6.15. connect *handle_reply_activity* as the next activity after *loop_head_activity*
// loop head is not contained in a component
2.6.1.7. **else**
// create pseudo-task for loop head
2.6.1.7.1. *loop_head_task* = new default task
2.6.1.7.2. *loop_head_entry* = new entry for *loop_head_task*
2.6.1.7.3. update the last activity on the CRS as making a call to *loop_head_entry*

2.6.1.7.4. push *loop_head_entry* on CRS
// create loop activity and pseudo-task for loop body
2.6.1.7.5. *loop_head_activity* = new default activity in *loop_head_task* for *loop_head*
2.6.1.7.6. connect *loop_head_activity* as the first activity of *loop_head_entry*
2.6.1.7.7. *loop_body_task* = new clone of *loop_head_task*
2.6.1.7.8. *loop_body_entry* = new entry for *loop_body_task*
2.6.1.7.9. update *loop_head_activity* as making a synchronous call to *loop_body_entry*
2.6.1.7.10. push *loop_body_entry* on CRS
2.6.1.7.11. *loop_body_activity* = new default activity in *loop_body_task*
2.6.1.7.12. connect *loop_body_activity* as first activity of *loop_body_entry*
2.6.1.7.13. continue traversal of the loop body path using the last item in *next_edges* (see Section 5.2.)
2.6.1.7.14. *loop_body_reply_activity* = new default activity in *loop_body_task*
2.6.1.7.15. update *loop_body_reply_activity* as making a reply to *loop_body_entry*
2.6.1.7.16. *handle_reply_activity* = new default activity in *loop_head_task*
2.6.1.7.17. connect *handle_reply_activity* as the next activity after *loop_head_activity*
// restore main path variables
2.6.1.8. CRS = *old_crs*
2.6.1.9. **delete** *loop_crs*
2.6.1.10. continue traversal of the main path using the first item in *next_edges* (see Section 5.2.)
2.6.2. **break**
2.7. **case** *stub*
// determine if the plug-in is there
2.7.1. **if** *stub* has properly bound plug-in **then**
2.7.1.1. *plug_in_entry_point* = plug-in *start_point* bound to the current *stub* input
2.7.1.2. mark *plug_in_entry_point* as visited
2.7.1.3. get *next_edges* for *plug_in_entry_point* (see Section 5.1.1.)
2.7.1.4. continue traversal of the plug-in path (see Section 5.2.)
// the plug-in is not there
2.7.2. **else**
2.7.2.1. treat stub as a *responsibility*
2.7.2.1.1. **goto** step 2.2.1.
2.7.2.2. use the stub output bound to the current stub input as the *next_edge*
2.7.2.3. continue path traversal (see Section 5.2.)
2.7.3. **break**
2.8. **case** *end_point*
// check if the end point is in a plug-in
2.8.1. **if** *end_point* is bound to a *stub* exit **then**

2.8.1.1. use the stub output bound to *end_point* as the *next_edge*
2.8.1.2. continue path traversal (see Section 5.2.)
// the end point is not in a plug-in
2.8.2. **else**
// check if the end point is in a component
2.8.2.1. **if** *end_point* is contained in a component **then**
2.8.2.1.1. *end_task* = task corresponding to the component
2.8.2.1.2. **if** currently processing a loop body **then**
2.8.2.1.2.1. *end_task* = clone of *end_task* for the loop body
2.8.2.1.3. *end_activity* = new default activity in *end_task*
2.8.2.1.4. connect *end_activity* as the next activity after the last activity added to *end_task*
2.8.2.1.5. **if** *end_point* is connected to a *start_point* **then**
2.8.2.1.5.1. this is a closed system
2.8.2.1.5.2. *end_entry* = pop entry off the CRS
2.8.2.1.5.3. update *end_activity* as making a reply to *end_entry*
2.8.2.1.5.4. *reference_activity* = pop item off the CRS
2.8.2.1.5.5. update *reference_activity* as making a synchronous call
2.8.2.1.6. **else**
2.8.2.1.6.1. this is an open system
2.8.2.1.6.2. **while** the CRS is not empty
2.8.2.1.6.2.1. *crs_element* = pop item off the CRS
2.8.2.1.6.2.2. **if** *crs_element* is an activity **then**
2.8.2.1.6.2.2.1. update activity as making an asynchronous call
// the end point is not in a component
2.8.2.2. **else**
2.8.2.2.1. **if** *end_point* is connected to a *start_point* **then**
2.8.2.2.1.1. this is a closed system
2.8.2.2.1.2. *first_task* = first task called by the reference task corresponding to the *start_point*
// check if this is a direct synchronous reply
2.8.2.2.1.3. **if** last task on the CRS == *first_task* **then**
2.8.2.2.1.3.1. this is a direct synchronous reply to the reference task
2.8.2.2.1.3.2. *reply_activity* = pop item off the CRS
2.8.2.2.1.3.3. *reply_entry* = pop item off the CRS
2.8.2.2.1.3.4. update *reply_activity* as making a reply to *reply_entry*
2.8.2.2.1.3.5. *reference_activity* = pop item off the CRS
2.8.2.2.1.3.6. update *reference_activity* as making a synchronous call
// this is not a direct synchronous
2.8.2.2.1.4. **else**
2.8.2.2.1.4.1. this is a forwarded reply to the reference task
2.8.2.2.1.4.2. *reply_activity* = pop item off the CRS
2.8.2.2.1.4.3. *reply_entry* = pop item off the CRS
2.8.2.2.1.4.4. update *reply_activity* as making a reply to *reply_entry*
// deal with the intermediary forwarding components

2.8.2.2.1.4.5. **while** last task on the CRS != *first_task*
2.8.2.2.1.4.5.1. *forward_entry* = *reply_entry*
2.8.2.2.1.4.5.2. *reply_activity* = pop item off the CRS
2.8.2.2.1.4.5.3. update *reply_activity* as making a synchronous call
2.8.2.2.1.4.5.4. *reply_entry* = pop item off the CRS
2.8.2.2.1.4.5.5. update *reply_activity* as making a reply to *reply_entry*
2.8.2.2.1.4.5.6. update *reply_entry* as forwarding its call to *forward_entry*
// deal with the first forwarding component
2.8.2.2.1.4.6. *forward_entry* = *reply_entry*
2.8.2.2.1.4.7. *reply_activity* = pop item off the CRS
2.8.2.2.1.4.8. update *reply_activity* as making a synchronous call to *forward_entry*
2.8.2.2.1.4.9. *reply_entry* = pop item off the CRS
2.8.2.2.1.4.10. update *reply_activity* as making a reply to *reply_entry*
2.8.2.2.1.4.11. update *reply_entry* as forwarding its call to *forward_entry*
2.8.2.2.1.4.12. *reference_activity* = pop item off the CRS
2.8.2.2.1.4.13. update *reference_activity* as making a synchronous call
2.8.2.2.2. **else**
2.8.2.2.2.1. this is an open system
2.8.2.2.2.2. *end_task* = new default activity
2.8.2.2.2.3. *end_entry* = new entry for *end_task*
2.8.2.2.2.4. update the last activity on the CRS as making a call to *end_entry*
// empty the CRS and treat everything left as asynchronous
2.8.2.2.2.5. **while** the CRS is not empty
2.8.2.2.2.5.1. *crs_element* = pop item off the CRS
2.8.2.2.2.5.2. **if** *crs_element* is an activity **then**
2.8.2.2.2.5.2.1. update activity as making an asynchronous call
2.8.3. **break**
2.9. **default**
2.9.1. no LQN objects need to be created
2.9.2. continue path traversal (see Section 5.2.)
2.9.3. **break**

5.6. The Call and Reply Stack

The Call and Reply Stack (CRS) stores activities and entries involved in sending and receiving unresolved messages. The CRS acts as kind of message memory that can be checked whenever component boundaries are crossed to see if there are calls waiting for replies.

- LQN entries and activities involved in unresolved calls are pushed on the CRS

- LQN entries and activities involved in calls that are being resolved are popped off the CRS and have their call types specified
- any LQN entries and activities remaining on the CRS when the end of the path is reached are updated as being involved in asynchronous calls - no calling information is ever lost

The CRS has two modes: main path or branch path. A CRS on the main path can remove elements from preceding CRSs, whereas a CRS on a branch path can only remove its own elements and cannot remove elements from any preceding CRSs. Whenever a fork is encountered along the path the CRS is also forked. The old part of the CRS is the now a fork CRS and it is set to branch mode, whereas the new parts are branch CRSs. As the branches of the fork are followed, the branch CRS's can check the contents of the fork CRS, but are not allowed to actually pop any elements from it. When the last branch is being traversed, it is assumed to be part of the main path and the fork CRS is reset to main path mode. Elements can now be popped of the CRS if needed.

Chapter 6 - Validation

This chapter describes the example systems used to validate the UCM2LQN conversion algorithm. The examples are that of a simple connection of a telephone call (POTS), a distributed ticket reservation system (TRS), and a group communication server (GCS). The UCM models for these systems were all developed at Carleton University and reflect an “in-house” style. These models validate the UCM2LQN converter by combining several of the UCM and LQN correspondence patterns in complex models that represent real systems.

The LQN models generated by UCM2LQN were used as inputs to the jLqnDef, LQNS, and ParaSRVN tools. This checked both the syntax and the semantics of the LQNs. All three tools have a syntax checker and will not load badly formed LQN files. jLqnDef can also generate a graphical view of the LQN and thus allow for a visual check of its composition. The ParaSRVN simulator further validates the semantics by successfully completing the required simulation runs. The LQNS analytical solver can also be used to verify the syntax and semantics of the LQN output, with the caveat that LQNS is unable to solve some valid models because of limitations in the approximations.[25]

6.1. Plain Old Telephone System

The Plain Old Telephone System (POTS) call connection example described in this section is the basis of a larger UCM design for a telephony system that was originally developed by Daniel Amyot and the author in the summer of 1998 for the feature interaction detection contest organized with the occasion of that year’s 5th Feature Interaction Workshop.[2] The aim of the contest was to detect as many feature interactions as possible when including a set of supplied features on top of the POTS system. The telephony system was specified using Chisel diagrams.[1] A base diagram described POTS and provided connection points for features. Each feature was described in isolation by a separate sub-diagram.[31] It was up to the contestants to interpret the diagrams and any descriptions provided, integrate the features into the overall telephony model, and detect potential feature interactions. Unfortunately, the UCM design we developed was not ready on time for submission to the contest, but it did provide a starting point for further research into using UCMs and LOTOS models to detect feature interactions.[4][6]

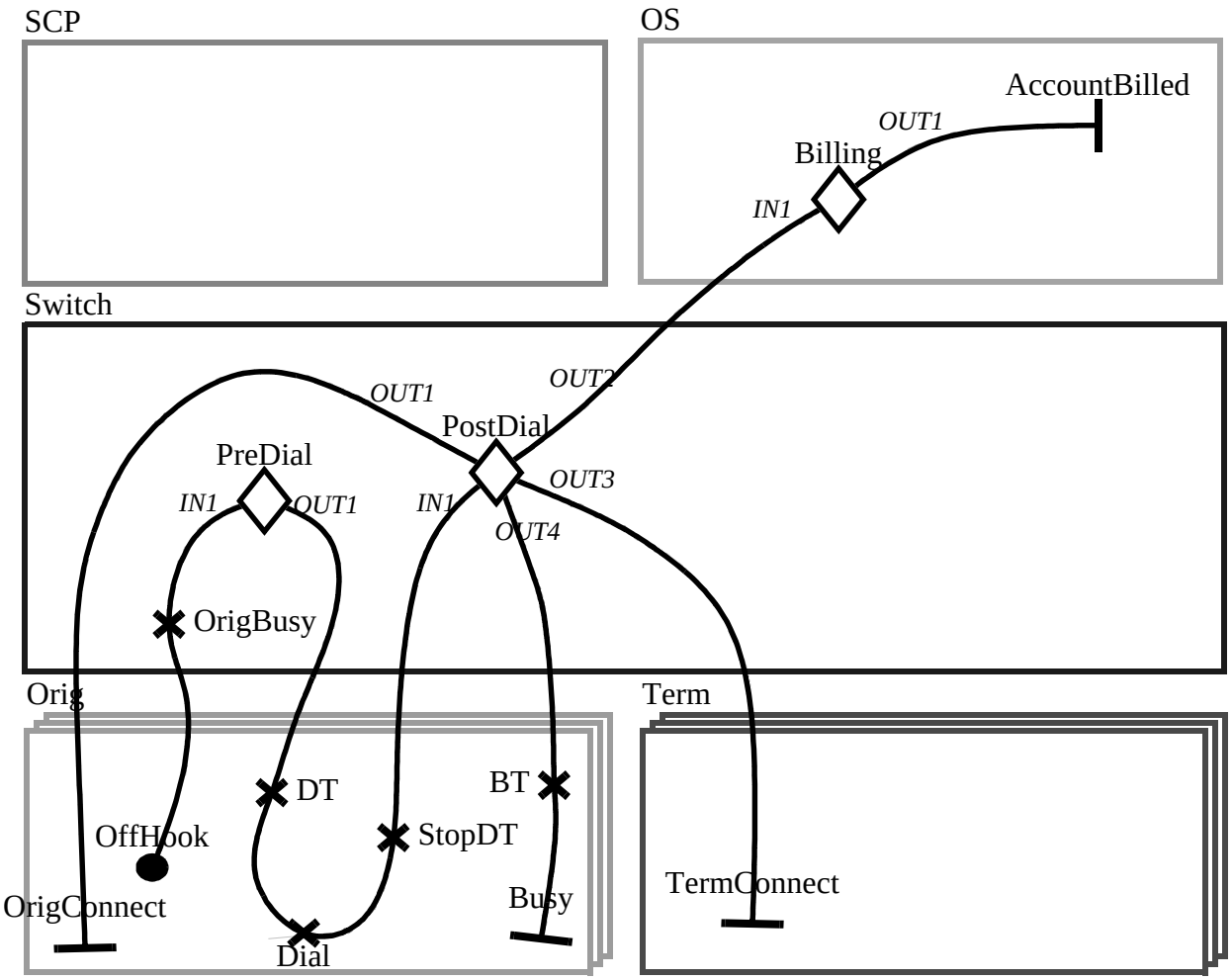


Figure 6-1: UCMNav root map for the POTS example.

6.1.1. POTS Design

6.1.1.1. POTS UCM Root Map

Figure 6-1 shows the UCM root map for the POTS system. The components for all the POTS maps are as follows:

- *Orig*: the caller's telephone set
- *Term*: the callee's telephone set
- *Switch*: the telephone company's switch gear
- *SCP*: the Service Control Point that processes IN features (not used in the POTS scenario)

- *OS*: the Operations System that does the billing

The POTS root map features the following stubs:

- *PreDial*: features that are activated before the number is dialed
- *PostDial*: features that are activated after the number is dialed
- *Billing*: different billing schemes depending on the kind of connection and which features are invoked

For POTS operation the PreDial stub has a default plug-in that merely connects the input and output paths. Similarly, the Billing stub has a straight path connecting its input and output. The path has a single responsibility to log the start time of the connection between the caller and the callee. The PostDial stub has more functionality and is discussed in further detail in Section 6.1.1.2.

PostDial Stub Path Binding	PostDial Plug-In Path Binding
IN1	call
OUT1	orig_connected
OUT2	billing
OUT3	term_connected
OUT4	busy

Table 6-1: Stub and plug-in bindings for the PostDial stub shown in Figure 6-1 and the PostDial UCM shown in Figure 6-2.

6.1.1.2. POTS PostDial Plug-In

The PostDial plug-in either contacts the callee and establishes a connection or notifies the caller that the callee is busy. The plug-in map is shown in Figure 6-2. The connection bindings to the PostDial stub are listed in Table 6-1. The PostDial map features the following stubs:

- *ProcessCall*: features dealing with making a connection
- *ProcessBusy*: features associated with the callee being busy
- *NumberDisplay*: feature displaying the caller's number

For POTS, both the ProcessBusy and NumberDisplay stubs have default plug-ins without any responsibilities. The plug-in for the ProcessCall stub is discussed in Section 6.1.1.3.

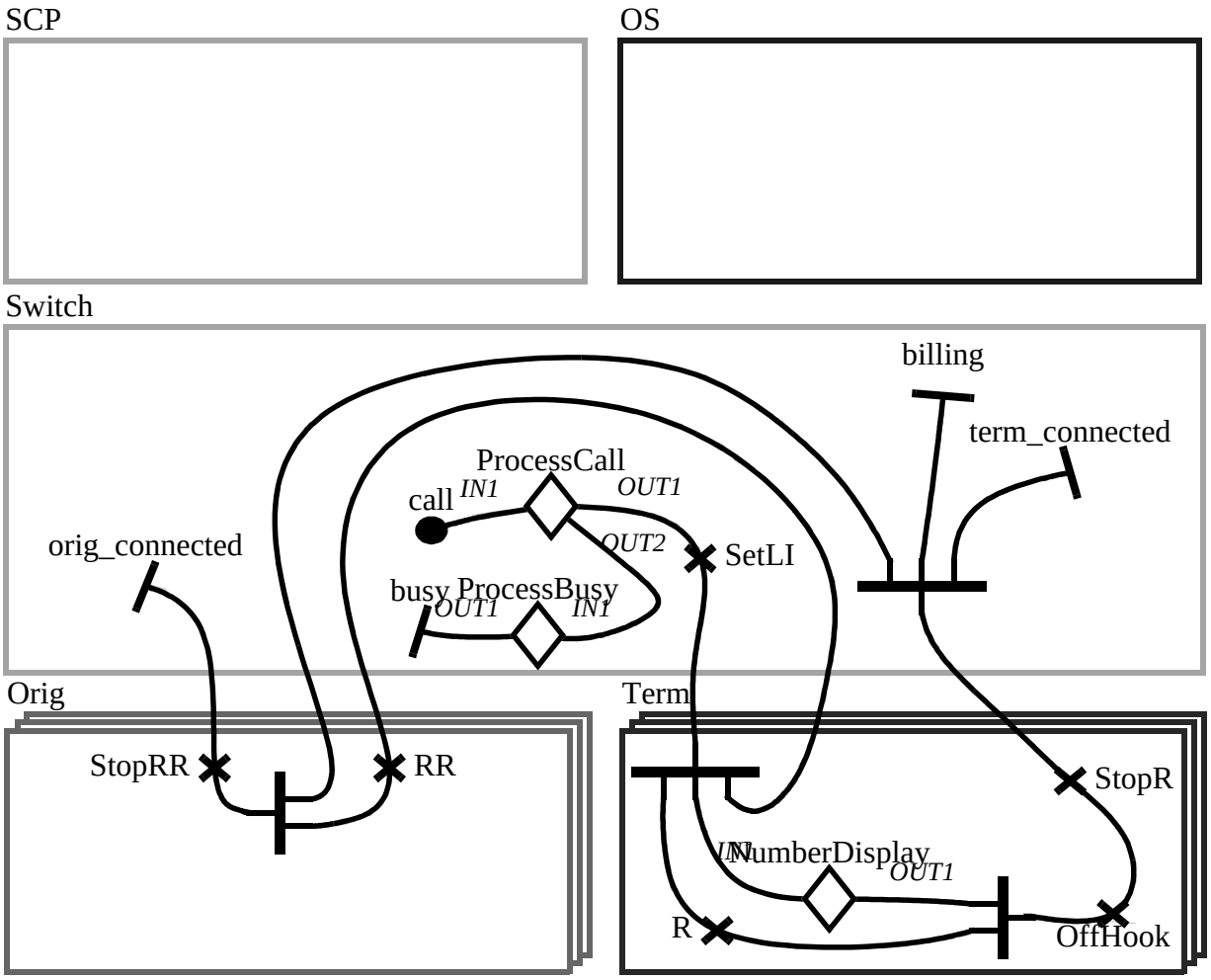


Figure 6-2: UCMNav plug-in map for the PostDial stub shown in Figure 6-1 for the POTS example.

6.1.1.3. POTS ProcessCall Plug-In

The ProcessCall plug-in for normal POTS operation checks whether the callee happens to be idle or not. If the callee is idle then its status is changed to busy and the process of making the connection is started. Otherwise the process of notifying the caller that the callee is busy starts. The UCM is shown in Figure 6-3.

The *POTS* start point is bound to the ProcessCall stub's *IN1* input. The *idle* and *busy* end point are bound to the stub's *OUT1* and *OUT2* outputs respectively.

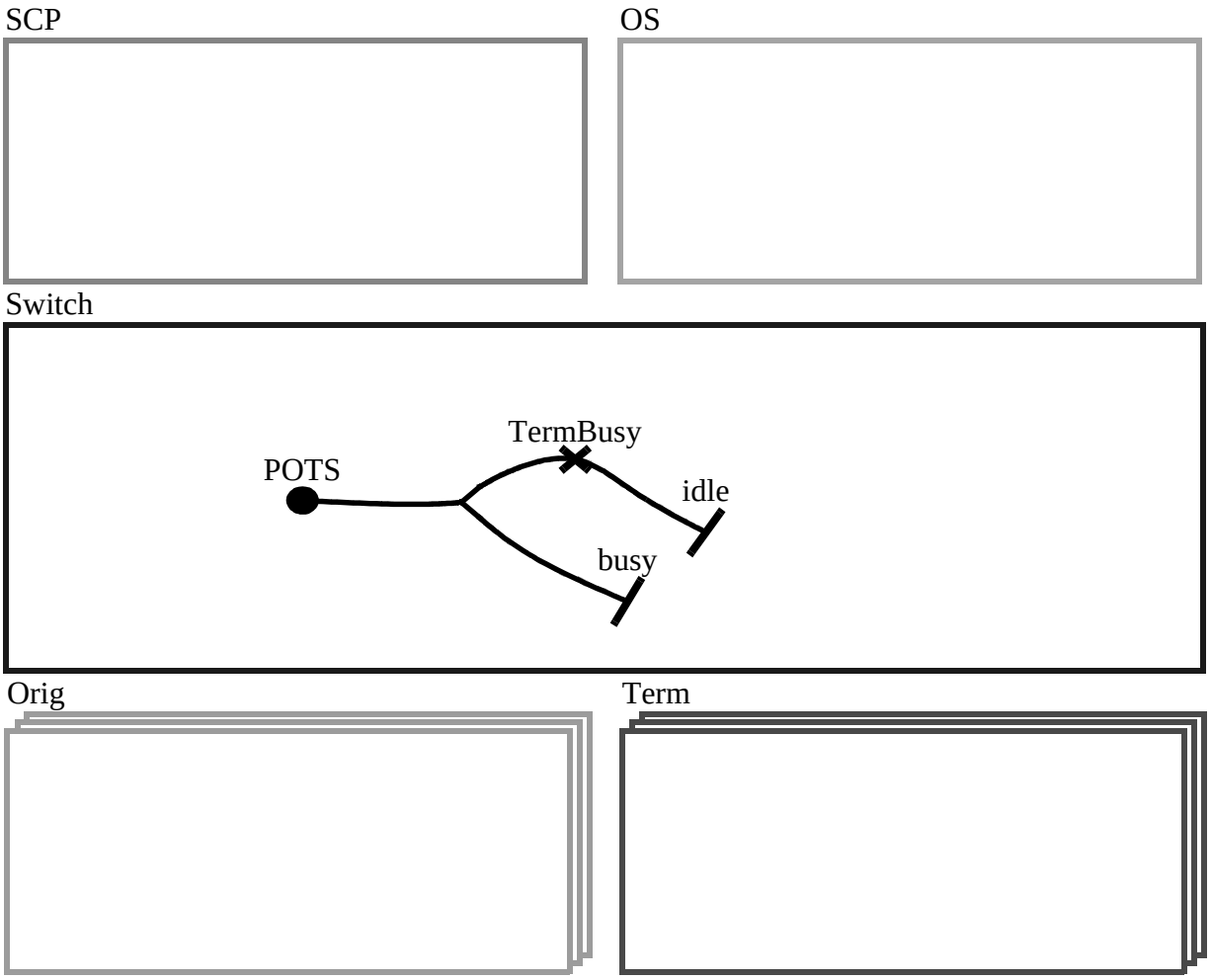


Figure 6-3: UCMNav plug-in map for the ProcessCall stub shown in Figure 6-2 for the POTS example.

6.1.2. POTS Usage

POTS has two possible scenarios that can happen when attempting to make a call. The call can either be set up successfully or the callee can be busy. If the call is placed successfully, the scenario unfolds as follows:

1. the originator (caller) picks up the receiver
2. the switch notes that the originator is now busy
3. the originator gets a dial tone
4. the originator dials the desired terminator's number (callee)

5. the dial tone stops
6. the switch checks and finds that the terminator is currently idle
7. the switch stores the originator's number as the terminator's last incoming number
8. the terminator gets a ring and the originator gets a remote ringing tone
9. the terminator picks up the receiver
10. the terminator's ring stops
11. the originator's remote ringing tone stops and the billing details are recorded by the operations system
12. the connection is now made

Otherwise, an unsuccessful call connection scenario unfolds as follows:

step 1 through step 5 are the same

6. the switch checks and finds that the terminator is currently busy
7. the originator gets a busy tone
8. the connection is not made

6.1.3. POTS LQN Conversion Results

The LQN model for POTS is shown in Figure 6-4 and Figure 6-5. The model features asynchronous and synchronous messaging, alternate and parallel paths, and stubs. The resulting LQN is syntactically sound and can be loaded into all three LQN tools.

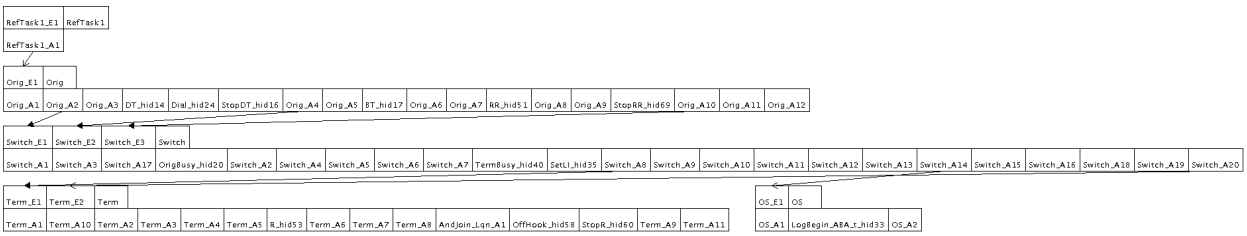


Figure 6-4: POTS LQN generated by the UCM2LQN converter from the UCM shown in Figure 6-1. (output from jLqnDef)

Figure 6-4 shows that a reference task was indeed created for the start point. Since the end of the UCM path did not come back to the start point, the system has open arrivals and the refer-

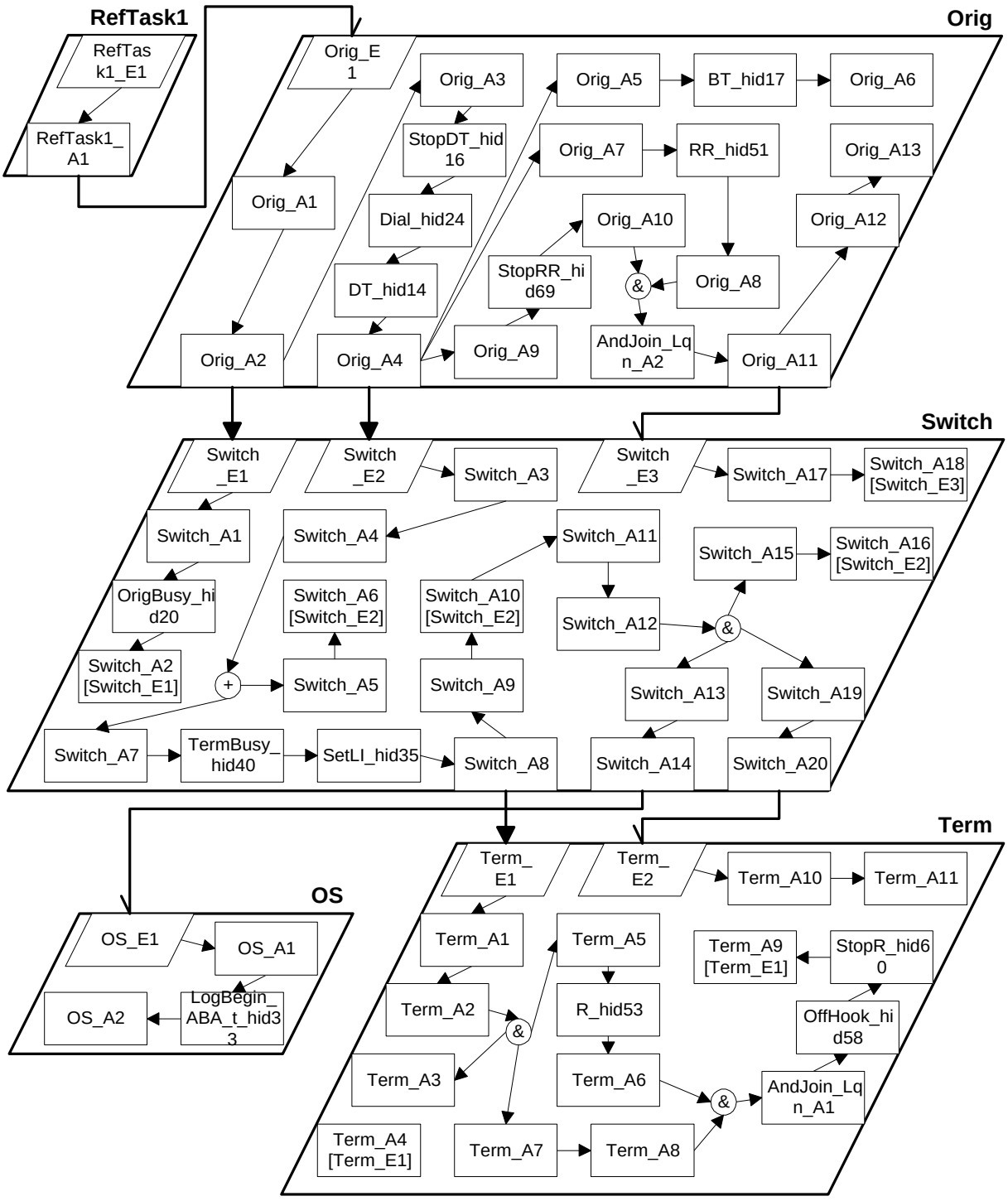


Figure 6-5: POTS LQN diagram showing the activity connections, based on the output of the UCM2LQN converter from the UCM shown in Figure 6-1.

ence task sends an asynchronous message to the *Orig* task indicating that the receiver is being picked up. The *Orig* task interacts synchronously with the *Switch*. The originator first requests a dial tone, then that a connection be established with the party whose number was dialed, and finally that the connection be "enabled" so the two parties can talk. The *Switch* in turn first makes a synchronous call to the *Term* process to create the call. After the callee picks up, the *Switch* sends an asynchronous message the *Term* process to confirm that the two parties can now talk to each other. The *Switch* also sends an asynchronous message to log the start time of the connection to the *OS*.

The UCM2LQN conversion is semantically correct and can be simulated by ParaSRVN, but the model cannot be solved by LQNS due to the presence of a distributed fork and join. Overcoming this problem is something that will be addressed in the future.

6.2. Ticket Reservation System

The Ticket Reservation System (TRS) allows users to browse through a catalogue of events and seat availability, and to buy tickets using a credit card. The TRS is a tutorial examples used as part of a course on the Design of High Performance Software.[49] The UCM design used in this section is based on that example.

6.2.1. TRS Design

The UCM design for the TRS is shown in Figure 6-6. The TRS components are as follows:

- *User*: TRS customer
- *WebServer*: web interface to the TRS, executes CGI scripts
- *Netware*: the underlying network software and the network itself
- *CCReq*: credit card verification and authorization server
- *Database*: database server

6.2.2. TRS Usage

The TRS can be used either to browse events by displaying an event schedule and seating

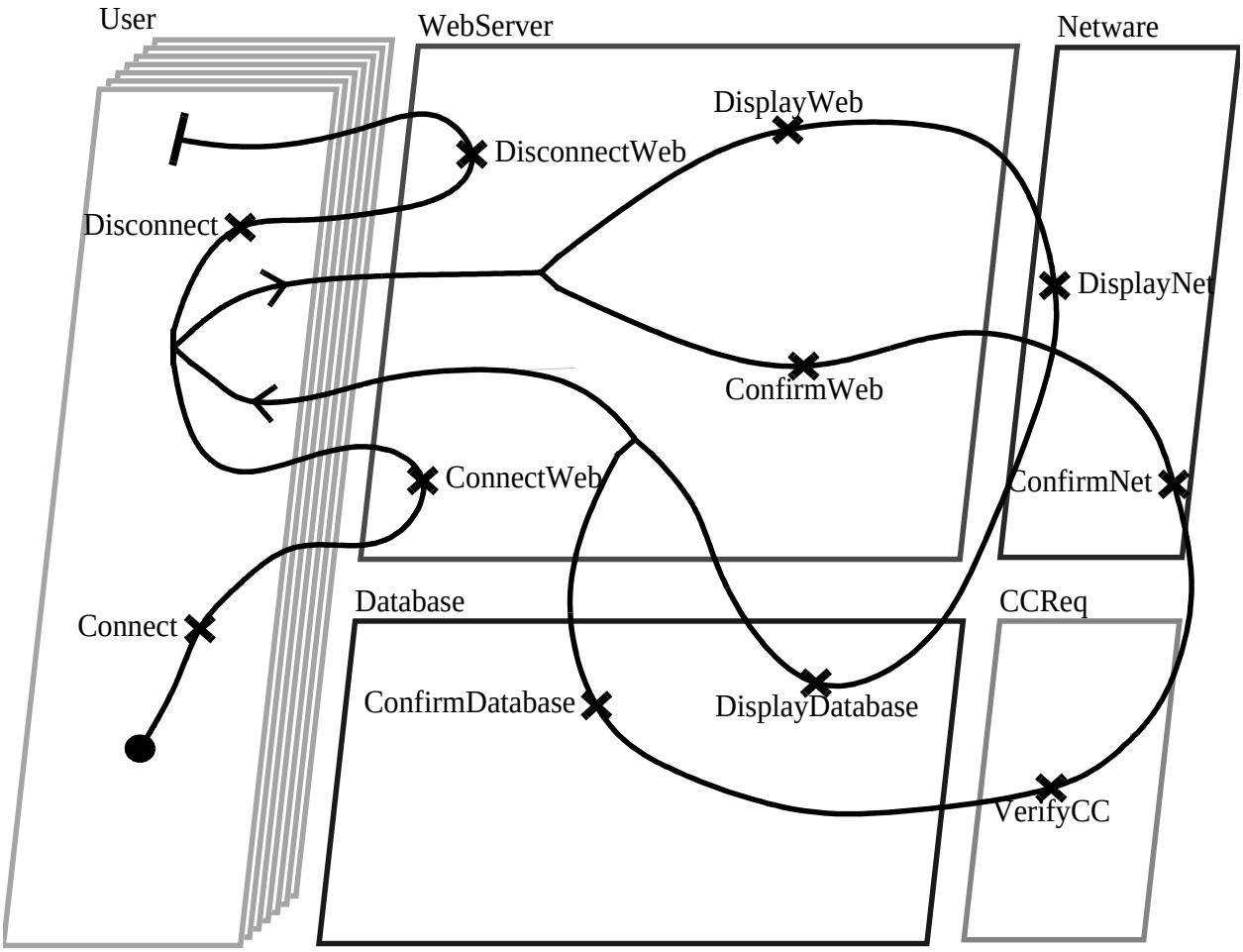


Figure 6-6: UCMNav map for the TRS example.

availability, or to buy tickets using a credit card. A typical scenario involves having the *User* log on to the system by requesting a connection. The *WebServer* then logs the user on and opens a session, then confirms that the connection was made. Once she is connected to the system, the *User* enters a loop where she has two options. She can either choose to browse and check information about an event, or she can buy a ticket. If the browsing option is chosen, the *WebServer* sends an event information request to the *Database,* through the *Netware.* The *Database* is responsible for retrieving the data requested and send it to the *WebServer.* The information can then be displayed back to the *User,* who can now choose whether to continue browsing, purchase tickets or disconnect. If the ticket purchasing option is chosen, the *User* must supply a credit card number to which the purchase price can be billed. The *WebServer* then begins to confirm the transaction by contact-

ing the *CCReq* through the *Netware* and requesting that the credit card be verified. Once the credit card is checks out, *CCReq* forwards the purchase request to the *Database* so it may update its records. The transaction is now completed and a confirmation is sent to the *WebServer*, which in turn relays it back to the *User*. The *User* may continue to browse or purchase more tickets as she wishes. Once the *User* is done, she can make a disconnection request and the *WebServer* closes the session and confirms that the she has been logged out.

6.2.3. TRS LQN Conversion Results

The TRS LQN is shown in Figure 6-7 as a jLqnDef graphic and in Figure 6-8 as a diagram showing the activity connections. The LQN shows that synchronous calls and forwarding are transformed properly. All of the interactions between the tasks in the TRS are of a synchronous nature, except for the initial asynchronous call from the reference task, due to the open nature of the model. The interesting feature of this example is that it requires the conversion of a complex loop, the body of which features forking and joining and makes service requests of other tasks. The loop head is shown as the activity *User_LH_48* in the *User* task. The loop body was abstracted away from the loop head and is represented by the *User_clone1_E1* entry in the *User_clone1* task. The rest of the loop body is taken care of by the activities in *User_clone1* and the call made from *User_clone1_A2* to the *WebServer_E2* entry in the *WebServer* task.

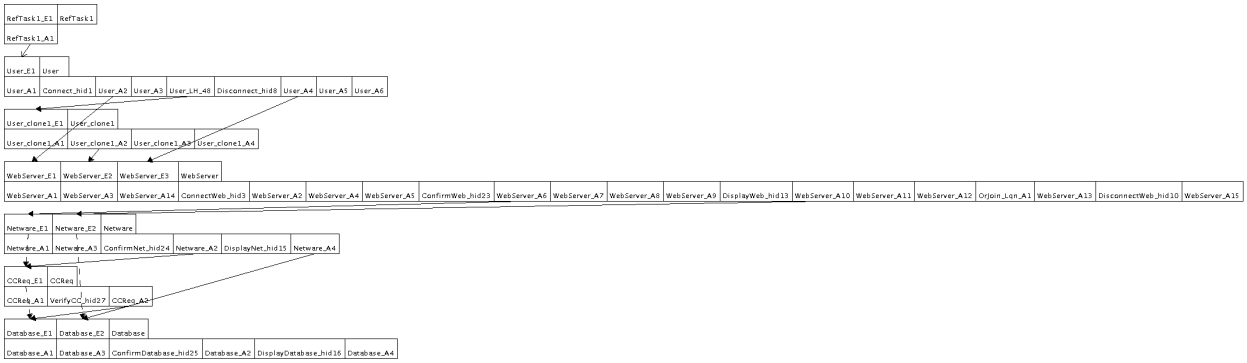


Figure 6-7: Ticket Reservation System LQN generated by the UCM2LQN converter from the UCM shown in Figure 6-6. (output from jLqnDef)

The resulting LQN for the TRS can be solved with LQNS as well as simulated with ParaS-

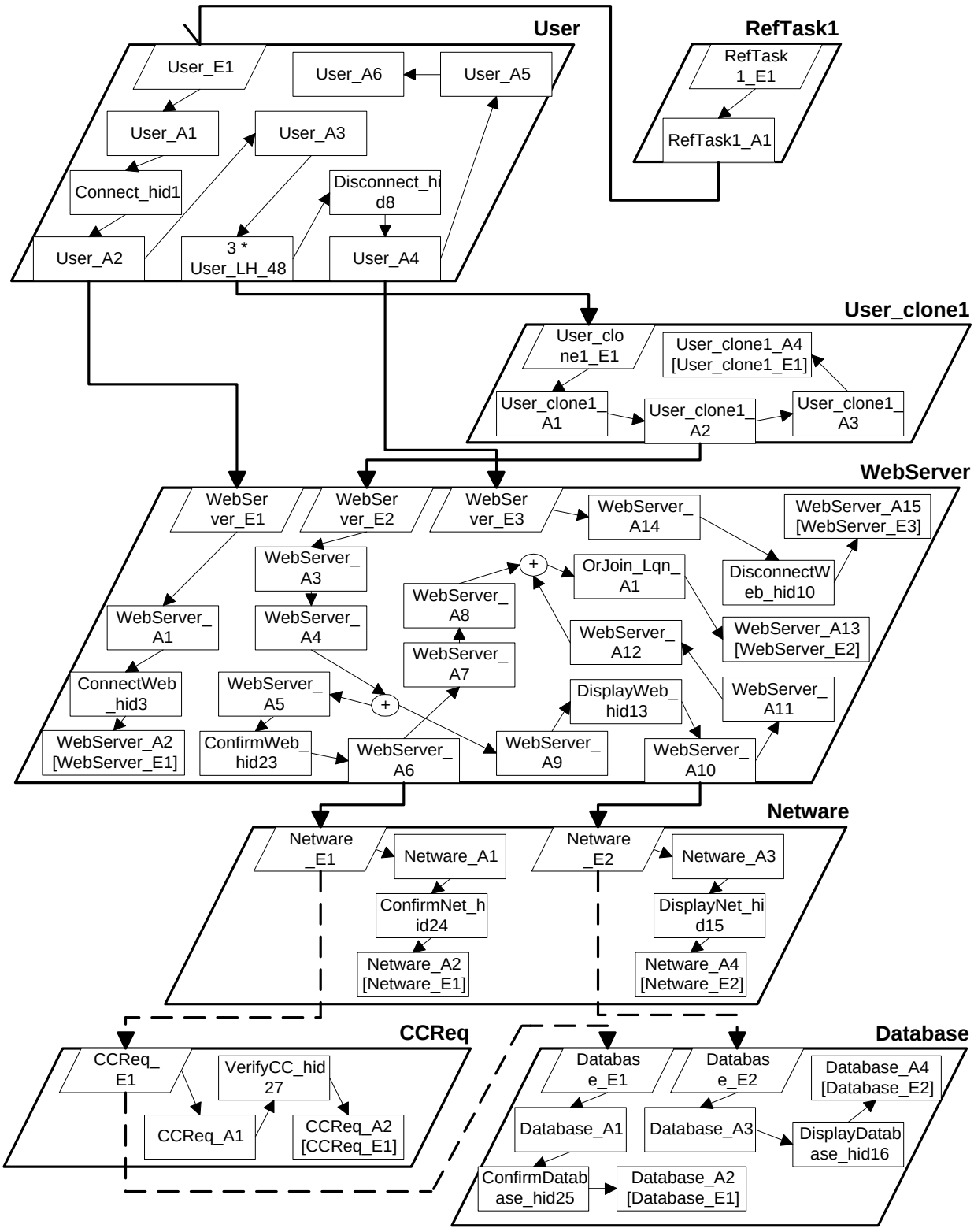


Figure 6-8: TRS LQN showing activity connections based on the output generated by the UCM2LQN converter from the UCM shown in Figure 6-6.

RVN. This shows that the UCM2LQN converter output is syntactically and semantically correct for both tools.

6.3. Group Communication Server

The Group Communication Server (GCS) design was developed by Craig Scratchley as part of his research on PERFECT, a methodology to evaluate the feasibility of implementing alternate software concurrency architectures.[40][41] The GCS is used to store documents and allow users to have later access to those documents. Users register with the GCS and can subscribe to certain sets of documents, submit new documents, or update documents. When a document is newly updated, the GCS then notifies its subscribed users in case they wish to request the latest version.[41]

The GCS presented in this section is part of a larger family of group communication systems that provide the same kind of subscription and multicast services to any type of client that has some type of computational capability. “[Such] groups [of computing entities] can be used in distributed computing systems to help master the complexity of large applications or to help provide non-functional properties, such as availability or security...The full benefits of the group concept, however, can be reaped only if we know how to set up and coordinate groups of processes that work together to fulfill a common purpose.”[37] Group communication systems are distributed by their very nature and as such can maintain the flow of communication even when some of their users no longer participate. Totem [30] and Hours [38] are two prime examples of group communication systems that help maintain fault-tolerant distributed computing groups.

6.3.1. GCS Design

Several GCS UCM designs based on different software architectures are available. The design chosen for this example is the one with the maximum amount of parallelism [40] - which translates into the design with the greatest number of tasks. This particular deployment was chosen because it will exercise the conversion algorithm the most by having the largest amount of communication. The UCM root map for the GCS is shown in Figure 6-9. The GCS components are the following:

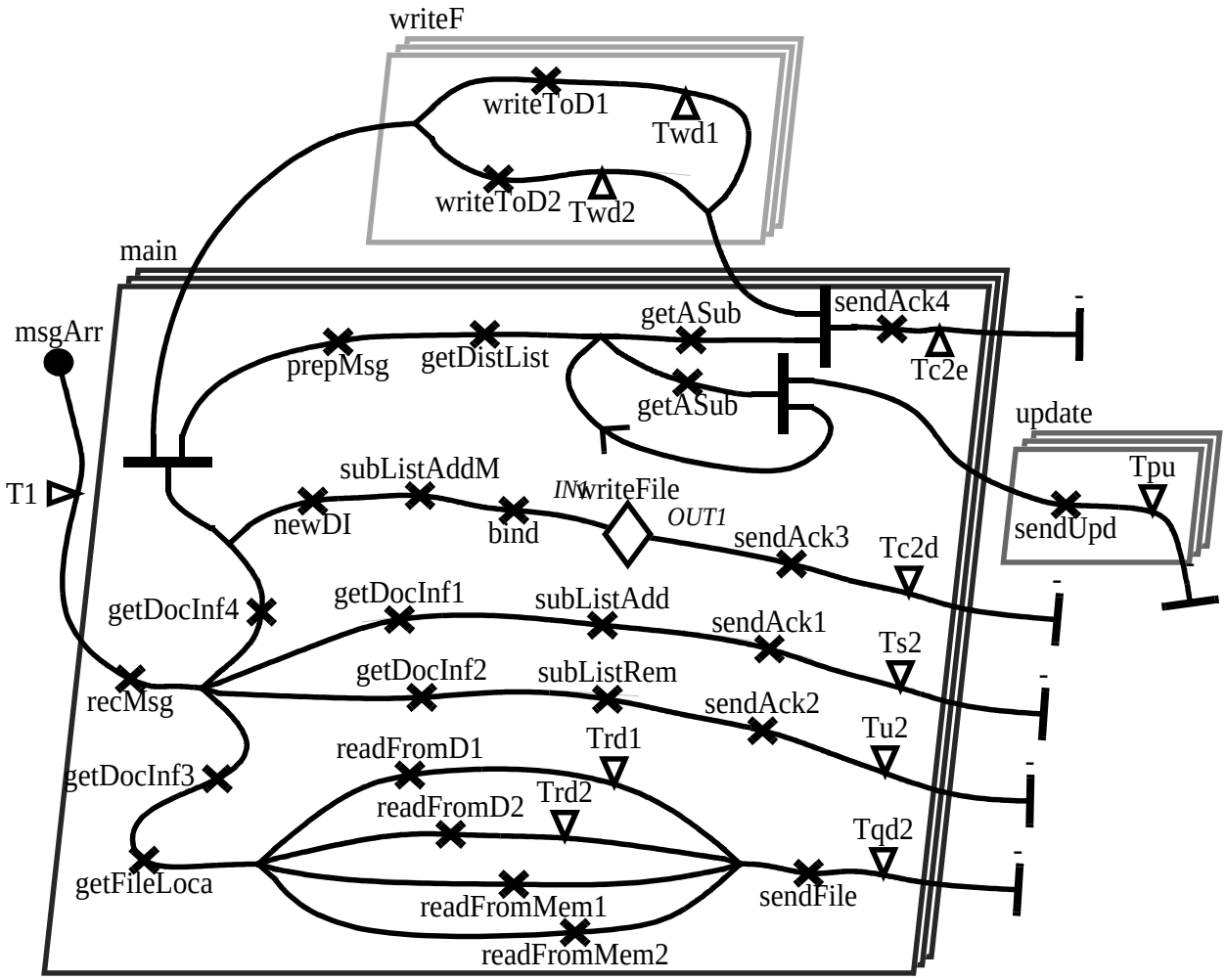


Figure 6-9: UCMNav root map for the GCS example.

- *main*: main GCS process
- *writeF*: process to write files to the disk
- *update*: process to send out document update notifications

The GCS root map also has a writeFile stub which is bound to the plug-in map shown in Figure 6-10.

6.3.2. GCS Usage

The GCS supports five usage scenarios [40] as follows:

- updating a document.

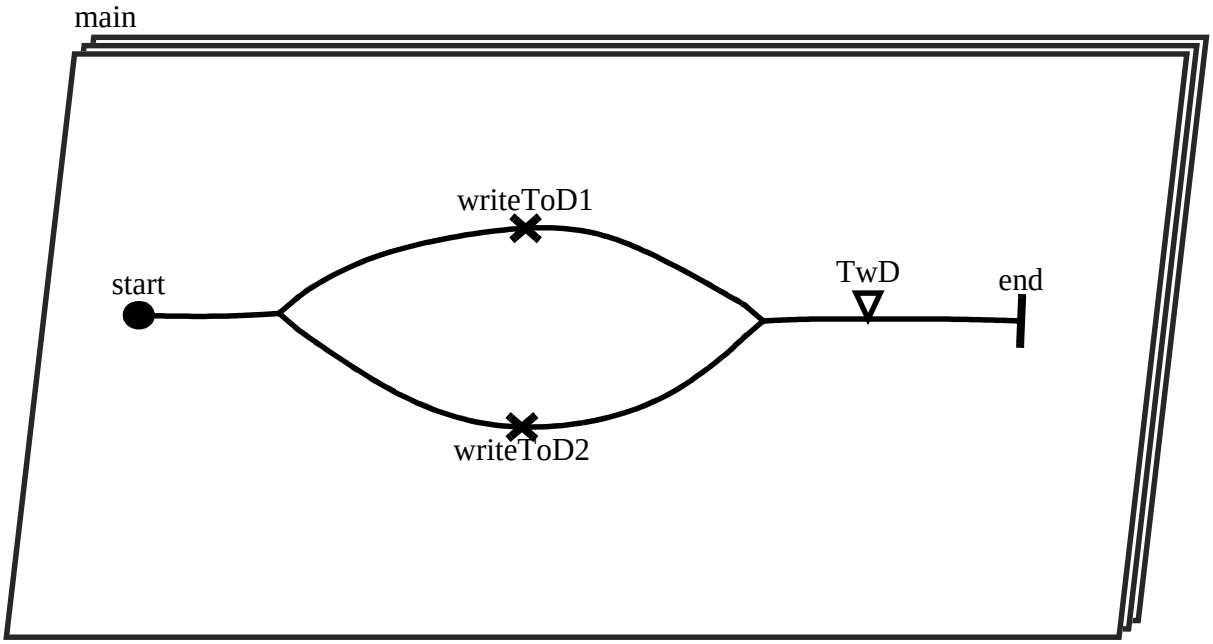


Figure 6-10: UCMNav plug-in map for the WriteFile stub shown in Figure 6-9 for the GCS example.

- submitting a new document.
- subscribing to a document
- unsubscribing from a document
- retrieving the most recent version of a document.

When updating a document, a user submits a new copy of a document that is already being stored on the server. The *main* task updates the information associated with the document and then sends it off to the *writeF* task for saving. The *writeF* task saves the document to one of the two disks. While the save is happening, the *main* task prepares a notification message and makes a temporary copy of the subscriber list for the document. The *main* task then proceeds to loop through the subscriber list and request that the *update* task dispatch a notification of the document update to each subscriber.

In the submitting a new document scenario, the user submits a new document to the *main* task. The *main* task creates a new subscriber list that is associated with the document and adds the user as its first subscriber. The document is added to the master index of the documents available on the server. The *main* task then saves the document to one of the two disks, as shown in Figure

6-10. Finally, an acknowledgment that the document has been accepted and saved is sent to the user.

If a user wants to subscribe to a document, the *main* task retrieves the subscriber list for the specified document, adds the user's name to the list, and then send an acknowledgment back. The unsubscribing from a document scenario unfolds the same way, except the user's name is removed from the subscriber list instead of being added to it.

In order to retrieve the most recent version of a document, the *main* task retrieves the information associated with it and then proceeds to retrieve the document. The document can either be found in one of the two memory caches, or on one of the two disks. After the document is read, the *main* task sends it out to the user.

6.3.3. GCS LQN Conversion Results

The UCM2LQN converter generates a large number of default activities in order to create activity connections and calling relationships that correspond to the UCM input. Every fork and join leads to the creation of as many default activities as there are path segments leading into or away from the given fork or join. Thus an OR fork with one input and three output branches will lead to the creation of four default activities to represent it. The GCS UCM design features six forks, four joins, and one loop. There are also 29 responsibilities in the system.

This results in an LQN model with a lot of activities. The *main* task alone contains 64 activities. The jLqnDef tool has a limited amount of space in its internal dictionary and the GCS LQN is too large to load into jLqnDef. Therefore, a graphical view cannot be generated for the resulting LQN. The LQN file for the GCS is included in Appendix C.

The GCS LQN can be solved by the LQNS analytic solver as well as by the ParaSRVN simulator. This demonstrates that the output from the UCM2LQN converter is both syntactically and semantically correct. In the absence of a visual representation of the resulting LQN model, the result files for the LQNS analysis and ParaSRVN simulation of the GCS have also been included.

Chapter 7 - Application

This chapter shows two industrial design specifications used to test the UCM2LQN converter. The examples are that of a Call Delivery for Wireless Intelligent Networks (WIN) and a Distributed Hand-off Protocol. The UCM models for these systems are based on industrial examples and provide interesting test results, since they do not necessarily conform to an “in-house” style of UCM specification.

7.1. WIN Call Delivery

The UCM root map for the Call Delivery system is shown in Figure 7-1 and the CallSetup plug-in map Figure 7-2 originates from research into Wireless Intelligent Networks (WIN) standards conducted in conjunction with a local telecommunications company.[3][27] This is an example of an industrial-style UCM design used to describe a real system.

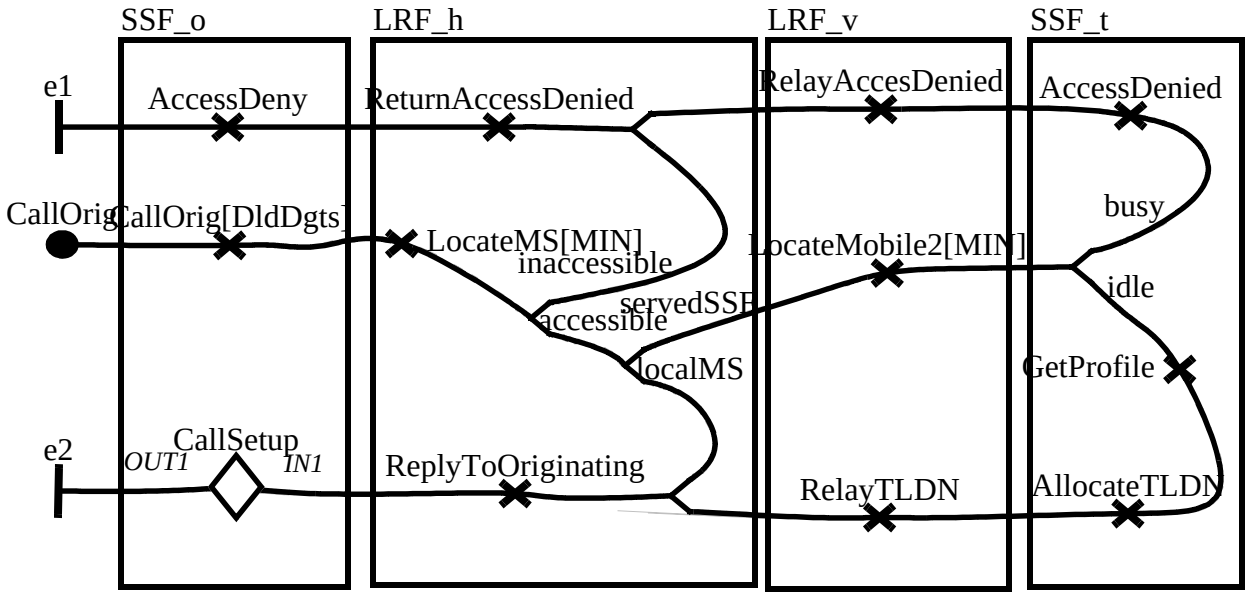


Figure 7-1: UCMNav root map for the WIN call delivery example.

The WIN example was not supplied with any additional documentation for this testing exercise. Therefore the call delivery scenario was inferred from the UCM design. Please note that a complete understanding of the details of this system is not and should not be required in order to

be able to generate a performance model. However, the basic scenario for this model as it emerges from the UCM can be summed up as a call delivery attempt begins when a call is originated by the calling side after dialing a number. As the scenario unfolds through the system, it generates two main outcomes: the call can either be set up or the caller is denied access to the callee. The *SSF_o* and *LRF_h* components represent the calling side of the system, and the *LRF_v* and *SSF_t* components represent the receiving end of the system.

If the call can be set up, then the plug-in for the *CallSetup* stub, shown in Figure 7-2, is traversed. The main path for the plug-in is the answer path and the answer end point is the one bound to the output of the stub in the root map. The plug-in does have an alternate time-out path where instead of receiving an answer after the call has been set up, the caller hears a recorded announcement explaining that a reply will not be forthcoming and releases the call.

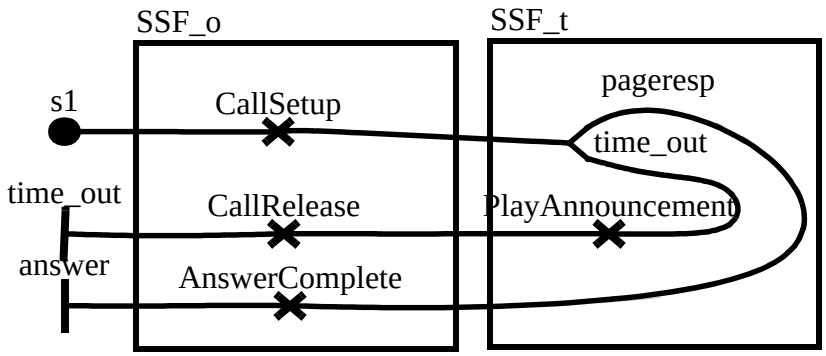


Figure 7-2: UCMNav plug-in map for the CallSetup stub shown in Figure 7-1 for the WIN call connection example.

7.1.1. WIN Call Delivery LQN Conversion Results

The first attempt at converting this example did not yield a result that could be read by any of the LQN tools (jLqnDef, LQNS, or ParaSRVN) because the original names of some of the activities included the character sequence ‘-1’, which is a protected end of field deliminator in the LQN file format. Although the LQN model that was generated was deemed to be valid after a manual inspection, the stray ‘-1’ character sequences made it unsuitable as an input for any of the tools. Thus the original WIN call delivery UCM that was submitted for testing had to be modified by removing the ‘-1’ from the names that had it. This incident did serve to illustrate how the “in-

house" style of the UCMs used for validation in Chapter 6 avoided exposing a possible weakness in the conversion results.

The jLqnDef output for the call delivery LQN is shown in Figure 7-3. The model shows both asynchronous and synchronous calling relationships between the tasks. The resulting model could only be solved by the ParaSRVN simulator. The LQNS analytic solver could not interpret the semantics of the OR forks and joins in different components.

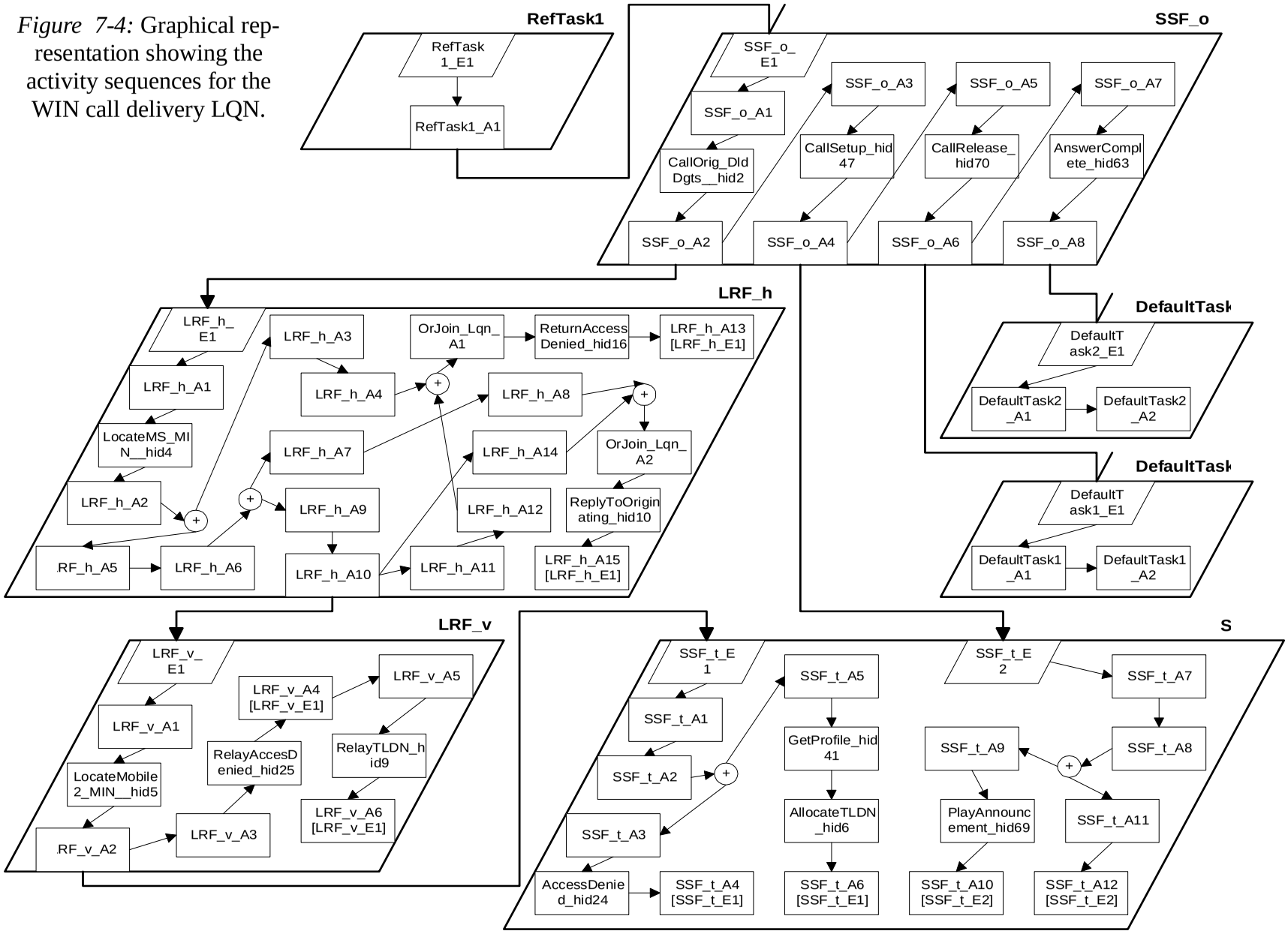
Figure 7-3: WIN call delivery LQN generated by the UCM2LQN converter from the UCM shown in Figure 7-1. (output from jLqnDef)

This model provides a good example of branching and joining, but the jLqnDef output does not show activity connections. The LQN model was thus manually redrawn based on the file output in a manner that shows the activity connections. The redrawn model is shown in Figure 7-4 and demonstrates that the UCM2LQN converted output does match the OR forking and joining structure of the original UCM.

7.2. Distributed Hand-Off Protocol

This system describes a distributed hand-off protocol. It is based on a teaching example developed by Gunther Mussbacher at Mitel Networks. As such, it is not a design document for a real product, but rather a theoretical model designed to showcase a particular style of drawing UCMs and to be used as a learning resource by designers who need to become familiar with the UCM notation. The UCM used for this test is shown in Figure 7-5. It was adapted from the industrial teaching example by Khalid H. Siddiqui as part of his M.Eng. research.

Figure 7-4: Graphical representation showing the activity sequences for the WIN call delivery LQN.



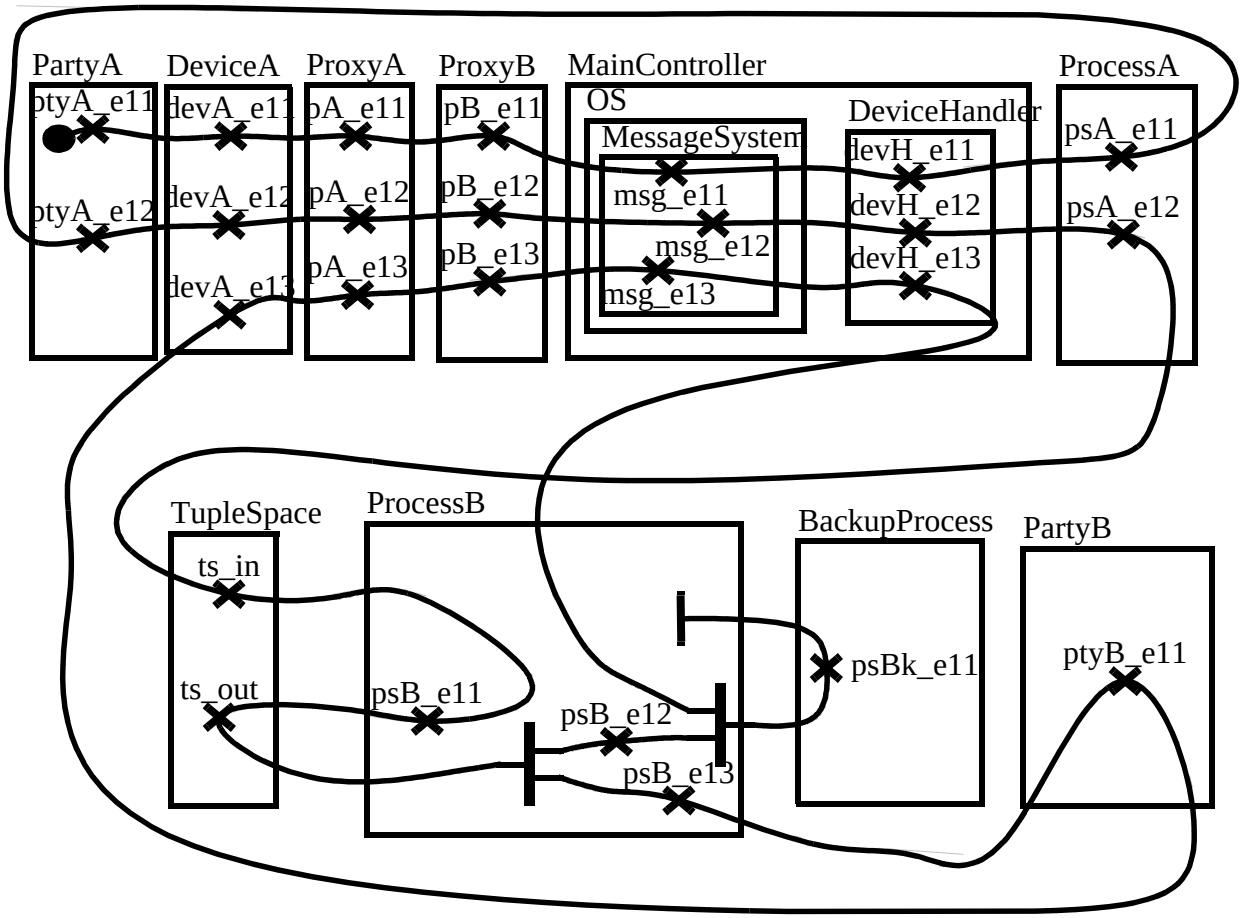


Figure 7-5: UCMNav root map for the distributed hand-off protocol example.

The protocol described in this specification is used to coordinate a hand-off of some processing task between two tasks, *PartyA* and *PartyB*, in a distributed environment. Each party has an associated device and proxy, and uses them to communicate through a distributed network. The *TupleSpace* process is based on the Linda shared memory tuple space paradigm.[47] A tuple space stores data as tuples, which are an arbitrary mix of actual and formal fields.[10][24] Linda provides four basic operations:

- out - inserts a tuple into the tuple space; the tuple becomes visible to all processes that have access to the tuple space
- in - extracts a tuple from the tuple space and returns it; the tuple is removed from the tuple space if it matches the argument provided with the *in* request

- rd - similar to *in*, except a copy of the tuple is returned and the tuple is not removed from the tuple space
- eval - similar to *out*, except separate processes are spawned to evaluate each of its fields [10][24]

The *TupleSpace* component in this example is used as a means to coordinate communication between the other tasks in the system.

The scenario illustrated in Figure 7-5 illustrates the transfer of the handling of some arbitrary duty from *PartyA* to *PartyB*. *PartyA* initiates the hand-off procedure, the initial phase of which passes through *DeviceA*, *ProxyA*, *ProxyB*, the *MessageSystem* and the *DeviceHandler* in the *MainController*, and *ProcessA* before returning to *PartyA*. The scenario then proceeds through the same set of tasks before arriving at the *TupleSpace*. The *TupleSpace* task performs an *in* operation and passes the tuple to *ProcessB*. *ProcessB* performs a responsibility psB_e11 and returns to the *TupleSpace*. The *TupleSpace* then performs an *out* operation and re-inserts the tuple back into the tuple space.

The scenario returns to *ProcessB* and the path splits into two parallel segments. One of the parallel segments traverses *PartyB*, *DeviceA*, *ProxyA*, *ProxyB*, the *MessageSystem* and *DeviceHandler* contained in the *MainController*, before returning to *ProcessB* and synchronizing with the other parallel segment that executed responsibility psB_e12 in *ProcessB*. Finally, a call is made to the *BackupProcess*.

7.2.1. Distributed Hand-Off Protocol LQN Conversion Results

In order to generate a UCM2LQN output file that could be read by the LQN tools, it was necessary to modify some of the names in the UCM so that they did not contain any spaces. The LQN file format uses a space as a delimiter between items, and as such a name with a space of any kind ends up being read as two different names. This generates syntax errors that make the output file unusable with any of the LQN tools. Just replacing the spaces with underscores was enough to solve this problem.

The jLqnDef graphical representation of the LQN model resulting from the hand-off protocol is shown in Figure 7-6. The LQN model shows the numerous asynchronous, synchronous,

and forwarding calls and replies that take place in the example.

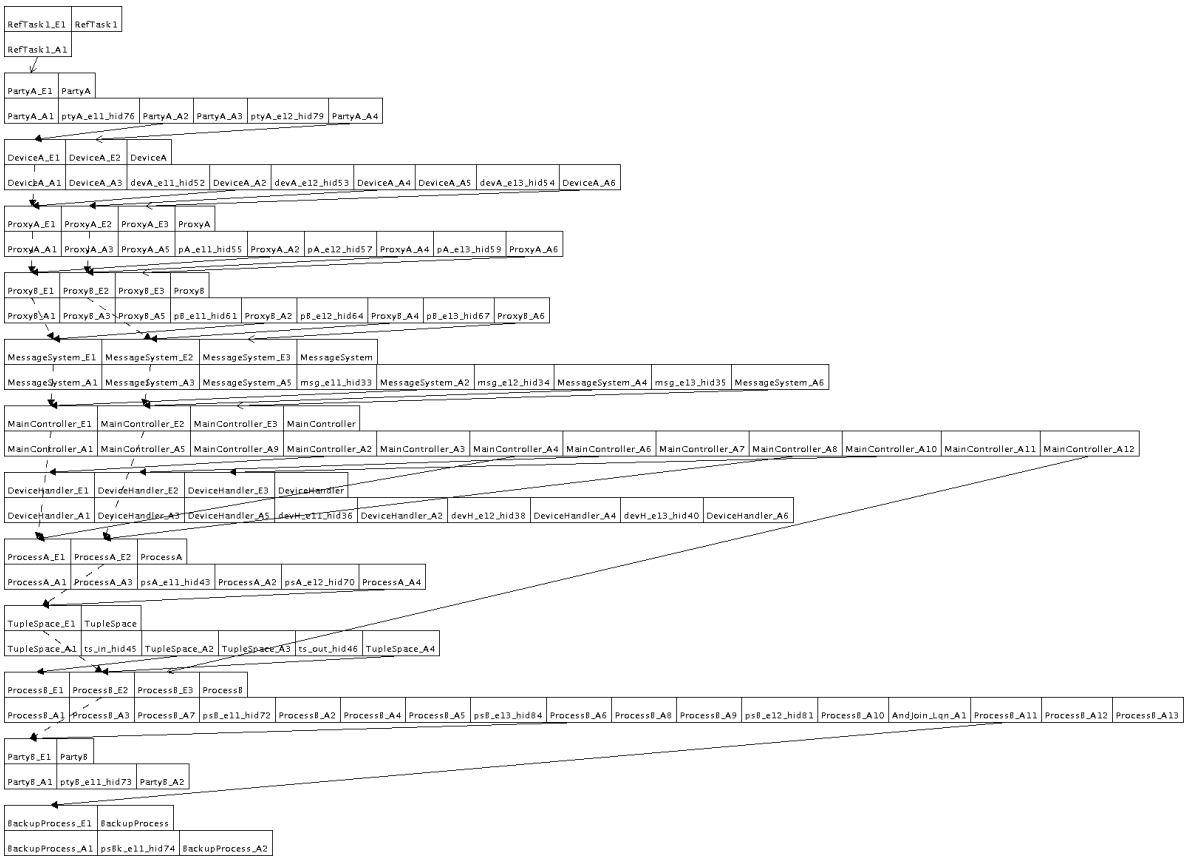


Figure 7-6: LQN for the distributed hand-off protocol generated by the UCM2LQN converter from the UCM shown in Figure 7-5. (output from jLqnDef)

An interesting result from the conversion is that the *OS* component does not have any entries or activities in the LQN model and is effectively removed from the functioning of the system, even though the UCM clearly shows the path going through the *OS*. This is explained by the fact that in the UCM model there are no path points enclosed exclusively in the *OS* component. This shows that in order for a component to be included in the converted model it must directly enclose at least one path point.

The model is solvable by both LQNS and ParaSRVN, which shows that the distributed hand-off protocol LQN generated by UCM2LQN is both syntactically and semantically correct.

Chapter 8 - Conclusions

8.1. Contributions

The overall contribution of this thesis is the development of a solution for integrating high level design and performance analysis at an early stage in the software development cycle. The UCM2LQN converter is the glue between high level design in the form of Use Case Maps and performance analysis using Layered Queueing Networks. The impact of this tool is further enhanced by its automated nature, as the converter is integrated with the UCM Navigator editing tool and the resulting output can be analysed using existing programs like the LQNS analytic solver and the ParaSRVN simulator.

The author has identified the basic corresponding constructs between the UCM and LQN notations, as well as documenting corresponding UCM and LQN models for certain patterns of interaction between components in a system. These correspondences were used as a basis for the development of an algorithm to convert UCM designs into LQN performance models. The conversion algorithm includes sections on traversing and parsing the internal data model of the UCMNav, detecting the crossing of component boundaries and interpreting the messaging nature of said crossings, and creating the appropriate LQN entities to correspond with the UCM path constructs and sequences.

This conversion algorithm has also been implemented in the UCM2LQN conversion tool. The author has also developed and implemented a design strategy for integrating the converter as an add-on module to the UCMNav.

8.2. Case Studies

The UCM2LQN converter was applied to five different case studies in order to validate the algorithm and test the tool. The validation was carried out by converting three UCM models for a Plain Old Telephone System, a Ticket Reservation System, and a Group Communications server. The resulting LQN models were viewed with jLqnDef and were then analysed and simulated using the LQNS and ParaSRVN tools respectively. The algorithm was validated as all three

example systems can be simulated without problem. The LQNs for the TRS and GCS systems can also be solved with the analytical solver, but the POTS model cannot due to a known limitation in the solver when it comes to dealing with distributed inter-task forks and joins. This points to an area of possible future research.

The converter was also tested successfully using models originating from industry. These examples were that of a call delivery in a Wireless Intelligent Network and a distributed hand-off protocol. Both of these models showed the importance of paying attention to the names of UCM objects, since certain names may employ restricted characters in the LQN file format. Both models converted to LQNs that can be simulated with ParaSRVN. The LQN for the hand-off protocol can also be solved with LQNS, whereas the call delivery model cannot be solved due to the same limitation of LQNS in solving distributed forks and joins.

8.3. Limitations

There are limitations to the conversion algorithm and the implementation of the UCM2LQN converter. The conversion algorithm does not currently cover the following UCM constructs:

- dynamic stubs
- waiting places
- timers and timeouts
- goals
- timestamps

The current implementation of the UCM2LQN converter does not yet handle corresponding forks and joins distributed across multiple components. AND forks and joins are handled in the same manner as OR forks and joins. This approach works well when corresponding forks and joins that occur in the same component. However, when corresponding forks and joins are distributed across several components ANDs and ORs must be handled differently.

8.4. Future Work

Further work should be done to address the limitations listed in Section 8.3. LQN conver-

sions will be defined for dynamic stubs, waiting places, timers and timeouts, goals, and timestamps. The call detection and LQN object creation algorithms will also be modified to properly deal with AND and OR forks and joins distributed across several components.

Additional research should also be carried out to identify and differentiate between UCM start points that represent scenario beginnings as opposed to UCM start points that are connected to end points along a path and represent sub-scenario beginnings. This will enable designers to describe open system arrival rates and closed system populations in a manner that makes the LQNs generated by the UCM2LQN converter more meaningful.

The user interface of the UCM2LQN converter should also be updated to be more interactive. The user should be given the option of specifying missing performance information or choosing to use the default values.

References

[1] A. Aho, S. Gallagher, N. Griffeth, C. Scheel, and D. Swayne, “Sculptor with Chisel: Requirements Engineering for Communications Services”, Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW’98), IOS Press, Amsterdam, October 1998, pp. 45–63.

[2] Daniel Amyot, “Use Case Maps for the Design and the Validation of Interaction-Free Telephony Features”, CITO report #1430, Ottawa, 1998

[3] D. Amyot and R. Andrade, “Description of Wireless Intelligent Network Services with Use Case Maps” 17th Brazilian Symposium on Computer Networks (SBRC'99), Salvador, Brazil, May 1999

[4] D. Amyot, L. Charfi, N. Gorse, T. Gray, L. Logrippo, J. Sincennes, B. Stepien, and T. Ware, “Feature Description and Feature Interaction Analysis with Use Case Maps and LOTOS”, Sixth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'00), Glasgow, Scotland, May 2000

[5] D. Amyot, L. Logrippo, and R.J.A. Buhr, “Spécification et conception de systèmes communicants: une approche rigoureuse basée sur des scénarios d’usage”, Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'97), Hermes, Paris, 1997, pp. 159-174

[6] D. Amyot, L. Logrippo, R.J.A. Buhr, and T. Gray, “Use Case Maps for the Capture and Validation of Distributed Systems Requirements”, Fourth International Symposium on Requirements Engineering (RE'99), Limerick, Ireland, June 1999

[7] D. Amyot and A. Miga, “XML DTD for Use Case Maps”, http://www.usecasemaps.org/UseCaseMaps/xml/ucm20xml.dtd, June 2000

[8] D. Amyot and G. Mussbacher, “On the Extension of UML with Use Case Maps Concepts”, The 3rd International Conference on the Unified Modeling Language (UML2000), York, UK, October 2000.

[9] S. Balsamo and M. Simeoni, “Deriving Performance Models from Software Architecture Specifications”, European Simulation Multiconference 2001 (ESM 2001), Prague, June 2001

[10] Gordon Blair, Nigel Davies, Adrian Friday and Stephen Wade, “Quality of Service Support in a Mobile Environment: An Approach Based on Tuple Spaces“, Proceedings of the 5th

IFIP International Workshop on Quality of Service (IWQoS '97), Columbia University, New York, May 1997, pp 37-48

[11] F. Bordeleau, “A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines”, Ph.D. thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1999

[12] F. Bordeleau and D. Cameron, “On the Relationship between Use Case Maps and Message Sequence Charts”, The 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000), Grenoble, France, June 2000

[13] F. Bordeleau, J.-P. Corriveau, and B. Selic, “A Scenario-Based Approach to Hierarchical State Machine Design”, The 3rd IEEE International Symposium on Object-Oriented Real-time distributed Computing (ISORC2000), Newport Beach, California, USA, March 2000

[14] F. Bordeleau and R. J. A. Buhr, “The UCM-ROOM Design Method: from Use Case Maps to Communicating State Machines”, Conference on the Engineering of Computer-Based Systems, Monterey, USA, March 1997

[15] R.J.A. Buhr, “Use Case Maps: A New Model to Bridge the Gap Between Requirements and Detailed Design”, OOPSLA'95 Real Time Workshop, Austin, October 1995

[16] R.J.A. Buhr, “Use Case Maps for Attributing Behaviour to System Architecture”, Fourth International Workshop on Parallel and Distributed Real Time Systems (WPDRTS), Honolulu, April 1996

[17] R.J.A. Buhr, “Making Behaviour a Concrete Architectural Concept”, 32nd Annual Hawaii International Conference on System Sciences (HICSS'99), Hawaii, USA, Jan 1999

[18] R.J.A. Buhr, D. Amyot, M. Elammari, D. Quesnel, T. Gray, and S. Mankovski, “Feature-Interaction Visualization and Resolution in an Agent Environment”, Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98), IOS Press, Amsterdam, Netherlands, pp. 135-149

[19] R.J.A. Buhr, M. Elammari, T. Gray, S. Mankovski, and D. Pinard, “Understanding and Defining the Behaviour of Systems of Agents with Use Case Maps”, Poster session at the Second Conference on Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM’97), London, 1997

[20] R.J.A. Buhr, M. Elammari, T. Gray, and S. Mankovski, “Applying Use Case Maps to Multi-agent Systems: A Feature Interaction Example”, 31st Annual Hawaii International Confer-

ence on System Sciences (HICSS'98), Hawaii, USA, Jan 1998

[21] R.J.A. Buhr and R.S. Casselman, “Use Case Maps for Object-Oriented Systems”, Prentice Hall, 1996

[22] D. Cameron et al., “Draft Specification of the User Requirements Notation”, Canadian Contribution CAN COM 10-12 to ITU-T Study Group 10, November 2000

[23] V. Cortellessa and R. Mirandola, “Deriving a Queueing Network-based Performance Model from UML Diagrams”, ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000, pp. 58-70

[24] N. Davies, S. Wade, A. Friday, and G. Blair, “Limbo: A Tuple Space Based Platform for Adaptive Mobile Applications”, Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97), Toronto, Canada, May 1997, pp 291-302.

[25] Greg Franks, “Performance Analysis of Distributed Server Systems”, Report OCIEE-00-01, Ph.D. thesis, Carleton University, Ottawa, Jan. 2000

[26] H. Gomaa and D. A. Menasce, “Design and Performance Modeling of Component Interconnection Patterns for Distributed Software Architectures”, ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000, pp. 117-126

[27] J. Hodges and J. Visser, “Accelerating Wireless Intelligent Network Standards Through Formal Techniques”, IEEE 1999 Vehicular Technology Conference (VTC’99), Houston, 1999

[28] A. Miga, “Application of Use Case Maps to System Design With Tool Support”, M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 1998

[29] O. Monkewich, “NEW QUESTION 12: URN: User Requirements Notation”, ITU-T Study Group 10, Temporary Document 99, November 1999

[30] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, “Totem: A Fault-Tolerant Multicast Group Communication System”, Communications of the ACM, Vol. 39, No. 4, April 1996, pp 54-63

[31] N. Griffeth, R. Blumenthal, J.-C. Gregoire and T. Ohta, “Feature Interaction Detection Contest Instructions”, 5th International Workshop on Feature Interactions, http://www.tts.lth.se/FIW98/contest.html, 1998

[32] E. Mascarenhas, “A System for Multithreaded Parallel Simulation and Computation with Migrant Threads and Objects”, Ph.D. Thesis, Department of Computer Sciences, Purdue University, West Lafayette, USA, 1996

[33] E. Mascarenhas, F. Knop, and V. Rego, “ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads”, Winter Simulation Conference, 1995

[34] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, “Software Bottlenecking in Client-Server Systems and Rendez-vous Networks”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995

[35] Dorin Petriu and C. M. Woodside, “Evaluating the Performance of Software Architectures”, The 5th Mitel Workshop (MICON2000), Mitel Networks, Ottawa, August 2000

[36] D. C. Petriu, C. Shousha, and A. Jalnapurkar, “Architecture-Based Performance Analysis Applied to a Telecommunication System”, IEEE Transactions on Software Engineering, Vol. 26, No. 11, Nov 2000, pp. 1049-1065

[37] D. Powell, “Introduction to Group Communication”, Communications of the ACM, Vol. 39, No. 4, April 1996, pp 50-53

[38] R. van Renesse, K. P. Birman, and S. Maffeis, “Horus: A Flexible Group Communication System”, Communications of the ACM, Vol. 39, No. 4, April 1996, pp 76-83

[39] J. A. Rolia, K. C. Sevcik, “The Method of Layers”, IEEE Transactions on Software Engineering, Vol. 21, No. 8, 1995, pp. 682-688

[40] Craig Scratchley, “Evaluation and Diagnosis of Concurrency Architectures”, Report OCIEE-00-07, Ph.D. thesis, Carleton University, Ottawa, Sept. 2000, pp 92-125

[41] C. Scratchley, C. M. Woodside, “Evaluating Concurrency Options in Software Specifications”, Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, Md., October 1999, pp 330 - 338

[42] K. H. Siddiqui, C. M. Woodside, “A description of Time/Performance Budgeting for UCM Designs”, The 5th Mitel Workshop (MICON2000), Mitel Networks, Ottawa, August 2000

[43] C. U. Smith, “Performance Engineering of Software Systems”, Addison-Wesley, 1990

[44] C. U. Smith, L.G. Williams, “Performance Engineering Evaluation of Object Oriented Systems with SPE-ED”, Computer Performance Evaluation: Modeling Techniques and Tools (LNCS 1245), Springer-Verlag, 1997

[45] C. U. Smith, Murray Woodside, “Performance Validation at Early Stages of Development”, Position paper, Performance 99, Istanbul, Turkey, October 99

[46] L. G. Williams, C. U. Smith, “Performance Evaluation of Software Architectures”, ACM Proceedings of Workshop on Software and Performance (WOSP’98), Santa Fe, NM, 1998, pp. 164-177

[47] D. W. Walker, “An Introduction to Message Passing Paradigms”, Proceedings of the 1995 CERN School of Computing (CERN 95-05), Arles, France, 1995, pp. 165-184

[48] C. M. Woodside, “Throughput Calculation for Basic Stochastic Rendezvous Networks”, Performance Evaluation, Vol. 9, No. 2, Apr 1988, pp. 143-160

[49] C. M. Woodside, “Performance-Oriented Patterns in Software Design (A Multi-Level Service Approach)”, Class notes, Carleton University, Ottawa, Sept. 1997, pp 35-67

[50] C. M. Woodside, J. E. Neilson, D. C. Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software”, IEEE Transactions on Computers, Vol. 44, No. 1, Jan 1995, pp. 20-34

[51] C. M. Woodside, S. Majumdar, J. E. Neilson, D. C. Petriu, J. A. Rolia, A. Hubbard and R. B. Franks, “A Guide to Performance Modeling of Distributed Client-Server Software Systems with Layered Queueing Networks”, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, Nov 1995

[52] ACM Proceedings of the Workshop on Software and Performance (WOSP’98), Santa Fe, USA, 1998

[53] ACM Proceedings of the Workshop on Software and Performance (WOSP2000), Ottawa, Canada, 2000

APPENDIX A - POTS LQN File

```
# UCM2LQN output

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
-1

T 0
t Switch f Switch_E1 Switch_E2 Switch_E3 -1  P_Infinite
t Orig f Orig_E1 -1  P_Infinite
t Term f Term_E1 Term_E2 -1  P_Infinite
# WARNING: task 'SCP' running on processor 'P_Infinite' (id -1) has no entries.

t OS f OS_E1 -1  P_Infinite
t RefTask1 r RefTask1_E1 -1  P_Infinite
-1

E 0
A Switch_E1 Switch_A1
A Switch_E2 Switch_A3
A Switch_E3 Switch_A17
A Orig_E1 Orig_A1
A Term_E1 Term_A1
A Term_E2 Term_A10
A OS_E1 OS_A1
A RefTask1_E1 RefTask1_A1
-1

A Switch
f Switch_A1 1
s Switch_A1 1.000000
f OrigBusy_hid20 1
s OrigBusy_hid20 1.000000
f Switch_A2 1
s Switch_A2 1.000000
f Switch_A3 1
s Switch_A3 1.000000
f Switch_A4 1
s Switch_A4 1.000000
f Switch_A5 1
s Switch_A5 1.000000
f Switch_A6 1
s Switch_A6 1.000000
f Switch_A7 1
s Switch_A7 1.000000
f TermBusy_hid40 1
s TermBusy_hid40 1.000000
f SetLI_hid35 1
```

```
s SetLI_hid35 1.000000
f Switch_A8 1
s Switch_A8 1.000000
y Switch_A8 Term_E1 1.000000
f Switch_A9 1
s Switch_A9 1.000000
f Switch_A10 1
s Switch_A10 1.000000
f Switch_A11 1
s Switch_A11 1.000000
f Switch_A12 1
s Switch_A12 1.000000
f Switch_A13 1
s Switch_A13 1.000000
f Switch_A14 1
s Switch_A14 1.000000
z Switch_A14 OS_E1 1.000000
f Switch_A15 1
s Switch_A15 1.000000
f Switch_A16 1
s Switch_A16 1.000000
f Switch_A17 1
s Switch_A17 1.000000
f Switch_A18 1
s Switch_A18 1.000000
f Switch_A19 1
s Switch_A19 1.000000
f Switch_A20 1
s Switch_A20 1.000000
z Switch_A20 Term_E2 1.000000

:
Switch_A1 -> OrigBusy_hid20;
OrigBusy_hid20 -> Switch_A2;
Switch_A2[Switch_E1];
Switch_A3 -> Switch_A4;
Switch_A4 -> (0.500000) Switch_A5 + (0.500000) Switch_A7;
Switch_A5 -> Switch_A6;
Switch_A6[Switch_E2];
Switch_A7 -> TermBusy_hid40;
TermBusy_hid40 -> SetLI_hid35;
SetLI_hid35 -> Switch_A8;
Switch_A8 -> Switch_A9;
Switch_A9 -> Switch_A10;
Switch_A10[Switch_E2];
Switch_A10 -> Switch_A11;
Switch_A11 -> Switch_A12;
Switch_A12 -> Switch_A13 & Switch_A15 & Switch_A19;
Switch_A13 -> Switch_A14;
Switch_A15 -> Switch_A16;
Switch_A16[Switch_E2];
Switch_A17 -> Switch_A18;
Switch_A18[Switch_E3];
Switch_A19 -> Switch_A20
-1

A Orig
f Orig_A1 1
s Orig_A1 1.000000
```

```
f Orig_A2 1
s Orig_A2 1.000000
y Orig_A2 Switch_E1 1.000000
f Orig_A3 1
s Orig_A3 1.000000
f DT_hid14 1
s DT_hid14 1.000000
f Dial_hid24 1
s Dial_hid24 1.000000
f StopDT_hid16 1
s StopDT_hid16 1.000000
f Orig_A4 1
s Orig_A4 1.000000
y Orig_A4 Switch_E2 1.000000
f Orig_A5 1
s Orig_A5 1.000000
f BT_hid17 1
s BT_hid17 1.000000
f Orig_A6 1
s Orig_A6 1.000000
f Orig_A7 1
s Orig_A7 1.000000
f RR_hid51 1
s RR_hid51 1.000000
f Orig_A8 1
s Orig_A8 1.000000
f Orig_A9 1
s Orig_A9 1.000000
f StopRR_hid69 1
s StopRR_hid69 1.000000
f Orig_A10 1
s Orig_A10 1.000000
y Orig_A10 Switch_E3 1.000000
f Orig_A11 1
s Orig_A11 1.000000
f Orig_A12 1
s Orig_A12 1.000000

:
Orig_A1 -> Orig_A2;
Orig_A2 -> Orig_A3;
Orig_A3 -> DT_hid14;
DT_hid14 -> Dial_hid24;
Dial_hid24 -> StopDT_hid16;
StopDT_hid16 -> Orig_A4;
Orig_A4 -> Orig_A5;
Orig_A5 -> BT_hid17;
BT_hid17 -> Orig_A6;
Orig_A6 -> Orig_A7;
Orig_A7 -> RR_hid51;
RR_hid51 -> Orig_A8;
Orig_A8 -> Orig_A9;
Orig_A9 -> StopRR_hid69;
StopRR_hid69 -> Orig_A10;
Orig_A10 -> Orig_A11;
Orig_A11 -> Orig_A12
-1

A Term
```

```
f Term_A1 1
s Term_A1 1.000000
f Term_A2 1
s Term_A2 1.000000
f Term_A3 1
s Term_A3 1.000000
f Term_A4 1
s Term_A4 1.000000
f Term_A5 1
s Term_A5 1.000000
f R_hid53 1
s R_hid53 1.000000
f Term_A6 1
s Term_A6 1.000000
f Term_A7 1
s Term_A7 1.000000
f Term_A8 1
s Term_A8 1.000000
f AndJoin_Lqn_A1 1
s AndJoin_Lqn_A1 1.000000
f OffHook_hid58 1
s OffHook_hid58 1.000000
f StopR_hid60 1
s StopR_hid60 1.000000
f Term_A9 1
s Term_A9 1.000000
f Term_A10 1
s Term_A10 1.000000
f Term_A11 1
s Term_A11 1.000000

:
Term_A1 -> Term_A2;
Term_A2 -> Term_A3 & Term_A5 & Term_A7;
Term_A3 -> Term_A4;
Term_A4[Term_E1];
Term_A5 -> R_hid53;
R_hid53 -> Term_A6;
Term_A7 -> Term_A8;
Term_A6 & Term_A8 -> AndJoin_Lqn_A1;
AndJoin_Lqn_A1 -> OffHook_hid58;
OffHook_hid58 -> StopR_hid60;
StopR_hid60 -> Term_A9;
Term_A9[Term_E1];
Term_A10 -> Term_A11
-1

A OS
f OS_A1 1
s OS_A1 1.000000
f LogBegin_ABA_t_hid33 1
s LogBegin_ABA_t_hid33 1.000000
f OS_A2 1
s OS_A2 1.000000

:
OS_A1 -> LogBegin_ABA_t_hid33;
LogBegin_ABA_t_hid33 -> OS_A2
-1
```

```
A RefTask1
f RefTask1_A1 1
s RefTask1_A1 1.000000
z RefTask1_A1 Orig_E1 1.000000

-1
```

APPENDIX B - TRS LQN File

```
# UCM2LQN output

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
p WebServer_Proc_pid5 f R 1.000000
p CCReq_Proc_pid6 f R 1.000000
p DB_Proc_pid9 f R 1.000000
p NetWare_Proc_pid10 f R 1.000000
p User_pid12 f R 1.000000
-1

T 0
t Database f Database_E1 Database_E2 -1  DB_Proc_pid9
t WebServer f WebServer_E1 WebServer_E2 WebServer_E3 -1  WebServer_Proc_pid5
t CCReq f CCReq_E1 -1  CCReq_Proc_pid6
t Netware f Netware_E1 Netware_E2 -1  NetWare_Proc_pid10
t User f User_E1 -1  User_pid12
t RefTask1 f RefTask1_E1 -1  P_Infinite
t User_clone1 f User_clone1_E1 -1  User_pid12
-1

E 0
A Database_E1 Database_A1
A Database_E2 Database_A3
A WebServer_E1 WebServer_A1
A WebServer_E2 WebServer_A3
A WebServer_E3 WebServer_A14
A CCReq_E1 CCReq_A1
F CCReq_E1 Database_E1 1.0 -1
A Netware_E1 Netware_A1
F Netware_E1 CCReq_E1 1.0 -1
A Netware_E2 Netware_A3
F Netware_E2 Database_E2 1.0 -1
A User_E1 User_A1
A RefTask1_E1 RefTask1_A1
a RefTask1_E1 0.001
A User_clone1_E1 User_clone1_A1
-1

A Database
f Database_A1 1
s Database_A1 1.000000
f ConfirmDatabase_hid25 1
s ConfirmDatabase_hid25 1.000000
f Database_A2 1
s Database_A2 1.000000
f Database_A3 1
```

```
s Database_A3 1.000000
f DisplayDatabase_hid16 1
s DisplayDatabase_hid16 1.000000
f Database_A4 1
s Database_A4 1.000000

:
Database_A1 -> ConfirmDatabase_hid25;
ConfirmDatabase_hid25 -> Database_A2;
Database_A2[Database_E1];
Database_A3 -> DisplayDatabase_hid16;
DisplayDatabase_hid16 -> Database_A4;
Database_A4[Database_E2]
-1

A WebServer
f WebServer_A1 1
s WebServer_A1 1.000000
f ConnectWeb_hid3 1
s ConnectWeb_hid3 1.000000
f WebServer_A2 1
s WebServer_A2 1.000000
f WebServer_A3 1
s WebServer_A3 1.000000
f WebServer_A4 1
s WebServer_A4 1.000000
f WebServer_A5 1
s WebServer_A5 1.000000
f ConfirmWeb_hid23 1
s ConfirmWeb_hid23 1.000000
f WebServer_A6 1
s WebServer_A6 1.000000
y WebServer_A6 Netware_E1 1.000000
f WebServer_A7 1
s WebServer_A7 1.000000
f WebServer_A8 1
s WebServer_A8 1.000000
f WebServer_A9 1
s WebServer_A9 1.000000
f DisplayWeb_hid13 1
s DisplayWeb_hid13 1.000000
f WebServer_A10 1
s WebServer_A10 1.000000
y WebServer_A10 Netware_E2 1.000000
f WebServer_A11 1
s WebServer_A11 1.000000
f WebServer_A12 1
s WebServer_A12 1.000000
f OrJoin_Lqn_A1 1
s OrJoin_Lqn_A1 1.000000
f WebServer_A13 1
s WebServer_A13 1.000000
f WebServer_A14 1
s WebServer_A14 1.000000
f DisconnectWeb_hid10 1
s DisconnectWeb_hid10 1.000000
f WebServer_A15 1
s WebServer_A15 1.000000
```

```
:
WebServer_A1 -> ConnectWeb_hid3;
ConnectWeb_hid3 -> WebServer_A2;
WebServer_A2[WebServer_E1];
WebServer_A3 -> WebServer_A4;
WebServer_A4 -> (0.500000) WebServer_A5 + (0.500000) WebServer_A9;
WebServer_A5 -> ConfirmWeb_hid23;
ConfirmWeb_hid23 -> WebServer_A6;
WebServer_A6 -> WebServer_A7;
WebServer_A7 -> WebServer_A8;
WebServer_A9 -> DisplayWeb_hid13;
DisplayWeb_hid13 -> WebServer_A10;
WebServer_A10 -> WebServer_A11;
WebServer_A11 -> WebServer_A12;
WebServer_A8 + WebServer_A12 -> OrJoin_Lqn_A1;
OrJoin_Lqn_A1 -> WebServer_A13;
WebServer_A13[WebServer_E2];
WebServer_A14 -> DisconnectWeb_hid10;
DisconnectWeb_hid10 -> WebServer_A15;
WebServer_A15[WebServer_E3]
-1

A CCReq
f CCReq_A1 1
s CCReq_A1 1.000000
f VerifyCC_hid27 1
s VerifyCC_hid27 1.000000
f CCReq_A2 1
s CCReq_A2 1.000000
y CCReq_A2 Database_E1 1.000000

:
CCReq_A1 -> VerifyCC_hid27;
VerifyCC_hid27 -> CCReq_A2;
CCReq_A2[CCReq_E1]
-1

A Netware
f Netware_A1 1
s Netware_A1 1.000000
f ConfirmNet_hid24 1
s ConfirmNet_hid24 1.000000
f Netware_A2 1
s Netware_A2 1.000000
y Netware_A2 CCReq_E1 1.000000
f Netware_A3 1
s Netware_A3 1.000000
f DisplayNet_hid15 1
s DisplayNet_hid15 1.000000
f Netware_A4 1
s Netware_A4 1.000000
y Netware_A4 Database_E2 1.000000

:
Netware_A1 -> ConfirmNet_hid24;
ConfirmNet_hid24 -> Netware_A2;
Netware_A2[Netware_E1];
Netware_A3 -> DisplayNet_hid15;
DisplayNet_hid15 -> Netware_A4;
```

```
Netware_A4[Netware_E2]
-1

A User
f User_A1 1
s User_A1 1.000000
f Connect_hid1 1
s Connect_hid1 1.000000
f User_A2 1
s User_A2 1.000000
y User_A2 WebServer_E1 1.000000
f User_A3 1
s User_A3 1.000000
f User_LH_48 1
s User_LH_48 1.000000
y User_LH_48 User_clone1_E1 3.000000
f Disconnect_hid8 1
s Disconnect_hid8 1.000000
f User_A4 1
s User_A4 1.000000
y User_A4 WebServer_E3 1.000000
f User_A5 1
s User_A5 1.000000
f User_A6 1
s User_A6 1.000000

:
User_A1 -> Connect_hid1;
Connect_hid1 -> User_A2;
User_A2 -> User_A3;
User_A3 -> 3.000000 * User_LH_48, Disconnect_hid8;
Disconnect_hid8 -> User_A4;
User_A4 -> User_A5;
User_A5 -> User_A6
-1

A RefTask1
f RefTask1_A1 1
s RefTask1_A1 1.000000
z RefTask1_A1 User_E1 1.000000

-1

A User_clone1
f User_clone1_A1 1
s User_clone1_A1 1.000000
f User_clone1_A2 1
s User_clone1_A2 1.000000
y User_clone1_A2 WebServer_E2 1.000000
f User_clone1_A3 1
s User_clone1_A3 1.000000
f User_clone1_A4 1
s User_clone1_A4 1.000000

:
User_clone1_A1 -> User_clone1_A2;
User_clone1_A2 -> User_clone1_A3;
User_clone1_A3 -> User_clone1_A4;
User_clone1_A4[User_clone1_E1]
```

```
-1
```

APPENDIX C - GCS LQN File

```
# UCM2LQN output

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
p pu1_pid3 f R 1.000000
-1

T 0
t writeF f writeF_E1 -1  P_Infinite
t update f update_E1 -1  P_Infinite
t main f main_E1 -1  pu1_pid3
t RefTask1 f RefTask1_E1 -1  P_Infinite
t DefaultTask1 f DefaultTask1_E1 -1  P_Infinite
t DefaultTask2 f DefaultTask2_E1 -1  P_Infinite
t DefaultTask3 f DefaultTask3_E1 -1  P_Infinite
t DefaultTask4 f DefaultTask4_E1 -1  P_Infinite
t DefaultTask5 f DefaultTask5_E1 -1  P_Infinite
t main_clone1 f main_clone1_E1 -1  pu1_pid3
t DefaultTask6 f DefaultTask6_E1 -1  P_Infinite
-1

E 0
A writeF_E1 writeF_A1
A update_E1 update_A1
A main_E1 main_A1
A RefTask1_E1 RefTask1_A1
a RefTask1_E1 0.00001
A DefaultTask1_E1 DefaultTask1_A1
A DefaultTask2_E1 DefaultTask2_A1
A DefaultTask3_E1 DefaultTask3_A1
A DefaultTask4_E1 DefaultTask4_A1
A DefaultTask5_E1 DefaultTask5_A1
A main_clone1_E1 main_clone1_A1
A DefaultTask6_E1 DefaultTask6_A1
-1

A writeF
f writeF_A1 1
s writeF_A1 1.000000
f writeF_A2 1
s writeF_A2 1.000000
f writeF_A3 1
s writeF_A3 1.000000
f writeToD2_hid8 1
s writeToD2_hid8 1.000000
f writeF_A4 1
s writeF_A4 1.000000
```

```
f writeF_A5 1
s writeF_A5 1.000000
f writeToD1_hid7 1
s writeToD1_hid7 1.000000
f writeF_A6 1
s writeF_A6 1.000000
f OrJoin_Lqn_A3 1
s OrJoin_Lqn_A3 1.000000
f writeF_A7 1
s writeF_A7 1.000000

:
writeF_A1 -> writeF_A2;
writeF_A2 -> (0.500000) writeF_A3 + (0.500000) writeF_A5;
writeF_A3 -> writeToD2_hid8;
writeToD2_hid8 -> writeF_A4;
writeF_A5 -> writeToD1_hid7;
writeToD1_hid7 -> writeF_A6;
writeF_A4 + writeF_A6 -> OrJoin_Lqn_A3;
OrJoin_Lqn_A3 -> writeF_A7;
writeF_A7[writeF_E1]
-1

A update
f update_A1 1
s update_A1 1.000000
f sendUpd_hid25 1
s sendUpd_hid25 1.000000
f update_A2 1
s update_A2 1.000000
z update_A2 DefaultTask6_E1 1.000000

:
update_A1 -> sendUpd_hid25;
sendUpd_hid25 -> update_A2
-1

A main
f main_A1 1
s main_A1 1.000000
f recMsg_hid16 1
s recMsg_hid16 1.000000
f main_A2 1
s main_A2 1.000000
f main_A3 1
s main_A3 1.000000
f getDocInf1_hid0 1
s getDocInf1_hid0 1.000000
f subListAdd_hid1 1
s subListAdd_hid1 1.000000
f sendAck1_hid2 1
s sendAck1_hid2 1.000000
f main_A4 1
s main_A4 1.000000
z main_A4 DefaultTask1_E1 1.000000
f main_A5 1
s main_A5 1.000000
f getDocInf2_hid5 1
s getDocInf2_hid5 1.000000
```

```
f subListRem_hid3 1
s subListRem_hid3 1.000000
f sendAck2_hid4 1
s sendAck2_hid4 1.000000
f main_A6 1
s main_A6 1.000000
z main_A6 DefaultTask2_E1 1.000000
f main_A7 1
s main_A7 1.000000
f getDocInf3_hid12 1
s getDocInf3_hid12 1.000000
f getFileLoca_hid13 1
s getFileLoca_hid13 1.000000
f main_A8 1
s main_A8 1.000000
f main_A9 1
s main_A9 1.000000
f readFromD2_hid10 1
s readFromD2_hid10 1.000000
f main_A10 1
s main_A10 1.000000
f main_A11 1
s main_A11 1.000000
f readFromMem1_hid15 1
s readFromMem1_hid15 1.000000
f main_A12 1
s main_A12 1.000000
f main_A13 1
s main_A13 1.000000
f readFromMem2_hid24 1
s readFromMem2_hid24 1.000000
f main_A14 1
s main_A14 1.000000
f main_A15 1
s main_A15 1.000000
f readFromD1_hid9 1
s readFromD1_hid9 1.000000
f main_A16 1
s main_A16 1.000000
f OrJoin_Lqn_A1 1
s OrJoin_Lqn_A1 1.000000
f sendFile_hid11 1
s sendFile_hid11 1.000000
f main_A17 1
s main_A17 1.000000
z main_A17 DefaultTask3_E1 1.000000
f main_A18 1
s main_A18 1.000000
f getDocInf4_hid14 1
s getDocInf4_hid14 1.000000
f main_A19 1
s main_A19 1.000000
f main_A20 1
s main_A20 1.000000
f newDI_hid17 1
s newDI_hid17 1.000000
f subListAddM_hid19 1
s subListAddM_hid19 1.000000
f bind_hid18 1
```

```
s bind_hid18 1.000000
f main_A21 1
s main_A21 1.000000
f main_A22 1
s main_A22 1.000000
f writeToD2_hid135 1
s writeToD2_hid135 1.000000
f main_A23 1
s main_A23 1.000000
f main_A24 1
s main_A24 1.000000
f writeToD1_hid132 1
s writeToD1_hid132 1.000000
f main_A25 1
s main_A25 1.000000
f OrJoin_Lqn_A2 1
s OrJoin_Lqn_A2 1.000000
f sendAck3_hid20 1
s sendAck3_hid20 1.000000
f main_A26 1
s main_A26 1.000000
z main_A26 DefaultTask4_E1 1.000000
f main_A27 1
s main_A27 1.000000
f main_A28 1
s main_A28 1.000000
f main_A29 1
s main_A29 1.000000
f main_A30 1
s main_A30 1.000000
y main_A30 writeF_E1 1.000000
f main_A31 1
s main_A31 1.000000
f main_A32 1
s main_A32 1.000000
f AndJoin_Lqn_A4 1
s AndJoin_Lqn_A4 1.000000
f sendAck4_hid23 1
s sendAck4_hid23 1.000000
f main_A33 1
s main_A33 1.000000
z main_A33 DefaultTask5_E1 1.000000
f main_A34 1
s main_A34 1.000000
f prepMsg_hid22 1
s prepMsg_hid22 1.000000
f getDistList_hid6 1
s getDistList_hid6 1.000000
f main_LH_121 1
s main_LH_121 1.000000
y main_LH_121 main_clone1_E1 3.000000
f getASub_hid21 1
s getASub_hid21 1.000000
f main_A35 1
s main_A35 1.000000

:
main_A1 -> recMsg_hid16;
recMsg_hid16 -> main_A2;
```

```
main_A2 -> (0.250000) main_A3 + (0.250000) main_A5 + (0.250000) main_A7 + (0.250000)
main_A18;
main_A3 -> getDocInf1_hid0;
getDocInf1_hid0 -> subListAdd_hid1;
subListAdd_hid1 -> sendAck1_hid2;
sendAck1_hid2 -> main_A4;
main_A5 -> getDocInf2_hid5;
getDocInf2_hid5 -> subListRem_hid3;
subListRem_hid3 -> sendAck2_hid4;
sendAck2_hid4 -> main_A6;
main_A7 -> getDocInf3_hid12;
getDocInf3_hid12 -> getFileLoca_hid13;
getFileLoca_hid13 -> main_A8;
main_A8 -> (0.250000) main_A9 + (0.250000) main_A11 + (0.250000) main_A13 + (0.250000)
main_A15;
main_A9 -> readFromD2_hid10;
readFromD2_hid10 -> main_A10;
main_A11 -> readFromMem1_hid15;
readFromMem1_hid15 -> main_A12;
main_A13 -> readFromMem2_hid24;
readFromMem2_hid24 -> main_A14;
main_A15 -> readFromD1_hid9;
readFromD1_hid9 -> main_A16;
main_A10 + main_A12 + main_A14 + main_A16 -> OrJoin_Lqn_A1;
OrJoin_Lqn_A1 -> sendFile_hid11;
sendFile_hid11 -> main_A17;
main_A18 -> getDocInf4_hid14;
getDocInf4_hid14 -> main_A19;
main_A19 -> (0.500000) main_A20 + (0.500000) main_A27;
main_A20 -> newDI_hid17;
newDI_hid17 -> subListAddM_hid19;
subListAddM_hid19 -> bind_hid18;
bind_hid18 -> main_A21;
main_A21 -> (0.500000) main_A22 + (0.500000) main_A24;
main_A22 -> writeToD2_hid135;
writeToD2_hid135 -> main_A23;
main_A24 -> writeToD1_hid132;
writeToD1_hid132 -> main_A25;
main_A23 + main_A25 -> OrJoin_Lqn_A2;
OrJoin_Lqn_A2 -> sendAck3_hid20;
sendAck3_hid20 -> main_A26;
main_A27 -> main_A28;
main_A28 -> main_A29 & main_A34;
main_A29 -> main_A30;
main_A30 -> main_A31;
main_A31 -> main_A32;
main_A32 & main_A35 -> AndJoin_Lqn_A4;
AndJoin_Lqn_A4 -> sendAck4_hid23;
sendAck4_hid23 -> main_A33;
main_A34 -> prepMsg_hid22;
prepMsg_hid22 -> getDistList_hid6;
getDistList_hid6 -> 3.000000 * main_LH_121, getASub_hid21;
getASub_hid21 -> main_A35
-1

A RefTask1
f RefTask1_A1 1
s RefTask1_A1 1.000000
z RefTask1_A1 main_E1 1.000000 # WARNING: unresolved call type, assuming asynchronous
```

```
-1

A DefaultTask1
f DefaultTask1_A1 1
s DefaultTask1_A1 1.000000
f DefaultTask1_A2 1
s DefaultTask1_A2 1.000000

:
DefaultTask1_A1 -> DefaultTask1_A2
-1

A DefaultTask2
f DefaultTask2_A1 1
s DefaultTask2_A1 1.000000
f DefaultTask2_A2 1
s DefaultTask2_A2 1.000000

:
DefaultTask2_A1 -> DefaultTask2_A2
-1

A DefaultTask3
f DefaultTask3_A1 1
s DefaultTask3_A1 1.000000
f DefaultTask3_A2 1
s DefaultTask3_A2 1.000000

:
DefaultTask3_A1 -> DefaultTask3_A2
-1

A DefaultTask4
f DefaultTask4_A1 1
s DefaultTask4_A1 1.000000
f DefaultTask4_A2 1
s DefaultTask4_A2 1.000000

:
DefaultTask4_A1 -> DefaultTask4_A2
-1

A DefaultTask5
f DefaultTask5_A1 1
s DefaultTask5_A1 1.000000
f DefaultTask5_A2 1
s DefaultTask5_A2 1.000000

:
DefaultTask5_A1 -> DefaultTask5_A2
-1

A main_clone1
f main_clone1_A1 1
s main_clone1_A1 1.000000
f getASub_hid123 1
s getASub_hid123 1.000000
f main_clone1_A2 1
```

```
s main_clone1_A2 1.000000
f main_clone1_A3 1
s main_clone1_A3 1.000000
f main_clone1_A4 1
s main_clone1_A4 1.000000
z main_clone1_A4 update_E1 1.000000
f main_clone1_A5 1
s main_clone1_A5 1.000000
f main_clone1_A6 1
s main_clone1_A6 1.000000

:
main_clone1_A1 -> getASub_hid123;
getASub_hid123 -> main_clone1_A2;
main_clone1_A2 -> main_clone1_A3 & main_clone1_A5;
main_clone1_A3 -> main_clone1_A4;
main_clone1_A5 -> main_clone1_A6;
main_clone1_A6[main_clone1_E1]
-1

A DefaultTask6
f DefaultTask6_A1 1
s DefaultTask6_A1 1.000000
f DefaultTask6_A2 1
s DefaultTask6_A2 1.000000

:
DefaultTask6_A1 -> DefaultTask6_A2
-1
```

APPENDIX D - WIN Call Delivery LQN File

```
G
""
1.0E-5
50
5
0.9
-1

P 1
p P_Infinite f  i
-1

T 7
t SSF_o f SSF_o_E1  -1 P_Infinite
t LRF_h f LRF_h_E1  -1 P_Infinite
t LRF_v f LRF_v_E1  -1 P_Infinite
t SSF_t f SSF_t_E1 SSF_t_E2  -1 P_Infinite
t RefTask1 r RefTask1_E1  -1 P_Infinite
t DefaultTask1 f DefaultTask1_E1  -1 P_Infinite
t DefaultTask2 f DefaultTask2_E1  -1 P_Infinite
-1

E 8
A SSF_o_E1 SSF_o_A1
A LRF_h_E1 LRF_h_A1
A LRF_v_E1 LRF_v_A1
A SSF_t_E1 SSF_t_A1
A SSF_t_E2 SSF_t_A7
A RefTask1_E1 RefTask1_A1
A DefaultTask1_E1 DefaultTask1_A1
A DefaultTask2_E1 DefaultTask2_A1
-1

A SSF_o
f SSF_o_A1 1
s SSF_o_A1 1.0

f CallOrig_DldDgts__hid2 1
s CallOrig_DldDgts__hid2 1.0

f SSF_o_A2 1
s SSF_o_A2 1.0
y SSF_o_A2 LRF_h_E1 1.0

f SSF_o_A3 1
s SSF_o_A3 1.0

f CallSetup_hid47 1
s CallSetup_hid47 1.0

f SSF_o_A4 1
s SSF_o_A4 1.0
y SSF_o_A4 SSF_t_E2 1.0

f SSF_o_A5 1
```

```
s SSF_o_A5 1.0

f CallRelease_hid70 1
s CallRelease_hid70 1.0

f SSF_o_A6 1
s SSF_o_A6 1.0
z SSF_o_A6 DefaultTask1_E1 1.0

f SSF_o_A7 1
s SSF_o_A7 1.0

f AnswerComplete_hid63 1
s AnswerComplete_hid63 1.0

f SSF_o_A8 1
s SSF_o_A8 1.0
z SSF_o_A8 DefaultTask2_E1 1.0

:
SSF_o_A1 ->  CallOrig_DldDgts__hid2;
CallOrig_DldDgts__hid2 ->  SSF_o_A2;
SSF_o_A2 ->  SSF_o_A3;
SSF_o_A3 ->  CallSetup_hid47;
CallSetup_hid47 ->  SSF_o_A4;
SSF_o_A4 ->  SSF_o_A5;
SSF_o_A5 ->  CallRelease_hid70;
CallRelease_hid70 ->  SSF_o_A6;
SSF_o_A6 ->  SSF_o_A7;
SSF_o_A7 ->  AnswerComplete_hid63;
AnswerComplete_hid63 ->  SSF_o_A8
-1

A LRF_h
f LRF_h_A1 1
s LRF_h_A1 1.0

f LocateMS_MIN__hid4 1
s LocateMS_MIN__hid4 1.0

f LRF_h_A2 1
s LRF_h_A2 1.0

f LRF_h_A3 1
s LRF_h_A3 1.0

f LRF_h_A4 1
s LRF_h_A4 1.0

f LRF_h_A5 1
s LRF_h_A5 1.0

f LRF_h_A6 1
s LRF_h_A6 1.0

f LRF_h_A7 1
s LRF_h_A7 1.0

f LRF_h_A8 1
```

```
s LRF_h_A8 1.0

f LRF_h_A9 1
s LRF_h_A9 1.0

f LRF_h_A10 1
s LRF_h_A10 1.0
y LRF_h_A10 LRF_v_E1 1.0

f LRF_h_A11 1
s LRF_h_A11 1.0

f LRF_h_A12 1
s LRF_h_A12 1.0

f LRF_h_A13 1
s LRF_h_A13 1.0

f LRF_h_A14 1
s LRF_h_A14 1.0

f OrJoin_Lqn_A1 1
s OrJoin_Lqn_A1 1.0

f OrJoin_Lqn_A2 1
s OrJoin_Lqn_A2 1.0

f ReplyToOriginating_hid10 1
s ReplyToOriginating_hid10 1.0

f ReturnAccessDenied_hid16 1
s ReturnAccessDenied_hid16 1.0

f LRF_h_A15 1
s LRF_h_A15 1.0

:
LRF_h_A1 -> LocateMS_MIN__hid4;
LocateMS_MIN__hid4 ->  LRF_h_A2;
LRF_h_A2 -> (0.500000) LRF_h_A3 + (0.500000) LRF_h_A5;
LRF_h_A3 -> LRF_h_A4;
LRF_h_A5 -> LRF_h_A6;
LRF_h_A6 -> (0.500000) LRF_h_A7 + (0.500000) LRF_h_A9;
LRF_h_A9 -> LRF_h_A10;
LRF_h_A7 -> LRF_h_A8;
LRF_h_A10 -> LRF_h_A11;
LRF_h_A11 -> LRF_h_A12;
LRF_h_A4 + LRF_h_A12 -> OrJoin_Lqn_A1;
OrJoin_Lqn_A1 -> ReturnAccessDenied_hid16;
ReturnAccessDenied_hid16 -> LRF_h_A13;
LRF_h_A13[LRF_h_E1];
LRF_h_A8 + LRF_h_A14 -> OrJoin_Lqn_A2;
OrJoin_Lqn_A2 -> ReplyToOriginating_hid10;
ReplyToOriginating_hid10 -> LRF_h_A15;
LRF_h_A15[LRF_h_E1]
-1

A LRF_v
f LRF_v_A1 1
```

```
s LRF_v_A1 1.0

f LocateMobile2_MIN__hid5 1
s LocateMobile2_MIN__hid5 1.0

f LRF_v_A2 1
s LRF_v_A2 1.0
y LRF_v_A2 SSF_t_E1 1.0

f LRF_v_A3 1
s LRF_v_A3 1.0

f RelayAccesDenied_hid25 1
s RelayAccesDenied_hid25 1.0

f LRF_v_A4 1
s LRF_v_A4 1.0

f LRF_v_A5 1
s LRF_v_A5 1.0

f RelayTLDN_hid9 1
s RelayTLDN_hid9 1.0

f LRF_v_A6 1
s LRF_v_A6 1.0

:
LRF_v_A1 ->  LocateMobile2_MIN__hid5;
LocateMobile2_MIN__hid5 ->  LRF_v_A2;
LRF_v_A2 ->  LRF_v_A3;
LRF_v_A3 ->  RelayAccesDenied_hid25;
RelayAccesDenied_hid25 ->  LRF_v_A4;
LRF_v_A4[LRF_v_E1];
LRF_v_A4 ->  LRF_v_A5;
LRF_v_A5 ->  RelayTLDN_hid9;
RelayTLDN_hid9 ->  LRF_v_A6;
LRF_v_A6[LRF_v_E1]
-1

A SSF_t
f SSF_t_A1 1
s SSF_t_A1 1.0

f SSF_t_A2 1
s SSF_t_A2 1.0

f SSF_t_A3 1
s SSF_t_A3 1.0

f AccessDenied_hid24 1
s AccessDenied_hid24 1.0

f SSF_t_A4 1
s SSF_t_A4 1.0

f SSF_t_A5 1
s SSF_t_A5 1.0
```

```
f GetProfile_hid41 1
s GetProfile_hid41 1.0

f AllocateTLDN_hid6 1
s AllocateTLDN_hid6 1.0

f SSF_t_A6 1
s SSF_t_A6 1.0

f SSF_t_A7 1
s SSF_t_A7 1.0

f SSF_t_A8 1
s SSF_t_A8 1.0

f SSF_t_A9 1
s SSF_t_A9 1.0

f PlayAnnouncement_hid69 1
s PlayAnnouncement_hid69 1.0

f SSF_t_A10 1
s SSF_t_A10 1.0

f SSF_t_A11 1
s SSF_t_A11 1.0

f SSF_t_A12 1
s SSF_t_A12 1.0

:
SSF_t_A1 ->  SSF_t_A2;
SSF_t_A2 ->  (0.500000) SSF_t_A3 + (0.500000) SSF_t_A5;
SSF_t_A3 ->  AccessDenied_hid24;
AccessDenied_hid24 ->  SSF_t_A4;
SSF_t_A4[SSF_t_E1];
SSF_t_A5 ->  GetProfile_hid41;
GetProfile_hid41 ->  AllocateTLDN_hid6;
AllocateTLDN_hid6 ->  SSF_t_A6;
SSF_t_A6[SSF_t_E1];
SSF_t_A7 ->  SSF_t_A8;
SSF_t_A8 ->  (0.500000) SSF_t_A9 + (0.500000) SSF_t_A11;
SSF_t_A9 ->  PlayAnnouncement_hid69;
PlayAnnouncement_hid69 ->  SSF_t_A10;
SSF_t_A10[SSF_t_E2];
SSF_t_A11 ->  SSF_t_A12;
SSF_t_A12[SSF_t_E2]
-1

A RefTask1
f RefTask1_A1 1
s RefTask1_A1 1.0
z RefTask1_A1 SSF_o_E1 1.0

-1

A DefaultTask1
f DefaultTask1_A1 1
s DefaultTask1_A1 1.0
```

```
f DefaultTask1_A2 1
s DefaultTask1_A2 1.0

:
DefaultTask1_A1 ->  DefaultTask1_A2
-1

A DefaultTask2
f DefaultTask2_A1 1
s DefaultTask2_A1 1.0

f DefaultTask2_A2 1
s DefaultTask2_A2 1.0

:
DefaultTask2_A1 ->  DefaultTask2_A2
-1
```

APPENDIX E - Hand-Off Protocol LQN File

```
# UCM2LQN output

G
""
1e-05
50
5
0.9
-1

P 0
p P_Infinite f i R 1.000000
-1

T 0
t ProxyA f ProxyA_E1 ProxyA_E2 ProxyA_E3 -1  P_Infinite
t DeviceA f DeviceA_E1 DeviceA_E2 -1  P_Infinite
t PartyA f PartyA_E1 -1  P_Infinite
t MainController f MainController_E1 MainController_E2 MainController_E3 -1
P_Infinite
# WARNING: task 'OS' running on processor 'P_Infinite' (id -1) has no entries.

t MessageSystem f MessageSystem_E1 MessageSystem_E2 MessageSystem_E3 -1  P_Infinite
t DeviceHandler f DeviceHandler_E1 DeviceHandler_E2 DeviceHandler_E3 -1  P_Infinite
t ProxyB f ProxyB_E1 ProxyB_E2 ProxyB_E3 -1  P_Infinite
t BackupProcess f BackupProcess_E1 -1  P_Infinite
t PartyB f PartyB_E1 -1  P_Infinite
t TupleSpace f TupleSpace_E1 -1  P_Infinite
t ProcessB f ProcessB_E1 ProcessB_E2 ProcessB_E3 -1  P_Infinite
t ProcessA f ProcessA_E1 ProcessA_E2 -1  P_Infinite
t RefTask1 r RefTask1_E1 -1  P_Infinite
-1

E 0
A ProxyA_E1 ProxyA_A1
F ProxyA_E1 ProxyB_E1 1.0 -1
A ProxyA_E2 ProxyA_A3
F ProxyA_E2 ProxyB_E2 1.0 -1
A ProxyA_E3 ProxyA_A5
A DeviceA_E1 DeviceA_A1
F DeviceA_E1 ProxyA_E1 1.0 -1
A DeviceA_E2 DeviceA_A3
A PartyA_E1 PartyA_A1
A MainController_E1 MainController_A1
F MainController_E1 ProcessA_E1 1.0 -1
A MainController_E2 MainController_A5
F MainController_E2 ProcessA_E2 1.0 -1
A MainController_E3 MainController_A9
A MessageSystem_E1 MessageSystem_A1
F MessageSystem_E1 MainController_E1 1.0 -1
A MessageSystem_E2 MessageSystem_A3
F MessageSystem_E2 MainController_E2 1.0 -1
A MessageSystem_E3 MessageSystem_A5
A DeviceHandler_E1 DeviceHandler_A1
A DeviceHandler_E2 DeviceHandler_A3
```

```
A DeviceHandler_E3 DeviceHandler_A5
A ProxyB_E1 ProxyB_A1
F ProxyB_E1 MessageSystem_E1 1.0 -1
A ProxyB_E2 ProxyB_A3
F ProxyB_E2 MessageSystem_E2 1.0 -1
A ProxyB_E3 ProxyB_A5
A BackupProcess_E1 BackupProcess_A1
A PartyB_E1 PartyB_A1
A TupleSpace_E1 TupleSpace_A1
F TupleSpace_E1 ProcessB_E2 1.0 -1
A ProcessB_E1 ProcessB_A1
A ProcessB_E2 ProcessB_A3
F ProcessB_E2 PartyB_E1 1.0 -1
A ProcessB_E3 ProcessB_A7
A ProcessA_E1 ProcessA_A1
A ProcessA_E2 ProcessA_A3
F ProcessA_E2 TupleSpace_E1 1.0 -1
A RefTask1_E1 RefTask1_A1
-1

A ProxyA
f ProxyA_A1 1
s ProxyA_A1 1.000000
f pA_e11_hid55 1
s pA_e11_hid55 1.000000
f ProxyA_A2 1
s ProxyA_A2 1.000000
y ProxyA_A2 ProxyB_E1 1.000000
f ProxyA_A3 1
s ProxyA_A3 1.000000
f pA_e12_hid57 1
s pA_e12_hid57 1.000000
f ProxyA_A4 1
s ProxyA_A4 1.000000
y ProxyA_A4 ProxyB_E2 1.000000
f ProxyA_A5 1
s ProxyA_A5 1.000000
f pA_e13_hid59 1
s pA_e13_hid59 1.000000
f ProxyA_A6 1
s ProxyA_A6 1.000000
z ProxyA_A6 ProxyB_E3 1.000000

:
ProxyA_A1 -> pA_e11_hid55;
pA_e11_hid55 -> ProxyA_A2;
ProxyA_A2[ProxyA_E1];
ProxyA_A3 -> pA_e12_hid57;
pA_e12_hid57 -> ProxyA_A4;
ProxyA_A4[ProxyA_E2];
ProxyA_A5 -> pA_e13_hid59;
pA_e13_hid59 -> ProxyA_A6
-1

A DeviceA
f DeviceA_A1 1
s DeviceA_A1 1.000000
f devA_e11_hid52 1
s devA_e11_hid52 1.000000
```

```
f DeviceA_A2 1
s DeviceA_A2 1.000000
y DeviceA_A2 ProxyA_E1 1.000000
f DeviceA_A3 1
s DeviceA_A3 1.000000
f devA_e12_hid53 1
s devA_e12_hid53 1.000000
f DeviceA_A4 1
s DeviceA_A4 1.000000
y DeviceA_A4 ProxyA_E2 1.000000
f DeviceA_A5 1
s DeviceA_A5 1.000000
f devA_e13_hid54 1
s devA_e13_hid54 1.000000
f DeviceA_A6 1
s DeviceA_A6 1.000000
z DeviceA_A6 ProxyA_E3 1.000000

:
DeviceA_A1 -> devA_e11_hid52;
devA_e11_hid52 -> DeviceA_A2;
DeviceA_A2[DeviceA_E1];
DeviceA_A3 -> devA_e12_hid53;
devA_e12_hid53 -> DeviceA_A4;
DeviceA_A4 -> DeviceA_A5;
DeviceA_A5 -> devA_e13_hid54;
devA_e13_hid54 -> DeviceA_A6
-1

A PartyA
f PartyA_A1 1
s PartyA_A1 1.000000
f ptyA_e11_hid76 1
s ptyA_e11_hid76 1.000000
f PartyA_A2 1
s PartyA_A2 1.000000
y PartyA_A2 DeviceA_E1 1.000000
f PartyA_A3 1
s PartyA_A3 1.000000
f ptyA_e12_hid79 1
s ptyA_e12_hid79 1.000000
f PartyA_A4 1
s PartyA_A4 1.000000
z PartyA_A4 DeviceA_E2 1.000000

:
PartyA_A1 -> ptyA_e11_hid76;
ptyA_e11_hid76 -> PartyA_A2;
PartyA_A2 -> PartyA_A3;
PartyA_A3 -> ptyA_e12_hid79;
ptyA_e12_hid79 -> PartyA_A4
-1

A MainController
f MainController_A1 1
s MainController_A1 1.000000
f MainController_A2 1
s MainController_A2 1.000000
y MainController_A2 DeviceHandler_E1 1.000000
```

```
f MainController_A3 1
s MainController_A3 1.000000
f MainController_A4 1
s MainController_A4 1.000000
y MainController_A4 ProcessA_E1 1.000000
f MainController_A5 1
s MainController_A5 1.000000
f MainController_A6 1
s MainController_A6 1.000000
y MainController_A6 DeviceHandler_E2 1.000000
f MainController_A7 1
s MainController_A7 1.000000
f MainController_A8 1
s MainController_A8 1.000000
y MainController_A8 ProcessA_E2 1.000000
f MainController_A9 1
s MainController_A9 1.000000
f MainController_A10 1
s MainController_A10 1.000000
y MainController_A10 DeviceHandler_E3 1.000000
f MainController_A11 1
s MainController_A11 1.000000
f MainController_A12 1
s MainController_A12 1.000000
z MainController_A12 ProcessB_E3 1.000000

:
MainController_A1 -> MainController_A2;
MainController_A2 -> MainController_A3;
MainController_A3 -> MainController_A4;
MainController_A4[MainController_E1];
MainController_A5 -> MainController_A6;
MainController_A6 -> MainController_A7;
MainController_A7 -> MainController_A8;
MainController_A8[MainController_E2];
MainController_A9 -> MainController_A10;
MainController_A10 -> MainController_A11;
MainController_A11 -> MainController_A12
-1

A MessageSystem
f MessageSystem_A1 1
s MessageSystem_A1 1.000000
f msg_e11_hid33 1
s msg_e11_hid33 1.000000
f MessageSystem_A2 1
s MessageSystem_A2 1.000000
y MessageSystem_A2 MainController_E1 1.000000
f MessageSystem_A3 1
s MessageSystem_A3 1.000000
f msg_e12_hid34 1
s msg_e12_hid34 1.000000
f MessageSystem_A4 1
s MessageSystem_A4 1.000000
y MessageSystem_A4 MainController_E2 1.000000
f MessageSystem_A5 1
s MessageSystem_A5 1.000000
f msg_e13_hid35 1
s msg_e13_hid35 1.000000
```

```
f MessageSystem_A6 1
s MessageSystem_A6 1.000000
z MessageSystem_A6 MainController_E3 1.000000

:
MessageSystem_A1 -> msg_e11_hid33;
msg_e11_hid33 -> MessageSystem_A2;
MessageSystem_A2[MessageSystem_E1];
MessageSystem_A3 -> msg_e12_hid34;
msg_e12_hid34 -> MessageSystem_A4;
MessageSystem_A4[MessageSystem_E2];
MessageSystem_A5 -> msg_e13_hid35;
msg_e13_hid35 -> MessageSystem_A6
-1

A DeviceHandler
f DeviceHandler_A1 1
s DeviceHandler_A1 1.000000
f devH_e11_hid36 1
s devH_e11_hid36 1.000000
f DeviceHandler_A2 1
s DeviceHandler_A2 1.000000
f DeviceHandler_A3 1
s DeviceHandler_A3 1.000000
f devH_e12_hid38 1
s devH_e12_hid38 1.000000
f DeviceHandler_A4 1
s DeviceHandler_A4 1.000000
f DeviceHandler_A5 1
s DeviceHandler_A5 1.000000
f devH_e13_hid40 1
s devH_e13_hid40 1.000000
f DeviceHandler_A6 1
s DeviceHandler_A6 1.000000

:
DeviceHandler_A1 -> devH_e11_hid36;
devH_e11_hid36 -> DeviceHandler_A2;
DeviceHandler_A2[DeviceHandler_E1];
DeviceHandler_A3 -> devH_e12_hid38;
devH_e12_hid38 -> DeviceHandler_A4;
DeviceHandler_A4[DeviceHandler_E2];
DeviceHandler_A5 -> devH_e13_hid40;
devH_e13_hid40 -> DeviceHandler_A6;
DeviceHandler_A6[DeviceHandler_E3]
-1

A ProxyB
f ProxyB_A1 1
s ProxyB_A1 1.000000
f pB_e11_hid61 1
s pB_e11_hid61 1.000000
f ProxyB_A2 1
s ProxyB_A2 1.000000
y ProxyB_A2 MessageSystem_E1 1.000000
f ProxyB_A3 1
s ProxyB_A3 1.000000
f pB_e12_hid64 1
s pB_e12_hid64 1.000000
```

```
f ProxyB_A4 1
s ProxyB_A4 1.000000
y ProxyB_A4 MessageSystem_E2 1.000000
f ProxyB_A5 1
s ProxyB_A5 1.000000
f pB_e13_hid67 1
s pB_e13_hid67 1.000000
f ProxyB_A6 1
s ProxyB_A6 1.000000
z ProxyB_A6 MessageSystem_E3 1.000000

:
ProxyB_A1 -> pB_e11_hid61;
pB_e11_hid61 -> ProxyB_A2;
ProxyB_A2[ProxyB_E1];
ProxyB_A3 -> pB_e12_hid64;
pB_e12_hid64 -> ProxyB_A4;
ProxyB_A4[ProxyB_E2];
ProxyB_A5 -> pB_e13_hid67;
pB_e13_hid67 -> ProxyB_A6
-1

A BackupProcess
f BackupProcess_A1 1
s BackupProcess_A1 1.000000
f psBk_e11_hid74 1
s psBk_e11_hid74 1.000000
f BackupProcess_A2 1
s BackupProcess_A2 1.000000

:
BackupProcess_A1 -> psBk_e11_hid74;
psBk_e11_hid74 -> BackupProcess_A2;
BackupProcess_A2[BackupProcess_E1]
-1

A PartyB
f PartyB_A1 1
s PartyB_A1 1.000000
f ptyB_e11_hid73 1
s ptyB_e11_hid73 1.000000
f PartyB_A2 1
s PartyB_A2 1.000000

:
PartyB_A1 -> ptyB_e11_hid73;
ptyB_e11_hid73 -> PartyB_A2;
PartyB_A2[PartyB_E1]
-1

A TupleSpace
f TupleSpace_A1 1
s TupleSpace_A1 1.000000
f ts_in_hid45 1
s ts_in_hid45 1.000000
f TupleSpace_A2 1
s TupleSpace_A2 1.000000
y TupleSpace_A2 ProcessB_E1 1.000000
f TupleSpace_A3 1
```

```
s TupleSpace_A3 1.000000
f ts_out_hid46 1
s ts_out_hid46 1.000000
f TupleSpace_A4 1
s TupleSpace_A4 1.000000
y TupleSpace_A4 ProcessB_E2 1.000000

:
TupleSpace_A1 -> ts_in_hid45;
ts_in_hid45 -> TupleSpace_A2;
TupleSpace_A2 -> TupleSpace_A3;
TupleSpace_A3 -> ts_out_hid46;
ts_out_hid46 -> TupleSpace_A4;
TupleSpace_A4[TupleSpace_E1]
-1

A ProcessB
f ProcessB_A1 1
s ProcessB_A1 1.000000
f psB_e11_hid72 1
s psB_e11_hid72 1.000000
f ProcessB_A2 1
s ProcessB_A2 1.000000
f ProcessB_A3 1
s ProcessB_A3 1.000000
f ProcessB_A4 1
s ProcessB_A4 1.000000
f ProcessB_A5 1
s ProcessB_A5 1.000000
f psB_e13_hid84 1
s psB_e13_hid84 1.000000
f ProcessB_A6 1
s ProcessB_A6 1.000000
y ProcessB_A6 PartyB_E1 1.000000
f ProcessB_A7 1
s ProcessB_A7 1.000000
f ProcessB_A8 1
s ProcessB_A8 1.000000
f ProcessB_A9 1
s ProcessB_A9 1.000000
f psB_e12_hid81 1
s psB_e12_hid81 1.000000
f ProcessB_A10 1
s ProcessB_A10 1.000000
f AndJoin_Lqn_A1 1
s AndJoin_Lqn_A1 1.000000
f ProcessB_A11 1
s ProcessB_A11 1.000000
y ProcessB_A11 BackupProcess_E1 1.000000
f ProcessB_A12 1
s ProcessB_A12 1.000000
f ProcessB_A13 1
s ProcessB_A13 1.000000

:
ProcessB_A1 -> psB_e11_hid72;
psB_e11_hid72 -> ProcessB_A2;
ProcessB_A2[ProcessB_E1];
ProcessB_A3 -> ProcessB_A4;
```

```
ProcessB_A4 -> ProcessB_A5 & ProcessB_A9;
ProcessB_A5 -> psB_e13_hid84;
psB_e13_hid84 -> ProcessB_A6;
ProcessB_A6[ProcessB_E2];
ProcessB_A7 -> ProcessB_A8;
ProcessB_A9 -> psB_e12_hid81;
psB_e12_hid81 -> ProcessB_A10;
ProcessB_A8 & ProcessB_A10 -> AndJoin_Lqn_A1;
AndJoin_Lqn_A1 -> ProcessB_A11;
ProcessB_A11 -> ProcessB_A12;
ProcessB_A12 -> ProcessB_A13
-1

A ProcessA
f ProcessA_A1 1
s ProcessA_A1 1.000000
f psA_e11_hid43 1
s psA_e11_hid43 1.000000
f ProcessA_A2 1
s ProcessA_A2 1.000000
f ProcessA_A3 1
s ProcessA_A3 1.000000
f psA_e12_hid70 1
s psA_e12_hid70 1.000000
f ProcessA_A4 1
s ProcessA_A4 1.000000
y ProcessA_A4 TupleSpace_E1 1.000000

:
ProcessA_A1 -> psA_e11_hid43;
psA_e11_hid43 -> ProcessA_A2;
ProcessA_A2[ProcessA_E1];
ProcessA_A3 -> psA_e12_hid70;
psA_e12_hid70 -> ProcessA_A4;
ProcessA_A4[ProcessA_E2]
-1

A RefTask1
f RefTask1_A1 1
s RefTask1_A1 1.000000
z RefTask1_A1 PartyA_E1 1.000000

-1
```