Chapter 1 - Introduction

This thesis presents UCM2LQN, an automated conversion tool that converts annotated Use Case Map (UCM) design models into Layered Queueing Nework (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 analisys are two vital, yet poorly coordinated aspects of the lifecycle 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 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 desing might have 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bit of reengineering, or even worse, the performance analysis results might be overlooked altogether since they are out of date.

Assuming that performance analysis is even done, as opposed to finding out how fast the system tuns after it has been implemented, this particular paradigm of separate fiefdoms between design and performance analysis is sadly enough pretty much the 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 Smyth, SPE came up in the late 1980's as a research idea and the first publications appearing in the early 1990's. The SPE software development model proposes developing performance models at the same time as the design takes place. The results from performance analysis performed early on in the 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 spared.

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 of 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 apropriate 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. 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. 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.

1.3. Contributions Of This Thesis

The thesis makes the following contributions:

- identification of corresponding constructs and patterns of interaction between the UCM and LQN notations
- software design methodology for incorporating additional modules in the UCMNav, both in terms of class and file structure and in defining an interface between the UCMNav and an additional module
- algorithm for traversing the internal UCM model of the UCMNav
- algorithm for detecting component boundary crossings in UCMs

- algorithm for interpreting the nature of messaging between UCM components
- algorithm for creating LQN objects based on UCM constructs and directing the traversal of the UCM accordingly
- validation of the UCM2LQN converter using “in-house” UCM models
- testing of the UCM2LQN converter using UCM models originating from industry

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 UCM-Nav 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 modelled using both notations. Chapter 4 describes the strategy used to integrate the UCM2LQN tool into the the UCM-Nav 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 reside at the same level of abstraction as Collaboration Diagrams but provide a better visual representation of how scenarios unfold.

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).

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 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.

UCM paths can be overlaid on components. Components represent functional or logical

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

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.

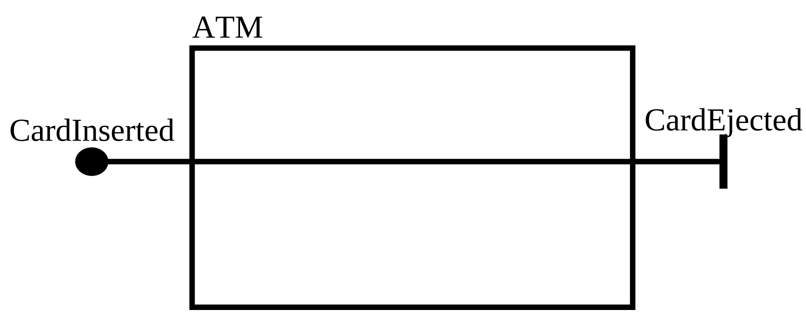

Figure 2-2: Simple UCM path for an ATM banking transaction overlaid on an ATM component.

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.

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

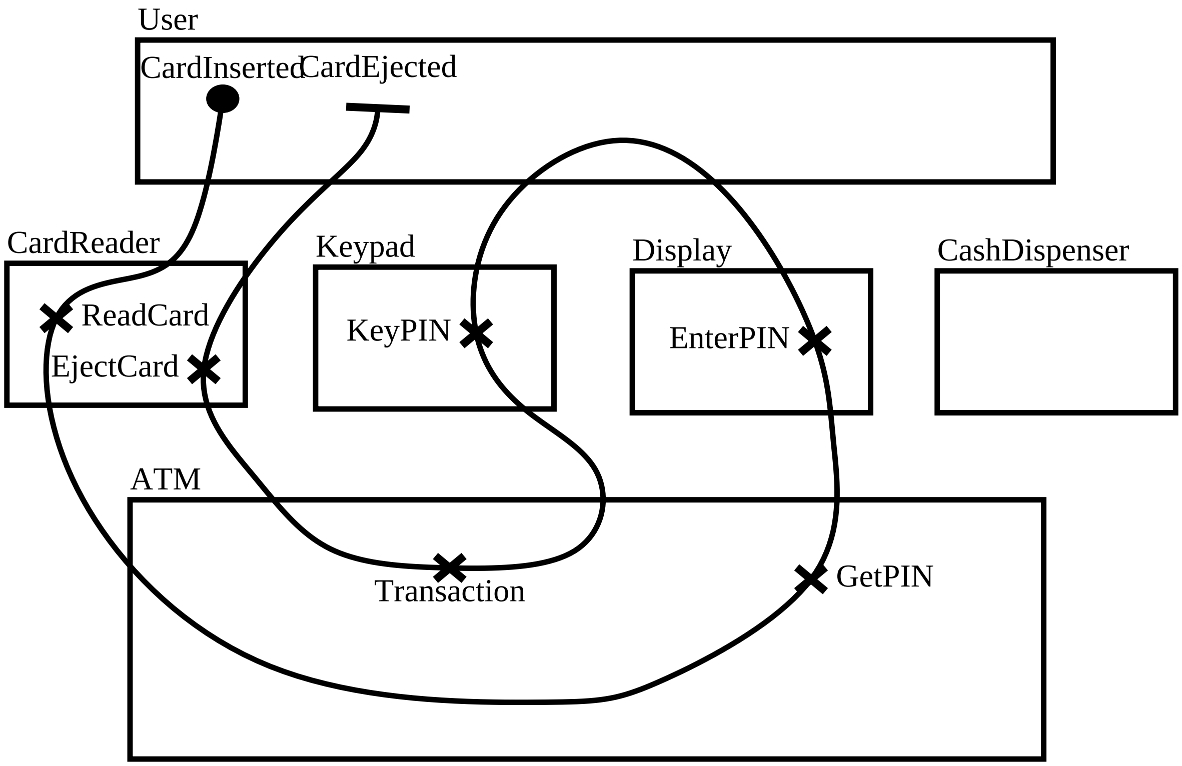

Figure 2-3: UCM with responsibilities and additional components for an ATM banking transaction.

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 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 plug-ins of their own.

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

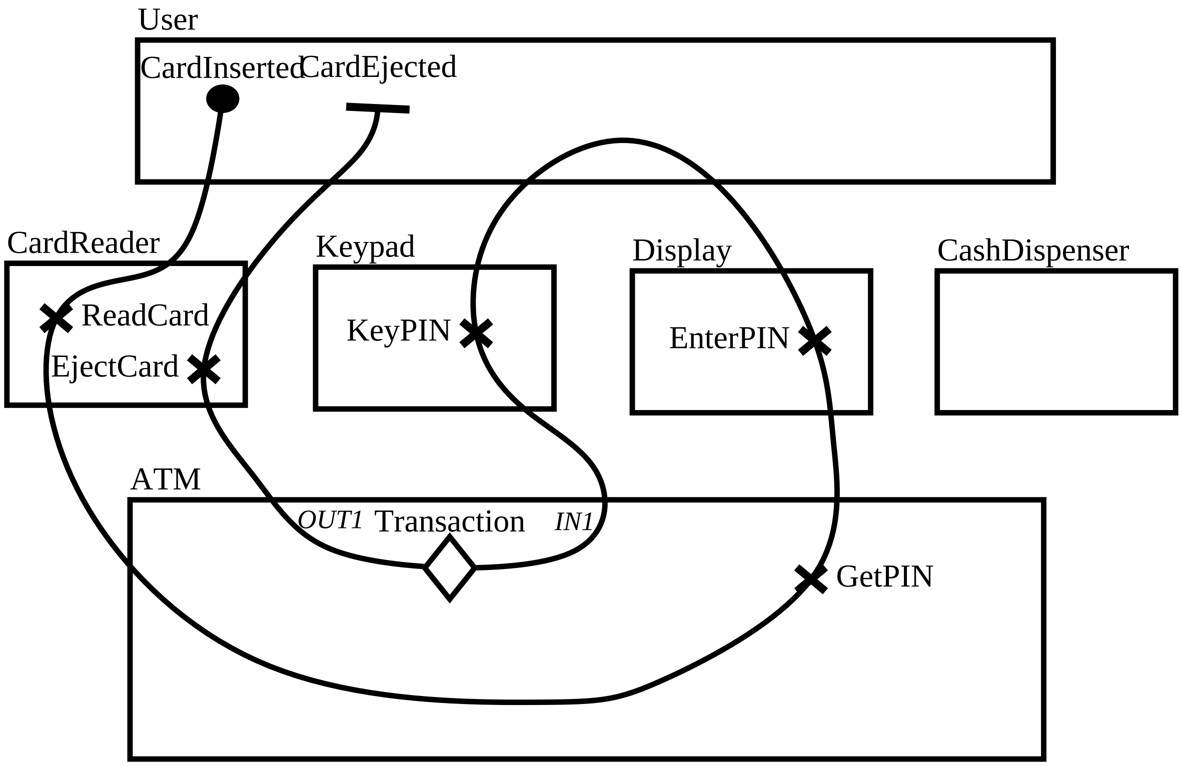

Figure 2-4: UCM for an ATM banking transaction with a stub for the transaction.

path splitting into two or more parallel paths) or logical AND joins (two or more parallel paths gathering into a single path), but there is no restriction as to how many paths must lead in or out of a synchronization. Any synchronization joining paths does require that tokens travelling along each incoming path all arrive at the synchronization before path traversal can proceed past the synchronization. Figure 2-6 shows a refined withdrawal plug-in with a parallel path segment that requests account information from a central bank database while 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

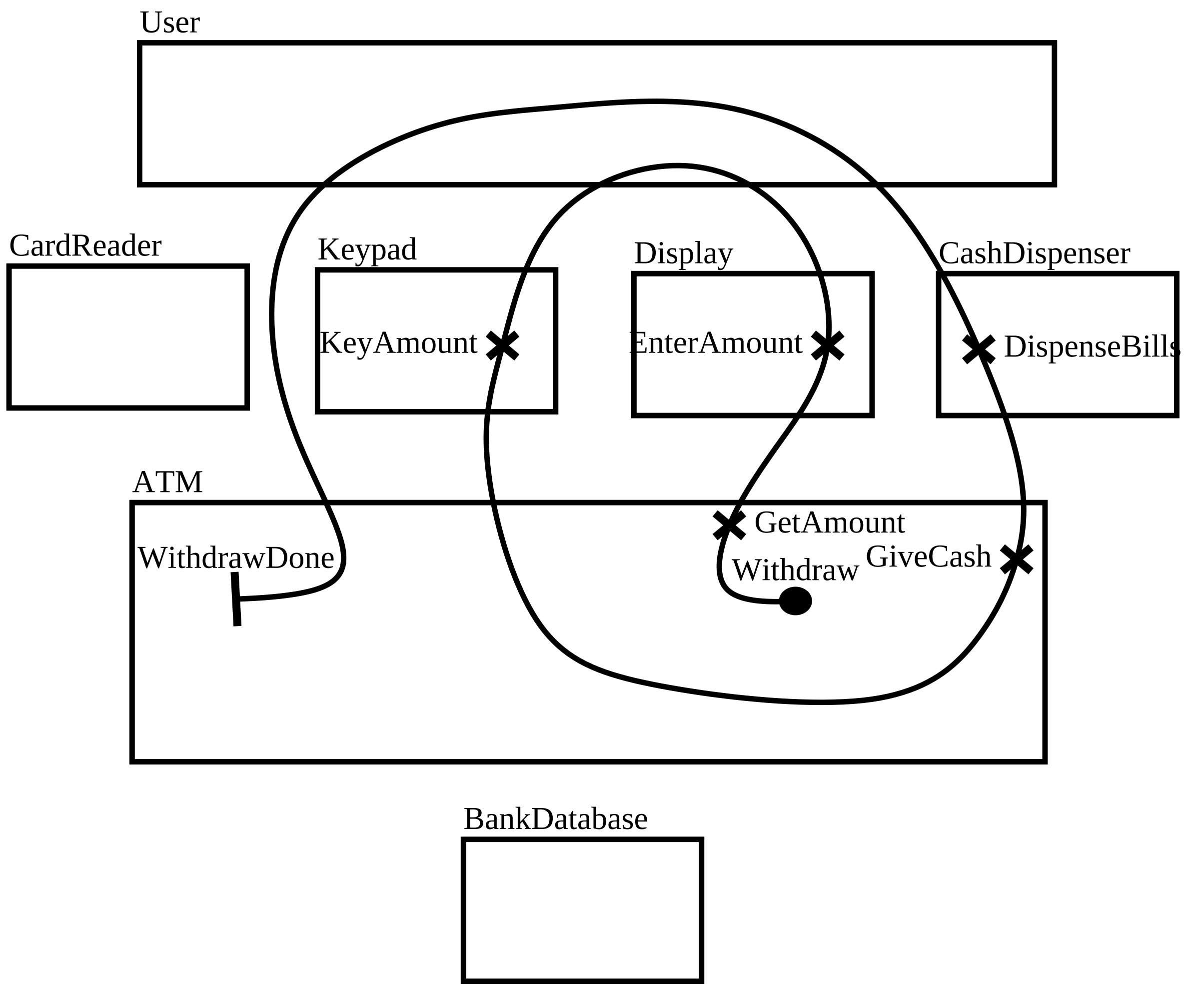

Figure 2-5: Withdrawal UCM used as a plug-in for the stub in Figure 2-4.

“cancel” button or is unable to provide a correct PIN.

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.

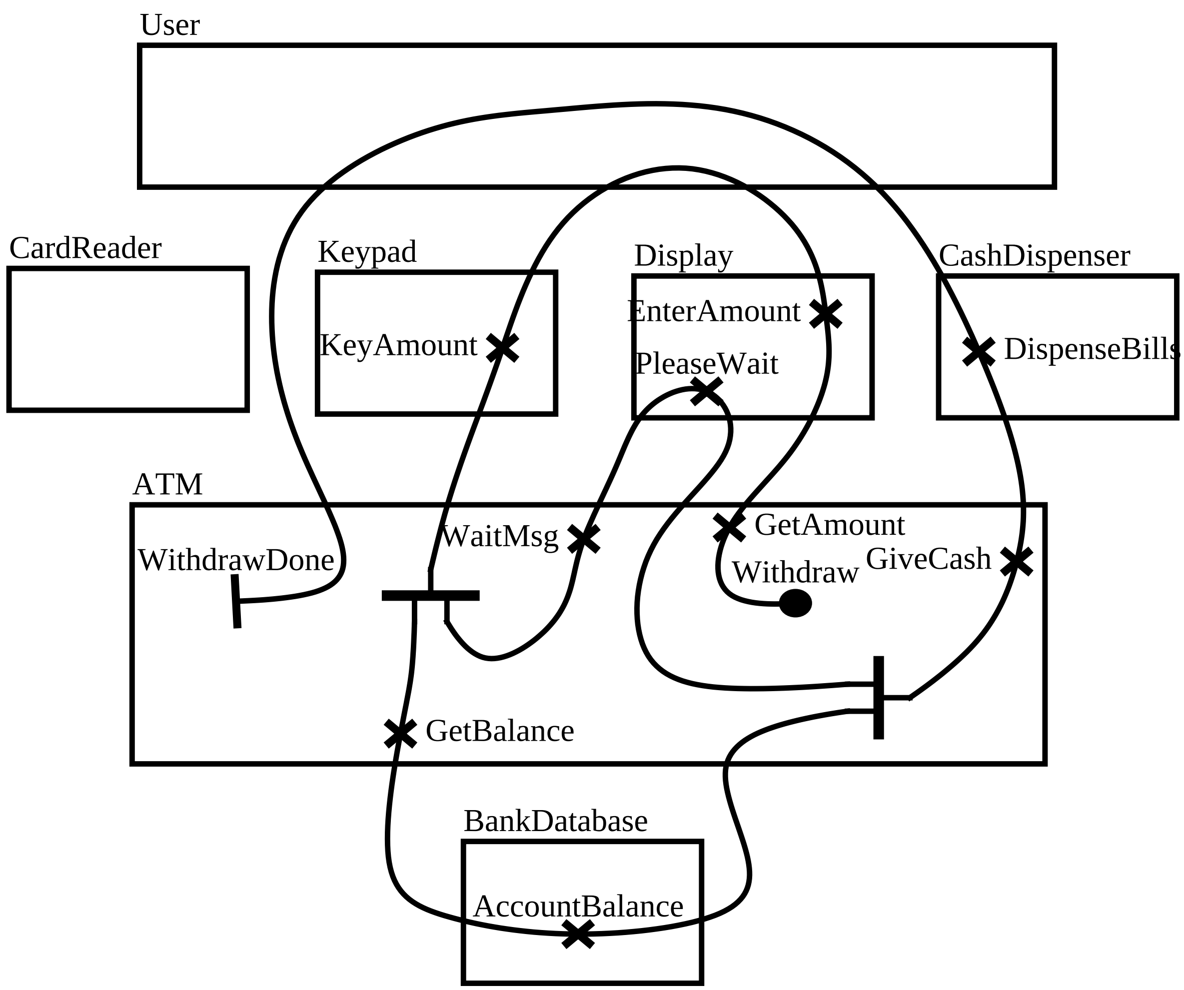

Figure 2-6: Refined withdrawal UCM with a parallel path segment getting account information from the bank database used as a plug-in for the Transaction stub defined in Figure 2-4.

2.1.2. The UCM Navigator

The UCM Navigator (UCMNav) is a UCM editing tool developed at Carleton University by Andrew Miga. It is currently used at Carleton University and the University of Ottawa, as well as within Nortel Networks and Mitel. 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.

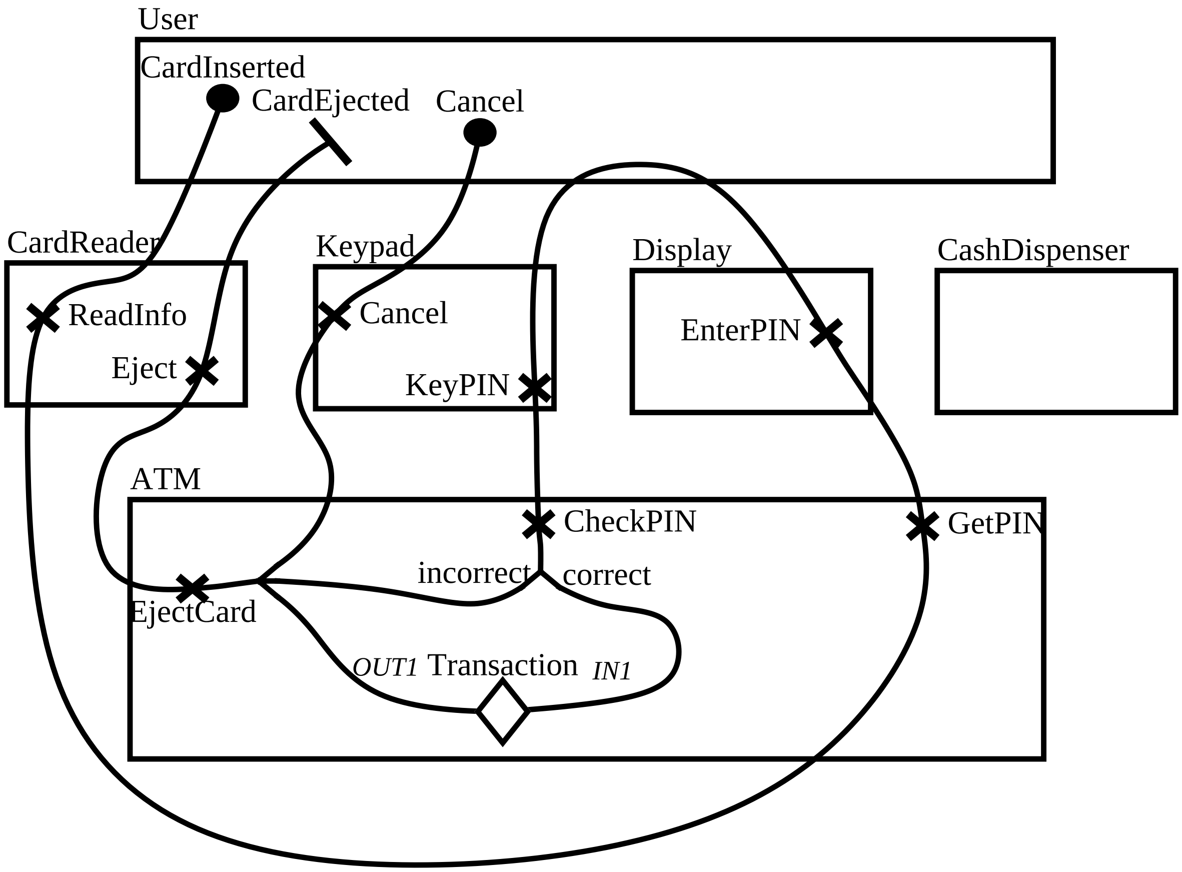

Figure 2-7: UCM for an ATM banking transaction with alternative paths that end the transaction.

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 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 ***.

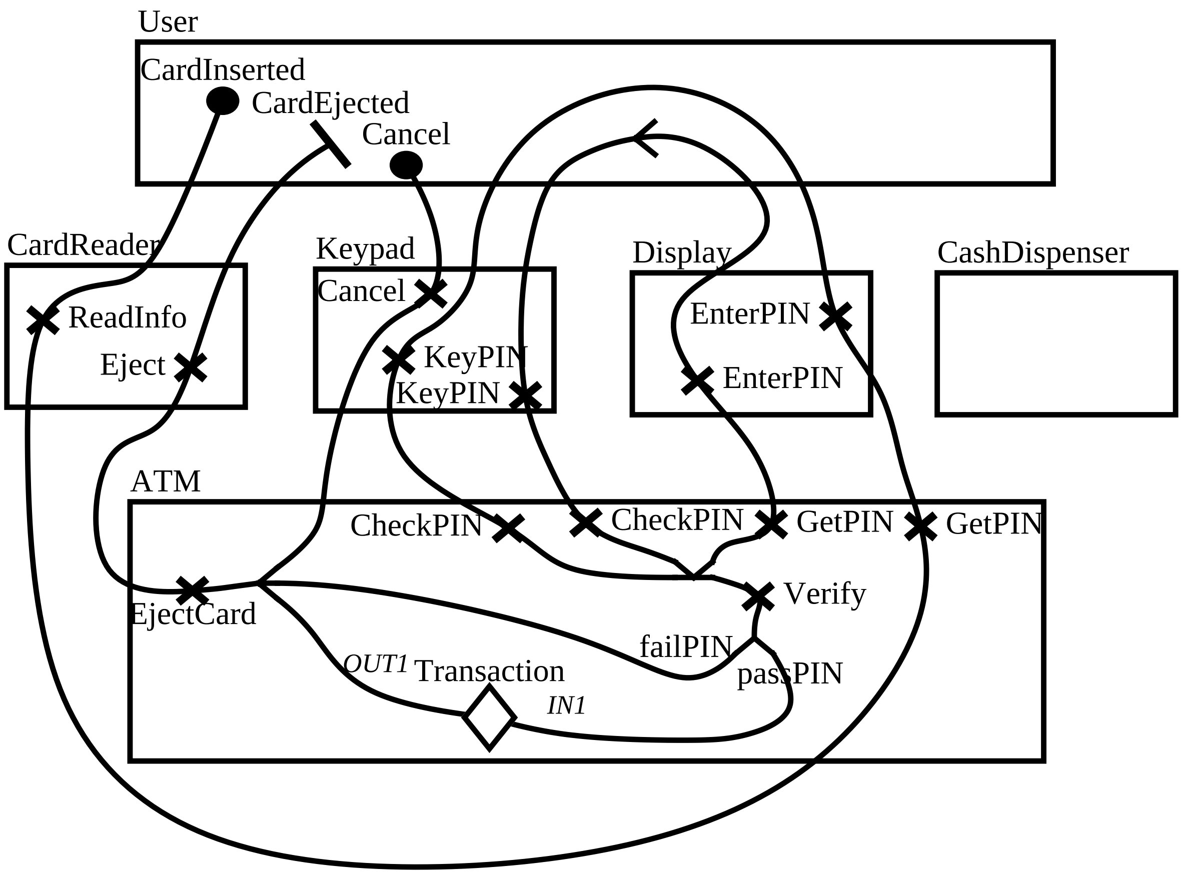

Figure 2-8: UCM path for an ATM banking transaction with a loop to get the correct PIN number.

2.2. Layered Queueing Network Background

Queueing Networks are based on a client-server paradigm. The clients 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 client-server relationships. In practice this means that queueing networks can only model hardware resources as pure servers and software tasks as pure clients.

Layered Queueing Networks (LQN) allow for of an arbitrary number of client-server levels. An LQN can thus model intermediate software servers and be used to detect software deadlocks.

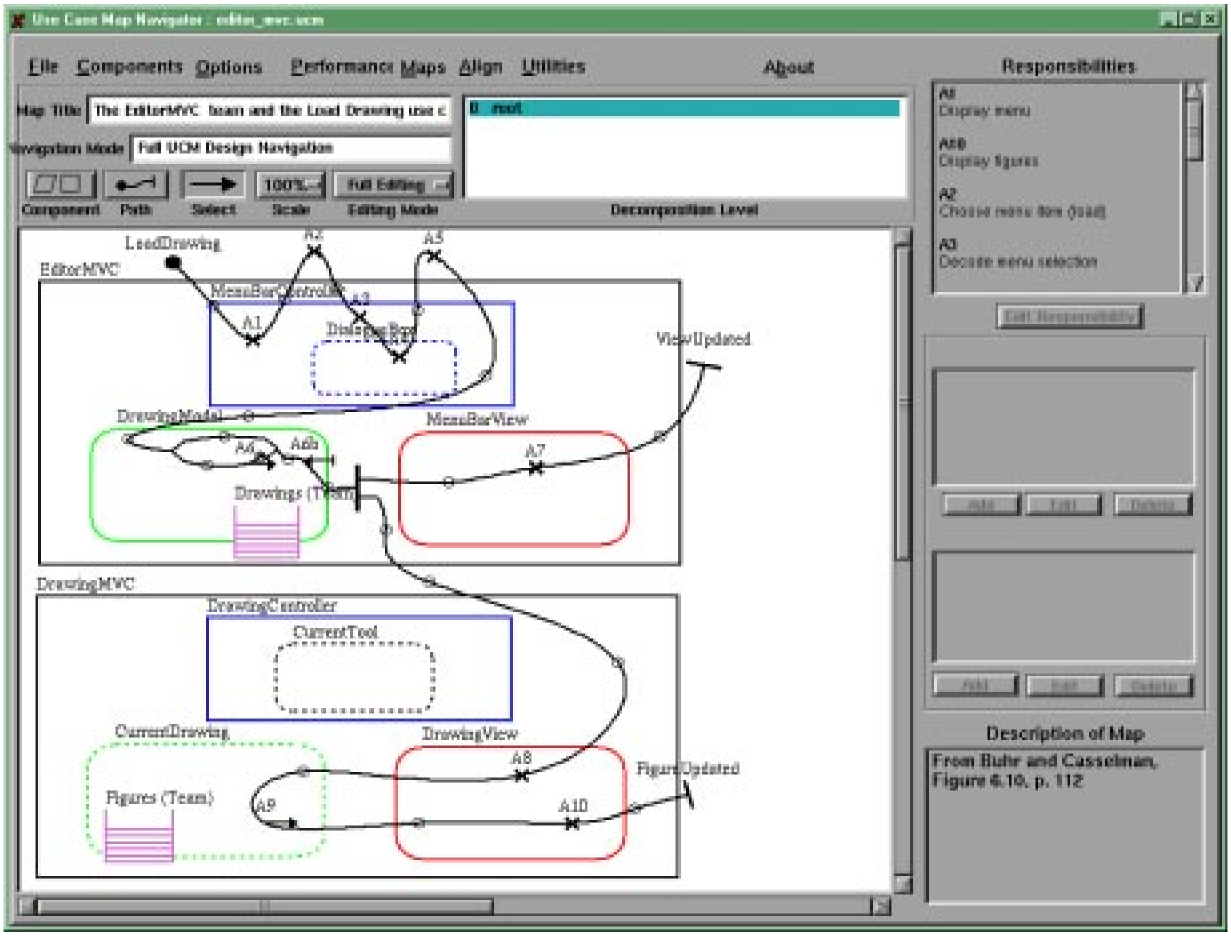

Figure 2-9: Screen shot of the UCMNav.

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 CPU's and disks. Figure 2-10 shows a task and two 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 messages: asynchronous and synchronous. Asynchronous mes-

Figure 2-10: LQN task and devices.

sages 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. Synchronous service requests may be forwarded to other tasks and it becomes their responsibility to provide the reply, in such a case the original task making the call remains blocked until it finally receives the reply. 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 correspond to method invocations, with a different entry for every kind of service a task provides. Entries may have their own service demands, either as requests from other tasks or from devices, or an entry may point to sequences 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. Thus entries and activities can be used to fully describe a task's functions. Figure 2-13 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.

For the purposes of this research it was assumed that LQN activities are the basic building blocks of LQN models. An LQN activity is assumed to directly correspond to a UCM responsi-

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

bility. 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

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

executed. Any entry or activity making a call must also specify the probability of that call being made, as well as its frequency if the call is repeated.

This type of information is not necessarily specified in UCM models, but the UCMNav does have facilities to specify it.

Figure 2-13: 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 supprt 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.

All three tools use the same 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 ***. LQNS is an analytic solver that breaks the LQN layers down into separate queueing network sub-models. The individual queueing networks can then 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 convergeance 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, where layers

2.2.3.2. ParaSRVN

The precursor to the current LQN notation was called Stochastic Rendez-Vous Networks (SRVN), hence the ‘SRVN’ in ParaSRVN.

2.2.3.3. LQN File Format

2.3. Creating Performance Models

- UCM2LQN is a method to create performance models

2.3.1. Software Performance Engineering

2.3.2. Creating Petri Net Models

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.

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.

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 jLqnDef 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.

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 3-1 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 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 3-2 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 another and does not returns 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 3-3 shows the corresponding UCM and LQN models for an asynchronous call.

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

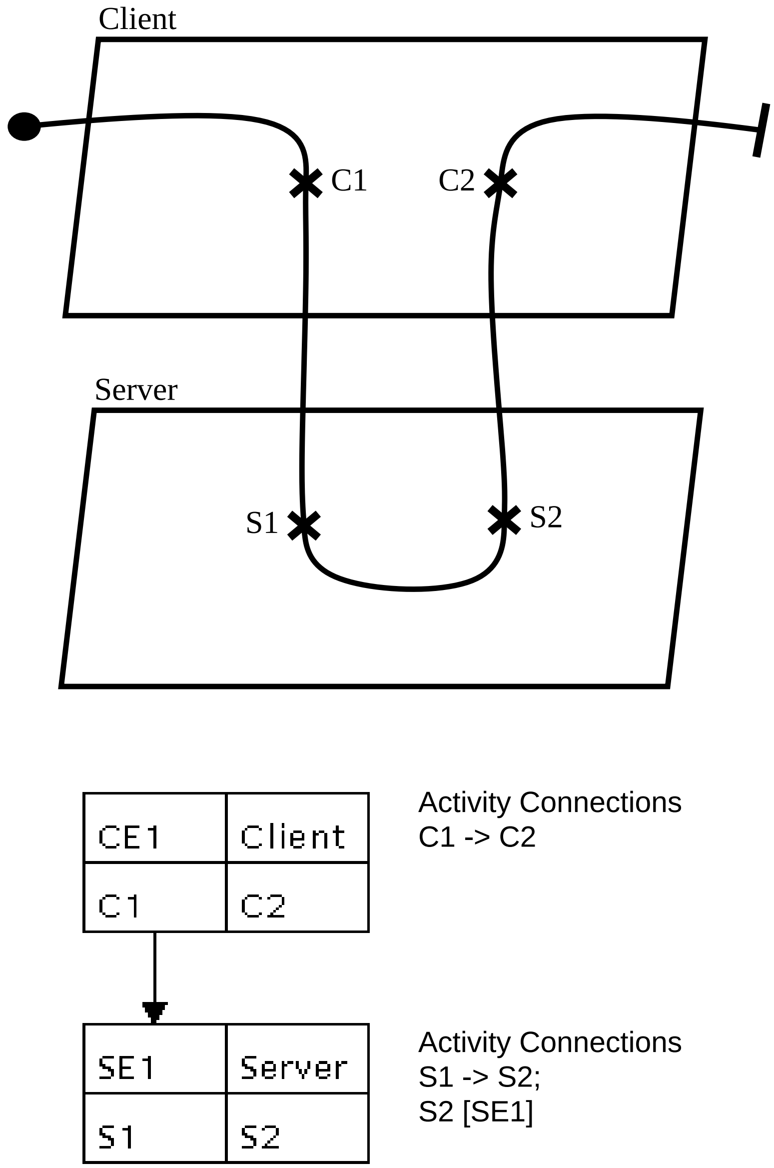

Figure 3-1: Corresponding UCM and LQN models for a simple synchronous call and return.

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 3-4 shows UCM and LQN models for a forwarding interaction.

3.2.4. Parallel Calls

The UCM path has an AND fork and then join in the calling component. By making calls

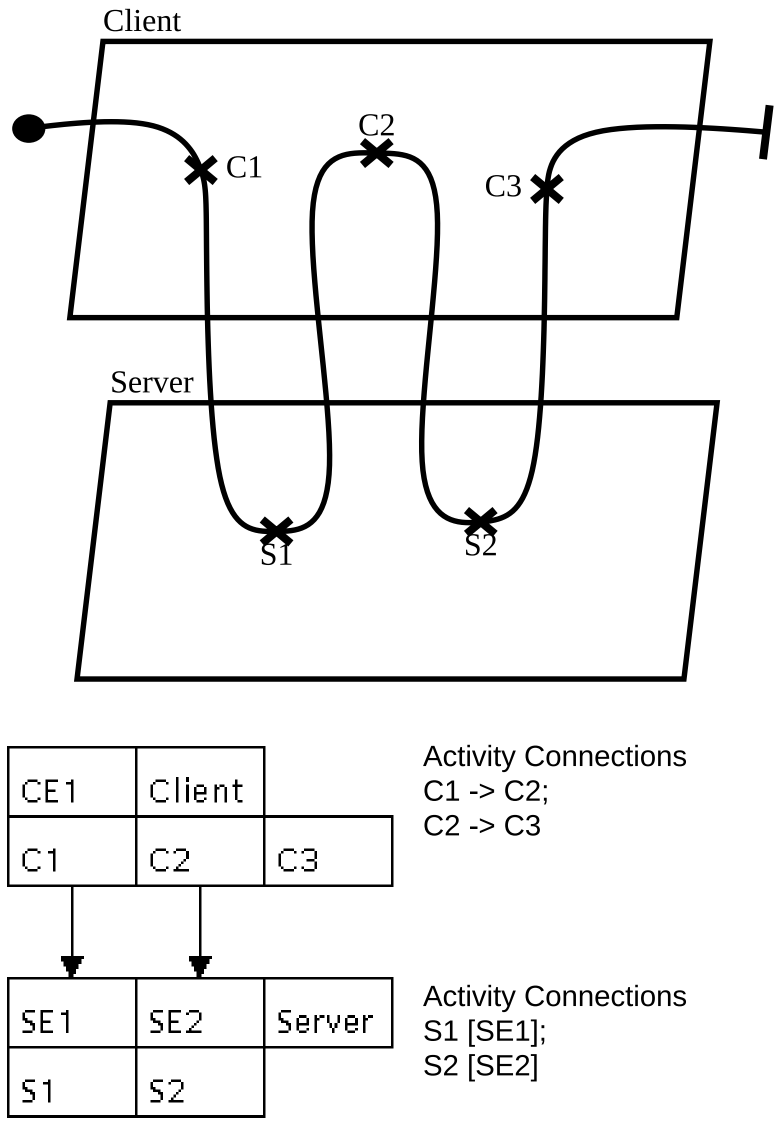

Figure 3-2: Corresponding UCM and LQN models for successive synchronous calls and returns.

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 3-5 shows the corresponding UCM and LQN models for such an instance.

3.2.5. Alternative Calls

Similarly to the parallel case above, the UCM path has an OR fork and join in the calling

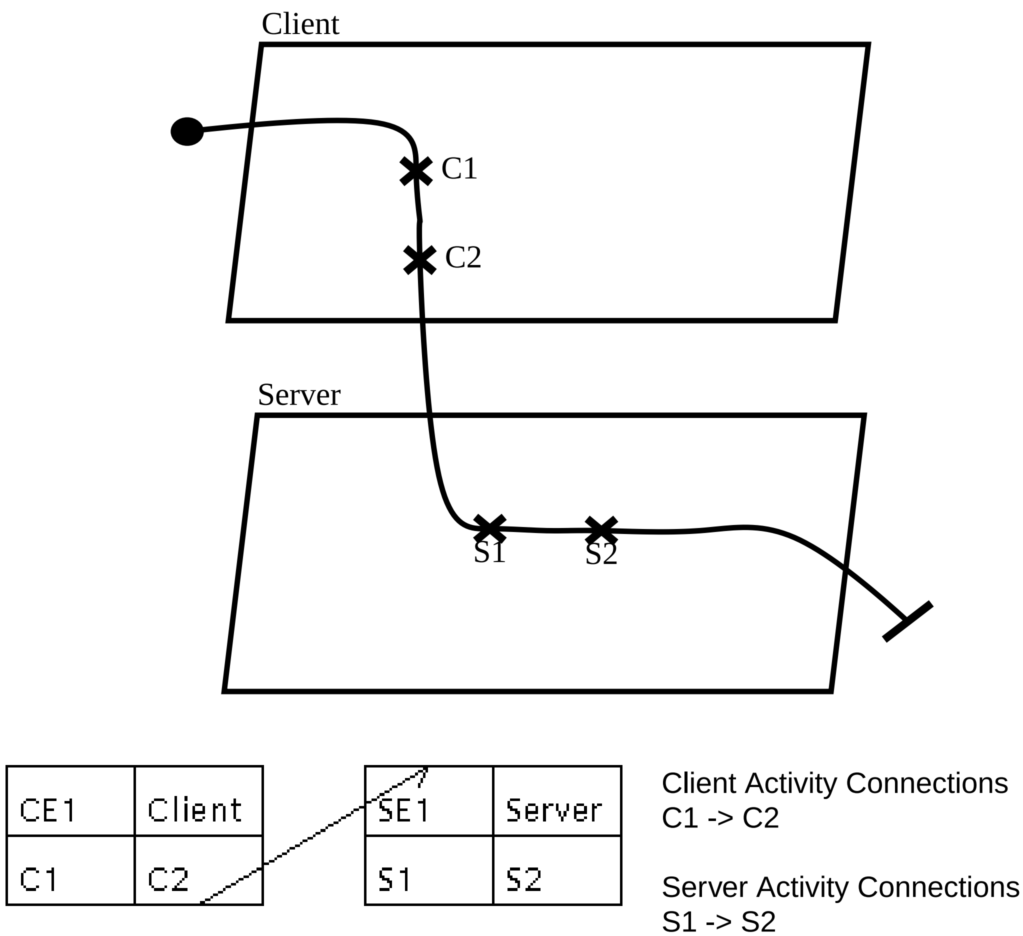

Figure 3-3: Corresponding UCM and LQN models for a simple synchronous call and return.

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 3-6 shows the corresponding UCM and LQN models for this case.

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 3-7 shows the corresponding UCM and LQN models for a synchronous interaction with a loop in the server.

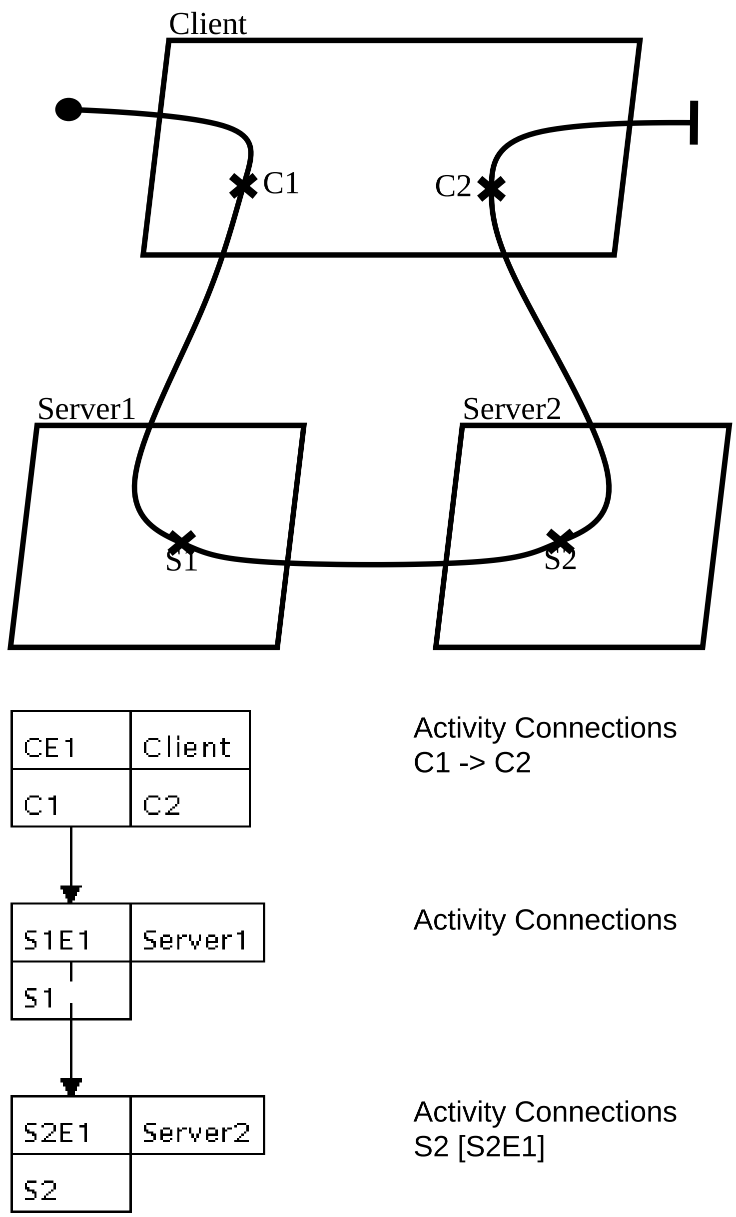

Figure 3-4: Corresponding UCM and LQN models for a forwarded synchronous call and subsequent return.

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.

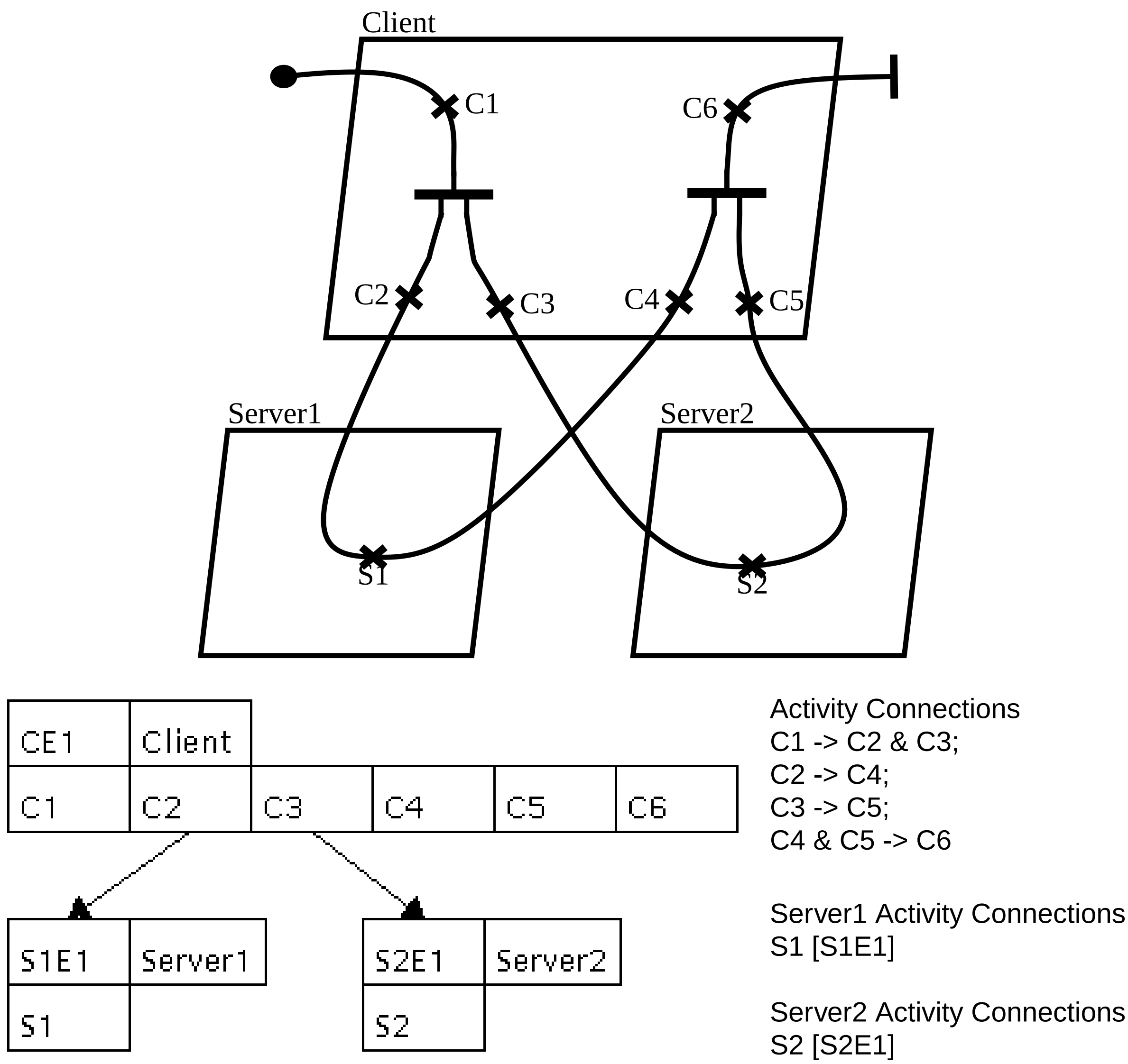

Figure 3-5: Corresponding UCM and LQN models for parallel synchronous calls and returns.

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 3-8.

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

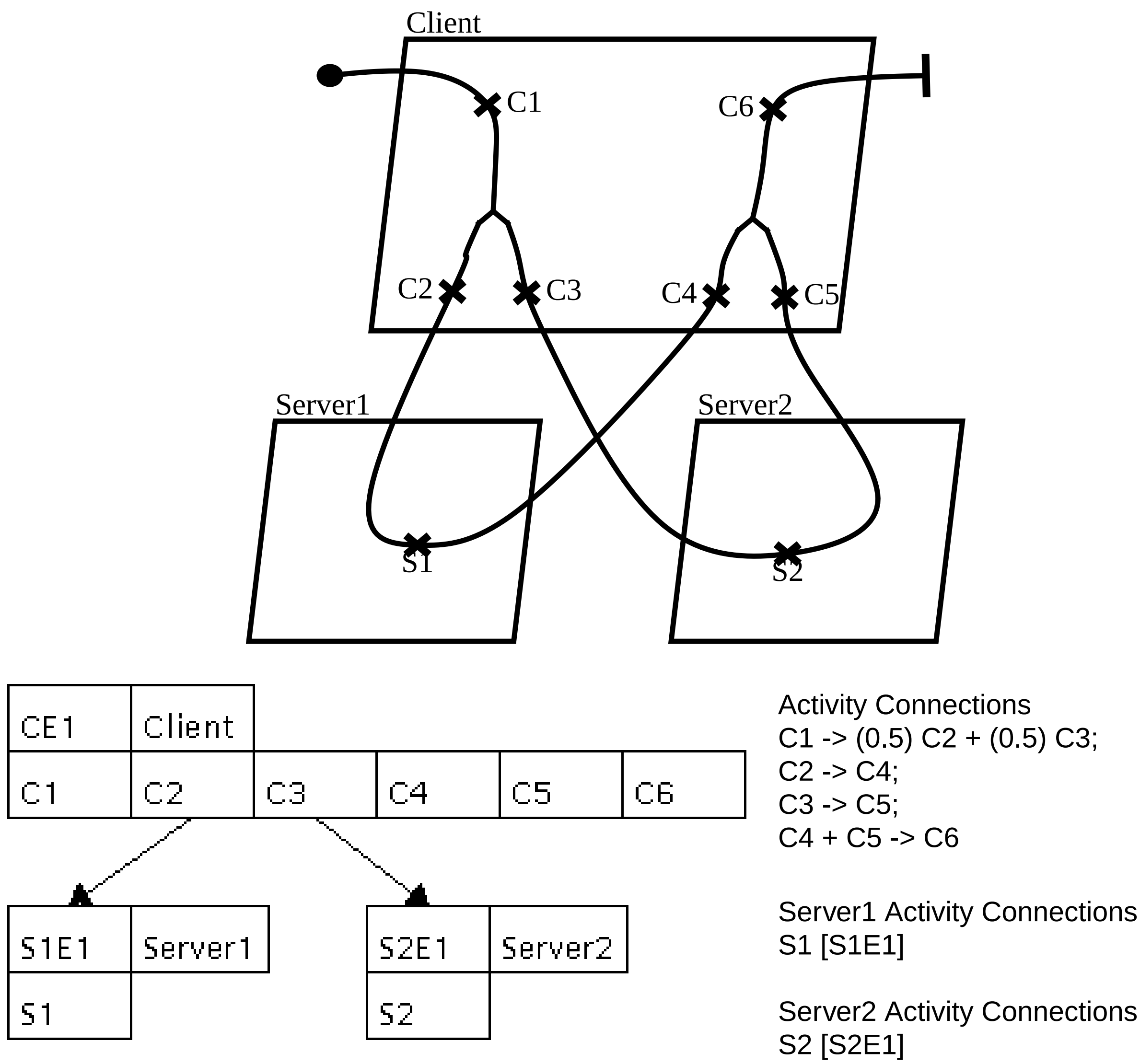

Figure 3-6: Corresponding UCM and LQN models for alternative synchronous calls and returns.

semantics of such a model since it does not break down into a tidy analytical solution. The ParaS-RVN 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 undesireable. A better solution is to create equivalent LQN models which do not have distributed forks and joins.

The example system shown in Figure 3-8 has a client Task_A making a synchronous ser-

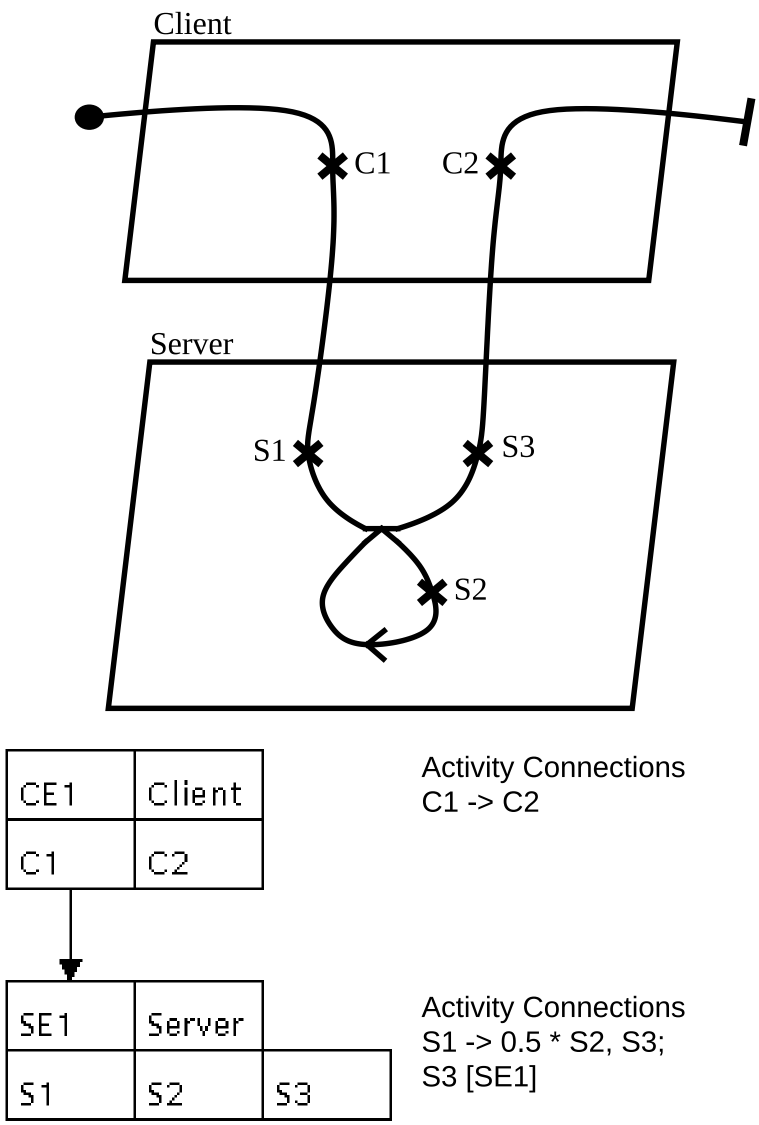

Figure 3-7: Corresponding UCM and LQN models for a loop.

vice 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

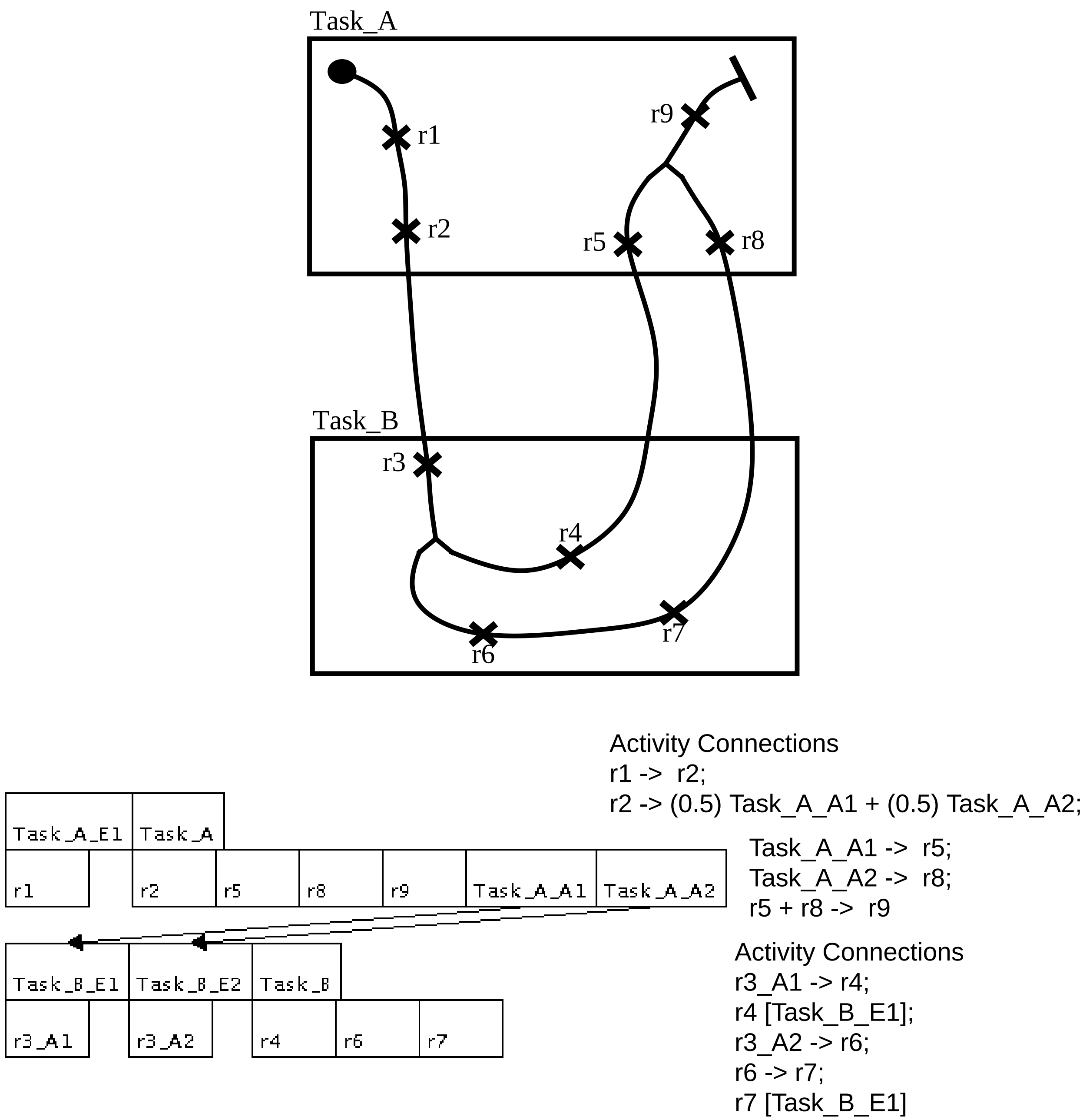

Figure 3-8: UCM and LQN models including an OR fork and join in separate tasks.

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 3-9 shows an example system with is a loop with multiple activities in the loop body.

The UCM in Figure 3-9 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 modelled 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.

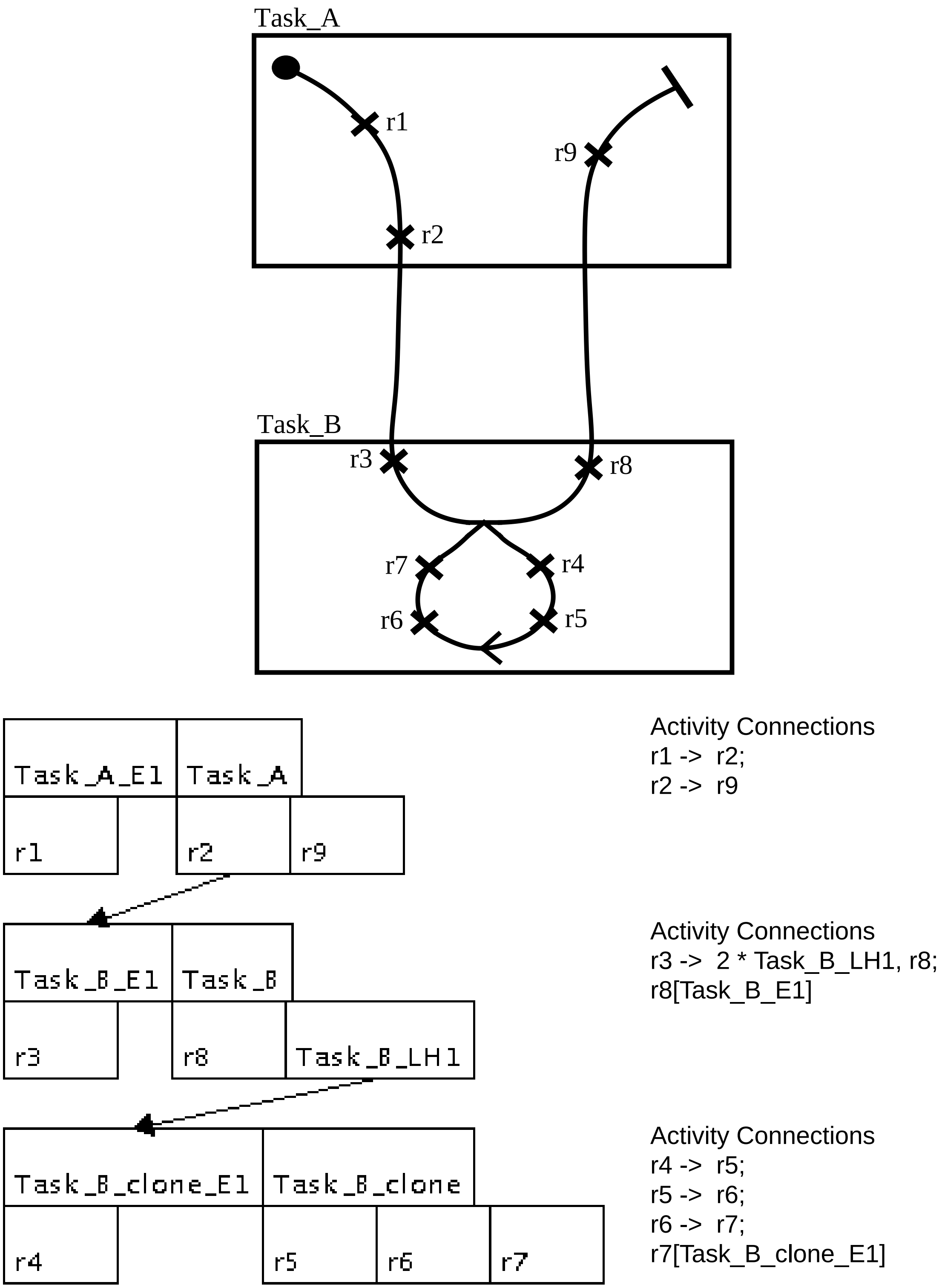

Figure 3-9: UCM and LQN models including a complex loop.

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. Franks' work did provide a suitable staring point for the research effort described in this thesis.

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. A complete class inheritance hierarchy of the UCMNav with the integrated UCM2LQN converter is shown in ***APPENDIX.

4.2.1. Design

The UCMNav can be said to have 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

UCM paths and path elements are represented by a hypergraph model in the UCMNav. A hypergraph is a sort of directed graph-in-reverse. It is composed of edges, also called hyperedges, and 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 some cases the figure class and the logical class will have a direct relationship where one or both

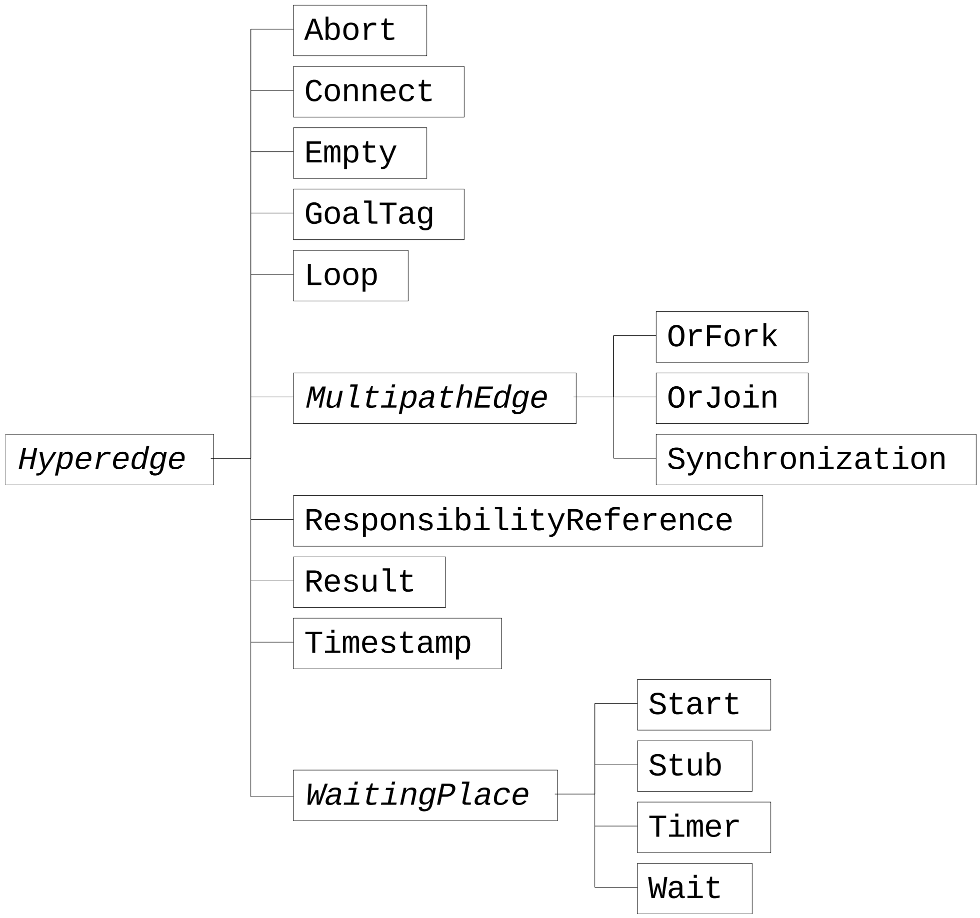

Figure 4-1: Inheritance hierarchy for the classes derived from *Hyperedge* (obtained using the WindRiver SNiFF+ code browser).

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 *Paths*, a collection of *ResponsibilityReferences*, a collection of *HyperedgeFigures*, and a pointer to

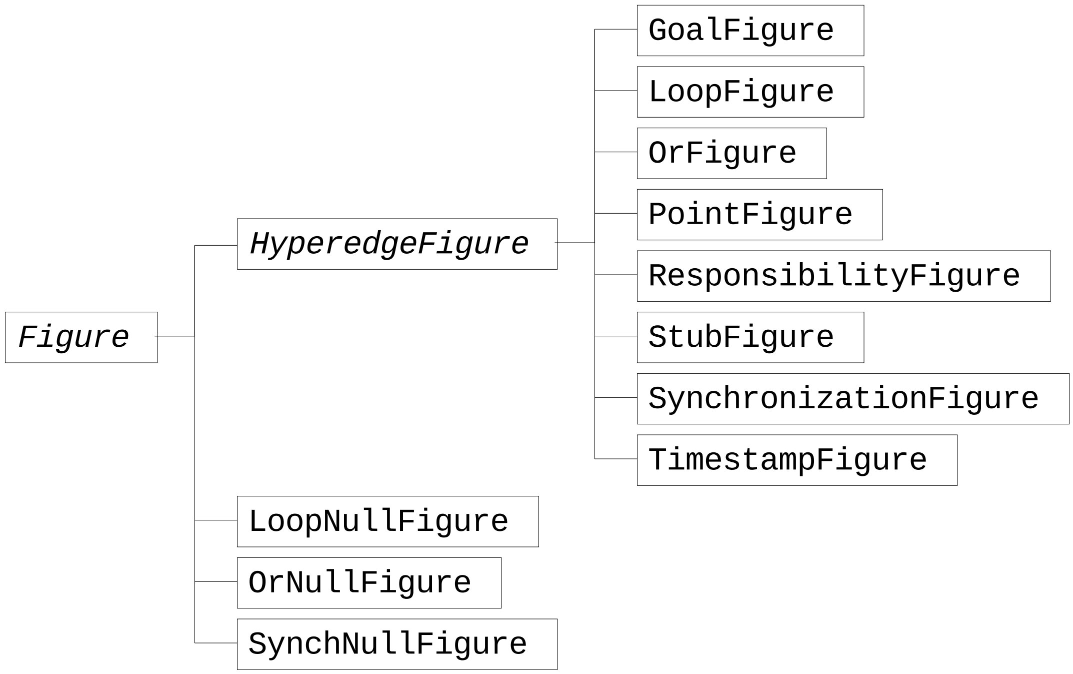

Figure 4-2: Inheritance hierarchy for the classes derived from *Figure* (obtained using the WindRiver SNiFF+ code browser).

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

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

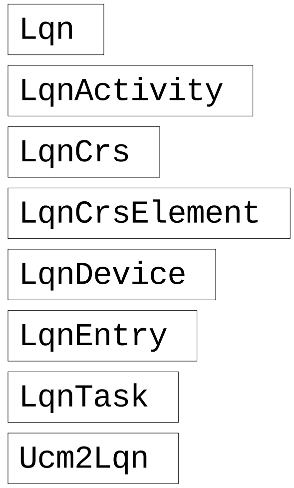

Figure 4-5: Inheritance hierarchy for the UCM2LQN LQN classes (obtained using the WindRiver SNiFF+ code browser).

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)

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 col-

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

lection 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 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 used to generate an LQN model from a UCM. The algorithm is broken down in the following sections:

- accessing the hyperedges sequentially
- path traversal
- identifying component boundary crossings
- determining the calling relationships between the components
- creating the LQN objects corresponding to UCM path elements
- manipulating the Call and Response Stack (CRS)

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 for a given hyperedge.

5.1.1. Getting the Next Hyperedges

There may be mutliple next hyperedges for a given hyperedge. As such 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 mutliple previous hyperedges for a given hyperedge. As such 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. Path Traversal Algorithm

A UCM path is traversed from a *current_edge* to a *next_edge*, without being concerned about the specific type of hyperedge. If the *current_edge* has *multiple next_edges*, then the *first next_edge* is arbitrarily assumed to be part of the main path and is skipped while the rest are followed. The *first next_edge* is then only followed after all the other *next_edges* have been exhausted. This is shown in greater detail in Section 5.5. A traversal is assumed to begin after the LQN objects corresponding to a start point have been created. The general path traversal algorithm is as follows:

```
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

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 from the figure for edge1
2. get enclosing component comp2 from the figure 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

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.

5.4.1. Handlling 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
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
   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*
   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
      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*

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. 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. The algorithm 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
      2.1.2. assign the *start_point* to a separate reference task running on an infinite processor
         2.1.2.1. *reference_task* = new reference task
         2.1.2.2. *reference_entry* = new entry for *reference_task*
         2.1.2.3. *reference_activity* = new default activity in *reference_task* for *start_point*
         2.1.2.4. connect *reference_activity* as the first activity of *reference_entry*
         2.1.2.5. push *reference_activity* on CRS
      2.1.3. **if** *start_point* is contained in a component **then**
         2.1.3.1. *first_task* = task corresponding to the component
         2.1.3.2. *first_entry* = new entry for *first_task*
         2.1.3.3. update *reference_activity* as making a call to *first_entry*
         2.1.3.4. push *first_entry* on CRS
         2.1.3.5. *first_activity* = new default activity in *first_task*
         2.1.3.6. connect *first_activity* as the first activity of *first_entry*
      2.1.4. continue path traversal (see Section 5.2.)
      2.1.5. **break**
   2.2. **case** *responsibility*
      2.2.1. **if** *responsibility* has not been visited **then**
         2.2.1.1. mark *responsibility* as visited
         2.2.1.2. **if** *responsibility* is contained in a component **then**
            2.2.1.2.1. *responsibility_task* = task corresponding to the compo-

nent
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.)
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**
2.3.1.1. mark *or_fork* as visited
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

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 <u>branch path</u>
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*
2.3.1.2.9. *branch_crs* = new CRS
2.3.1.2.10. set *branch_crs* as being on the <u>main path</u>
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.)
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
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 <u>branch path</u>
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*
2.3.1.3.11. *branch_crs* = new CRS
2.3.1.3.12. set *branch_crs* as being on the <u>main path</u>
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**
2.4.1.1. mark *or_join* as visited
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*
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**
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*
2.4.2.2. **else**
2.4.2.2.1. get *or_join_task* corresponding to the 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*
2.4.2.3. **if** CRS on the <u>main path</u> **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**
2.5.3.2.1. mark *and_fork* as visited
2.5.3.2.2. **if** *and_fork* is contained in a component **then**
2.5.3.2.2.1. *and_fork_task* = task candresponding 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* fand the loop body
2.5.3.2.2.3. *and_fork_activity* = new default activity in *and_fork_task* fand *and_fork*
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* fand *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
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 <u>branch path</u>
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*
2.5.3.2.2.9. *branch_crs* = new CRS
2.5.3.2.2.10. set *branch_crs* as being on the <u>main path</u>
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.)

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 fand *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* fand *and_fork*

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* fand *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

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 <u>branch path</u>

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*

2.5.3.2.3.11. *branch_crs* = new CRS

2.5.3.2.3.12. set *branch_crs* as being on the <u>main path</u>

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**

2.5.4.2.1. mark *and_join* as visited

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*
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**
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*
2.5.4.3.2. **else**
2.5.4.3.2.1. get *and_join_task* corresponding to the 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*

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.)
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
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
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.)
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*
2.6.1.7. **else**
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
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*
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*
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.)
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*
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.)
2.8.2. **else**

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 <u>closed system</u>
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 <u>open system</u>
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
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 <u>closed system</u>
2.8.2.2.1.2. *first_task* = first task called by the reference task corresponding to the *start_point*
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
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*
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*
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 <u>open system</u>
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*
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**

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 model 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 number of simulation runs. The LQNS analytical solver can also be used to verify the syntax and semantics of the LQN output.

6.1. Plain Old Telephone System

The POTS call connection example described in this section is the basis of a larger UCM model for a telephony system that was originally developed by Daniel Amyot and myself in the summer of 1998 for the feature interaction detection contest organized with the occasion of that year’s Feature Interaction Workshop. 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 described using a collection of activity graphs, one general graph for POTS and a separate subgraph one for every feature in isolation, and a minimum of any other documentation. It was the task of the contestants to intepret the descriptions, integrate them into the telephony model, and detect potential feature interactions. The UCM model we developed was unfortunately not ready on time for submission to the contest, but it did prove a good case study for using UCMs to detect feature interactions, as well as testing the useability of early versions of the UCMNav.

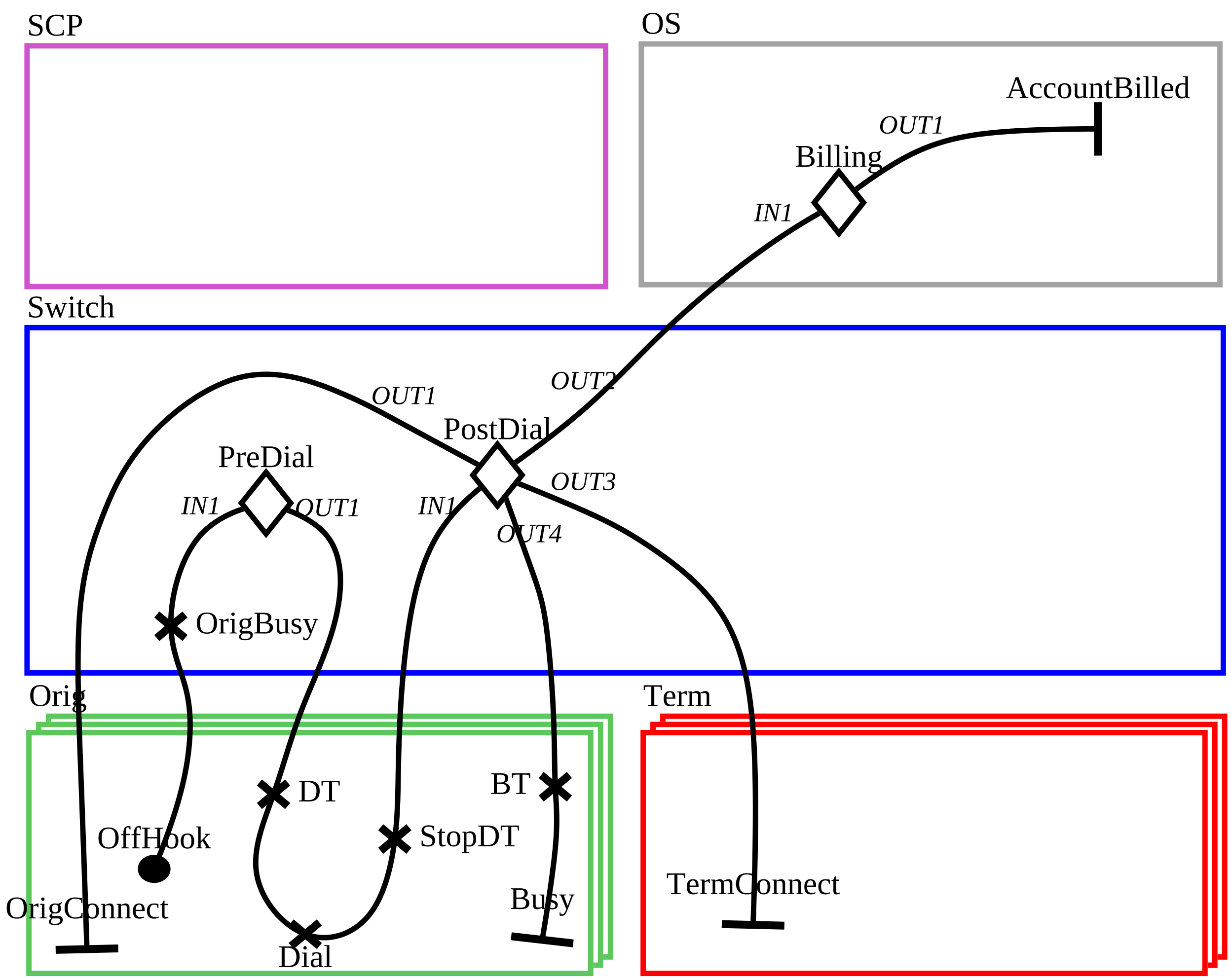

Figure 6-1: UCMNav root map for the POTS example.

6.1.1. POTS UCM Model

6.1.1.1. POTS 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 intended call recipient'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 responsiibility 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.

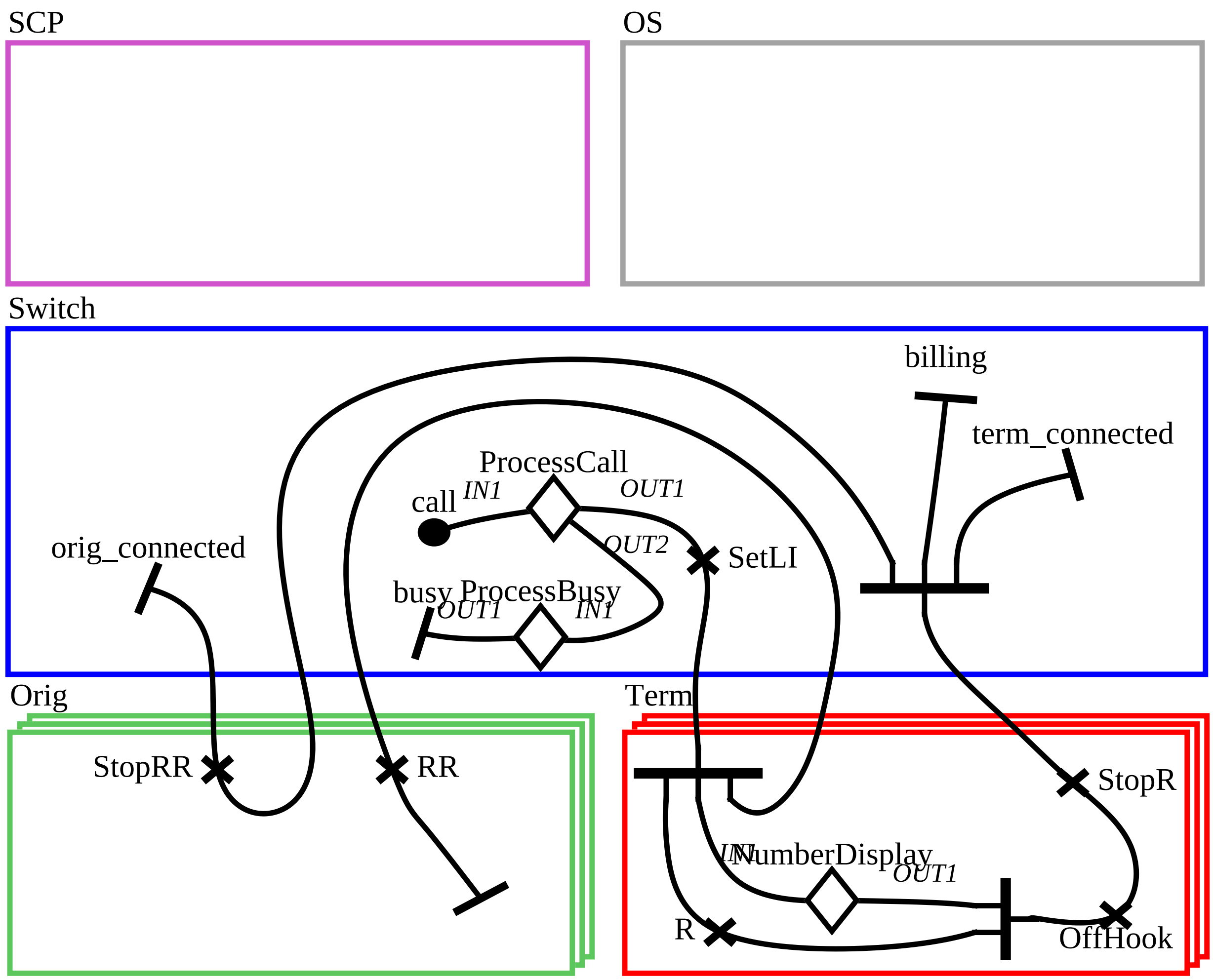

Figure 6-2: UCMNav plug-in map for the PostDial stub shown in Figure 6-1 for the POTS example.

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.

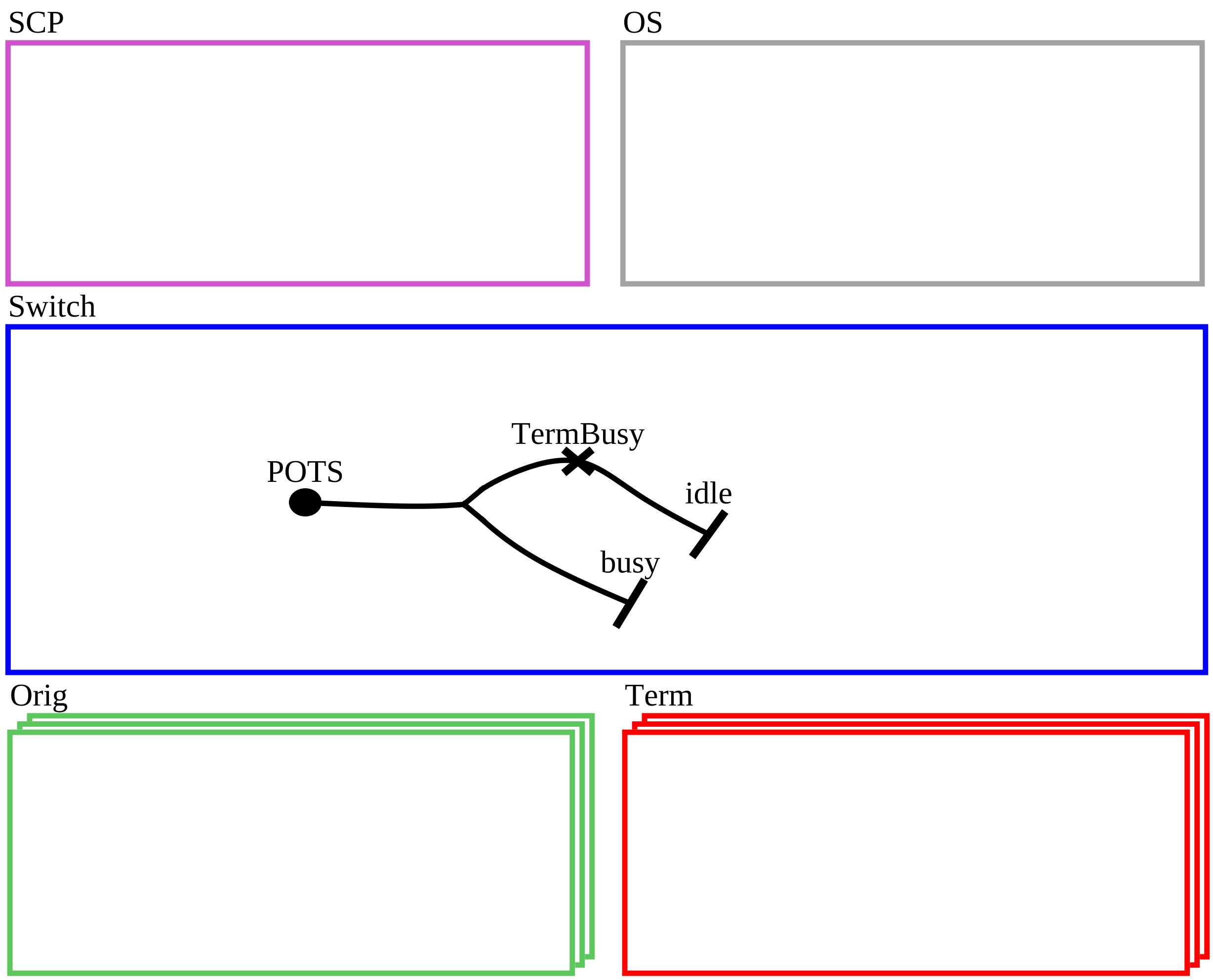

Figure 6-3: UCMNav plug-in map for the ProcessCall stub shown in Figure 6-2 for the POTS example.

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 for the POTS model is shown in Figure 6-4. The model incorporates 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.

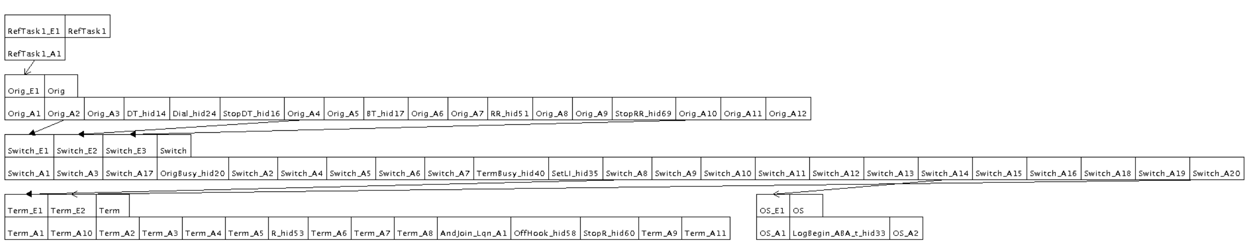

Figure 6-4: POTS LQN generated by the UCM2LQN converter from the UCM shown in Figure 6-1. (output from jLqnDef)

Figure 6-4 shows that a reference task was 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 reference 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* and requests a dial tone, then that a connection be established with the party whose number was entered, and finally that the creation be enabled after it was made successgully. The *Switch*, in turn, makes a synchronous call to the *Term* process to create the call, and then sends an asynchronous message to indicate that the call has been enabled and the caller and callee 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 calendar of events and buy tickets using a credit card. The TRS is one of the tutorial examples used as part of a course on the Design of High Performance Software and the UCM model is based on that tutorial example.[4]

6.2.1. TRS UCM Model

The UCM model for the TRS is shown in Figure 6-5. The components are as follows:

- *User*: TRS customer
- *WebServer*: web server that provides the interface to the TRS
- *Netware*: the underlying network
- *CCReq*: credit card verification and authorization server
- *Database*: database server

6.2.2. TRS Usage

The TRS can be used to either browse events by displaying the schedule and checking for ticket availability, or to buy tickets using a credit card. A typical scenario involves having the user log on to the system by making connection request. The web server registers the session and con-

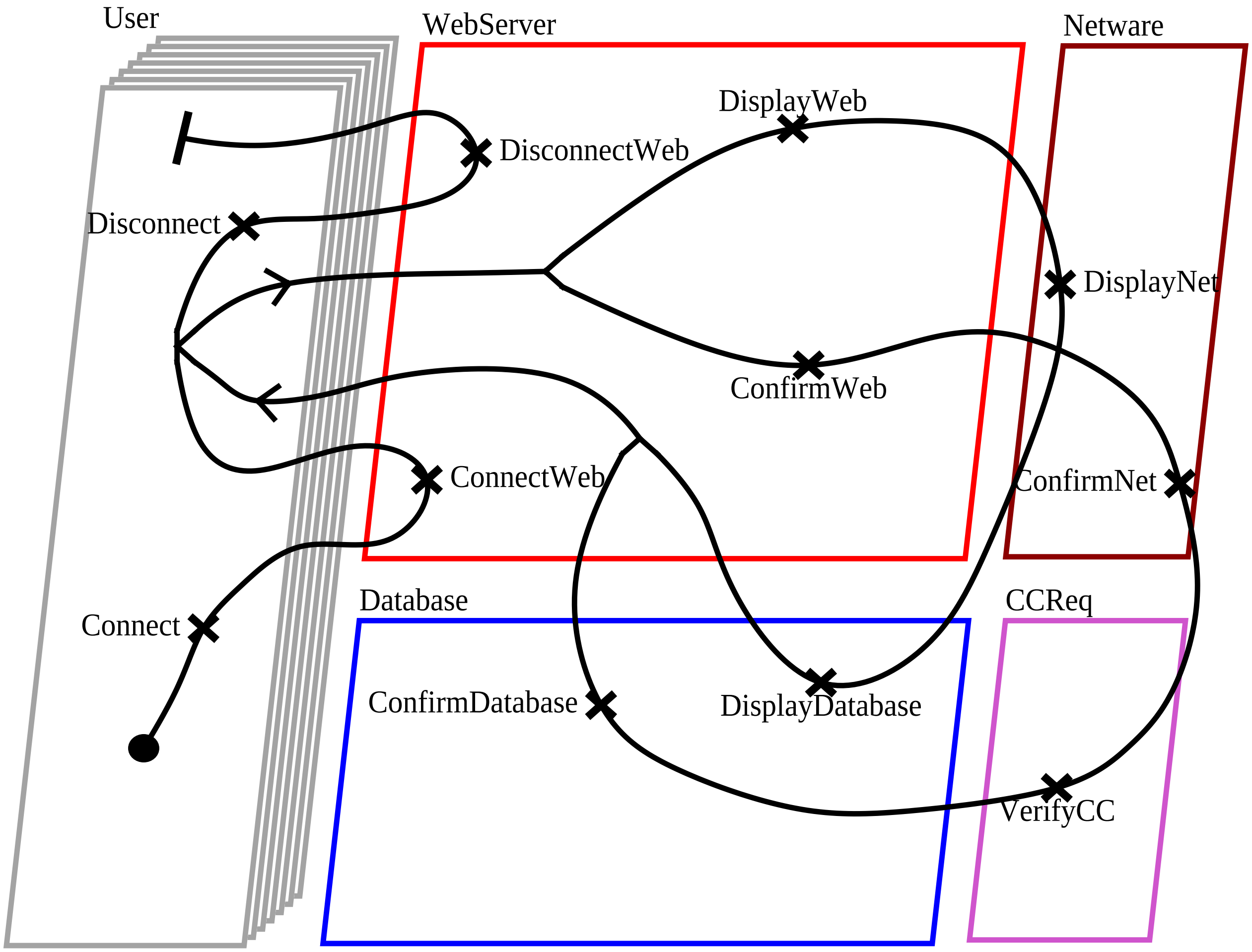

Figure 6-5: UCMNav map for the Ticket Reservation System example.

firms that the user is connected. Once she has connected to the system, the user enters a loop where she has two options. She can either request to browse and check information about an event, or she can buy a ticket. If the browsing option is chosen, the web server sends a request for the event information to the database through the netware. The databse is responsible for delivering the requested data back to the web server. The information is then displayed back to the user. If the user wishes to burchase a ticket, she can choose the buy option and supply a credit card number to which the ticket purchase can be billed. The web server then confirms the transaction by going through the netware and requesting that the credit card verification service verify the credit card information. Once the credit card is verified, the credit card service forwards the purchase request to the database so it may update its records. The transaction is now done and a confirmation is returned to the web server, which in turn relays it to the user. The user may browse or purchase tickets as often as she wishes. Once the user is done she can request to be logged out and the web server closes the session and confirms that the user has disconnected.

6.2.3. TRS LQN Conversion Results

The resulting TRS LQN is shown as jLqnDef graphical output in Figure 6-6. The LQN shows that synchronous calls and forwarding are tansformed 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 *WebServer_E2* entry in the *WebServer* task.

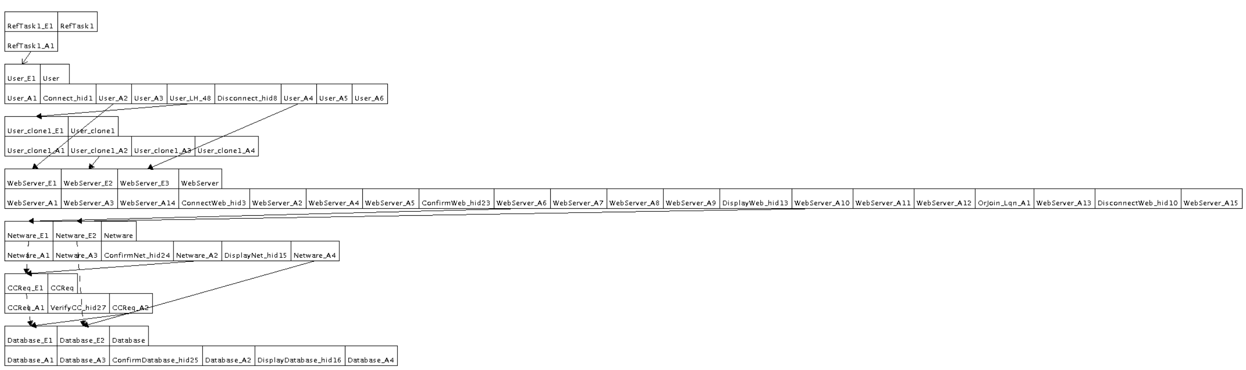

Figure 6-6: Ticket Reservation System LQN generated by the UCM2LQN converter from the UCM shown in Figure 6-5. (output from jLqnDef)

The resulting LQN for the TRS can be solved with LQNS as well as simulated with ParaSRVN. 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) is an example developed as a test for PERFECT, a methodology to evaluate the feasibility of alternate software concurrency architectures.[2][3] The GCS is used to store documents and allow users later access to those documents. Users are registered with the GCS and can subscribe to a set of documents, submit new documents, or update documents. If a document is updated, the GCS then notifies any subscribed users in case they wish to request the latest version.[3]

6.3.1. GCS UCM Model

Several UCM models for different software concurrency architectures are available for the GCS. The model chosen for this example is the one with the maximum amount of parallelism.[2] The UCM root map for the GCS is shown in Figure 6-7. The components are as follows:

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

The GCS root map has a writeFile stub which is bound to the plug-in map shown in Figure 6-8.

6.3.2. GCS Usage

The GCS supports five usage scenarios as follows [2]:

- updating a document.
- 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 an updated copy of a document that is already stored on the server. The GCS then proceeds to save the document to disk and in parallel with the save also notify the subscribers of the new update. The notification process involves preparing a notification message, making a temporary copy of the subscriber list, and then looping through

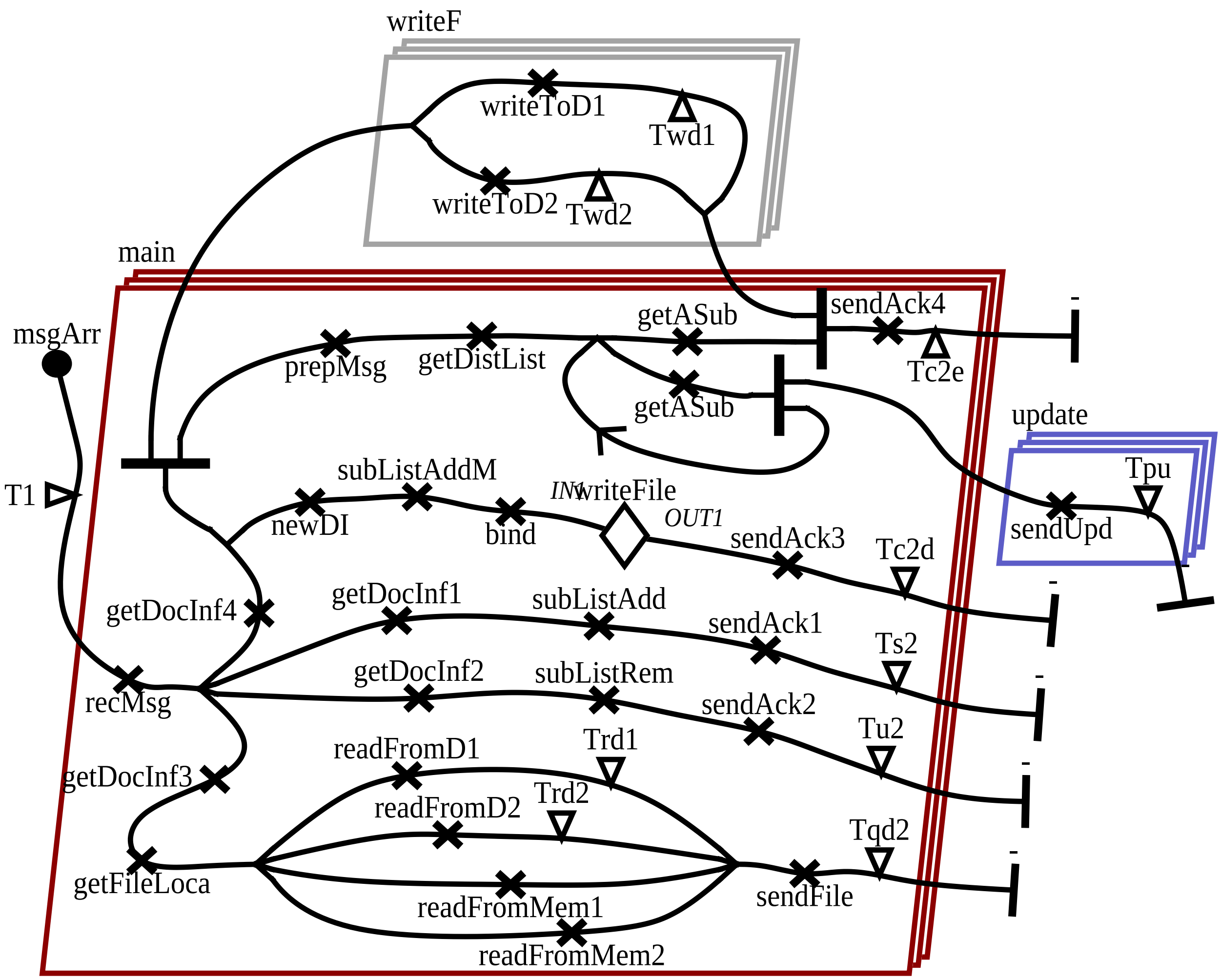

Figure 6-7: UCMNav root map for the GCS example.

the list and dispatching an update notification to each subscriber.

In the submitting a new document scenario, the user submits a new document which is then associated with a new subscriber list. The user is added as the first subscriber to the list and the document is added to the index of documents available on the server. Finally, the document is saved to disk and an acknowledgement is sent back to the user.

If a user wants to subscribe to a document the list of subsribers for that document is retrieved and the user is is added to it. An acknowledgement is then sent back to the user. The unsubscribing from a document scenario unfolds in much the same way, except the user is removed from the subscriber list instead of being added to.

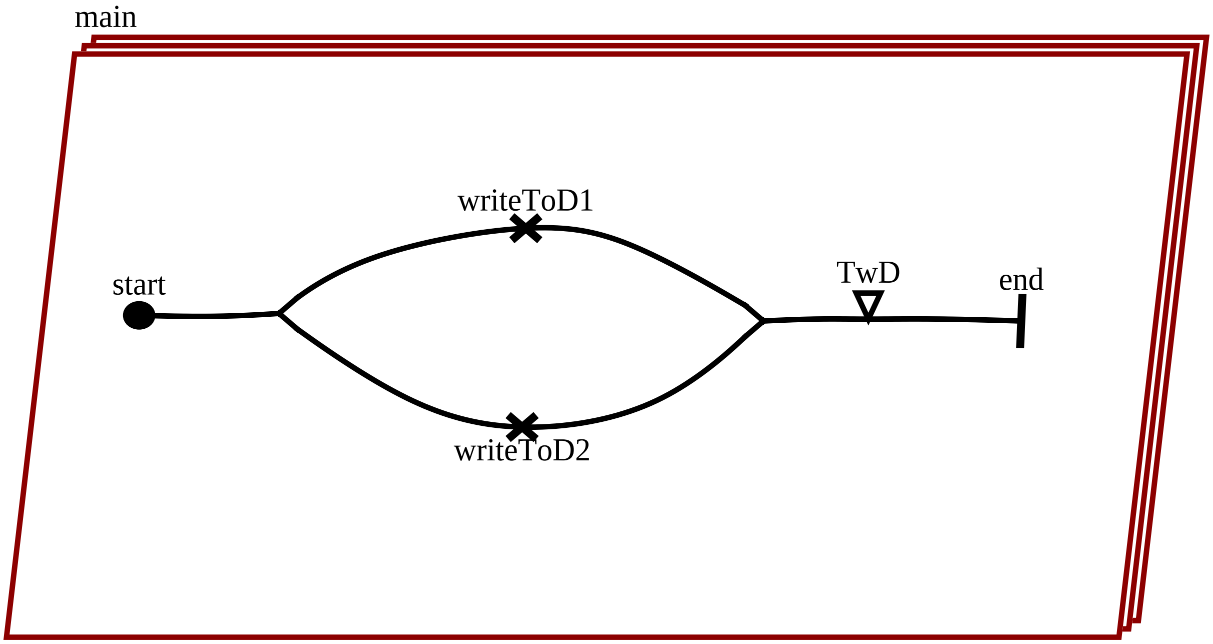

Figure 6-8: UCMNav plug-in map for the WriteFile stub shown in Figure 6-7 for the GCS example.

In order to retrieve the most recent version of a document, the information associated with the document is checked and then the document is read either from one of the disks or from a memory cache. The file is then sent out to the user.

6.3.3. GCS LQN Conversion Results

A partial view of an early LQN conversion of the GCS is shown in Figure 6-9. The jLqn-Def tool has a limit on the number of activities that it can handle, and the latest version of the GCS model contains 64 activities in the *main* task alone. jLqnDef runs out of space in its internal dictionary when attempting to load this model. An UCM2LQN output file lising for this example is included as an appendix in instead. *** The partial view in Figure 6-9 does provide an idea of how the system is layered and what kind of calls are made.

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.

Figure 6-9: Partial view of the GCS LQN generated by the UCM2LQN converter from the UCM shown in Figure 6-7. (output from jLqnDef)

Chapter 7 - Application

This chapter shows two example systems used to test the UCM2LQN converter. The examples are that of a wireless call delivery and a distributed hand-off protocol. The UCM models for these systems are based on industrial examples and did not originate at Carleton University. These examples provide good testing material since they do not necessarily conform to an "in-house" style of creating UCMs.

7.1. WIN Call Delivery

The wireless call delivery system shown by the UCM root map in Figure 7-1 and the CallSetup plug-in map Figure 7-2 originates from research into Wireless Intelligent Networks (WIN) standards conducted by a local telecommunications company. This is an example of an industrial-stlye UCM that is used to describe a real system.

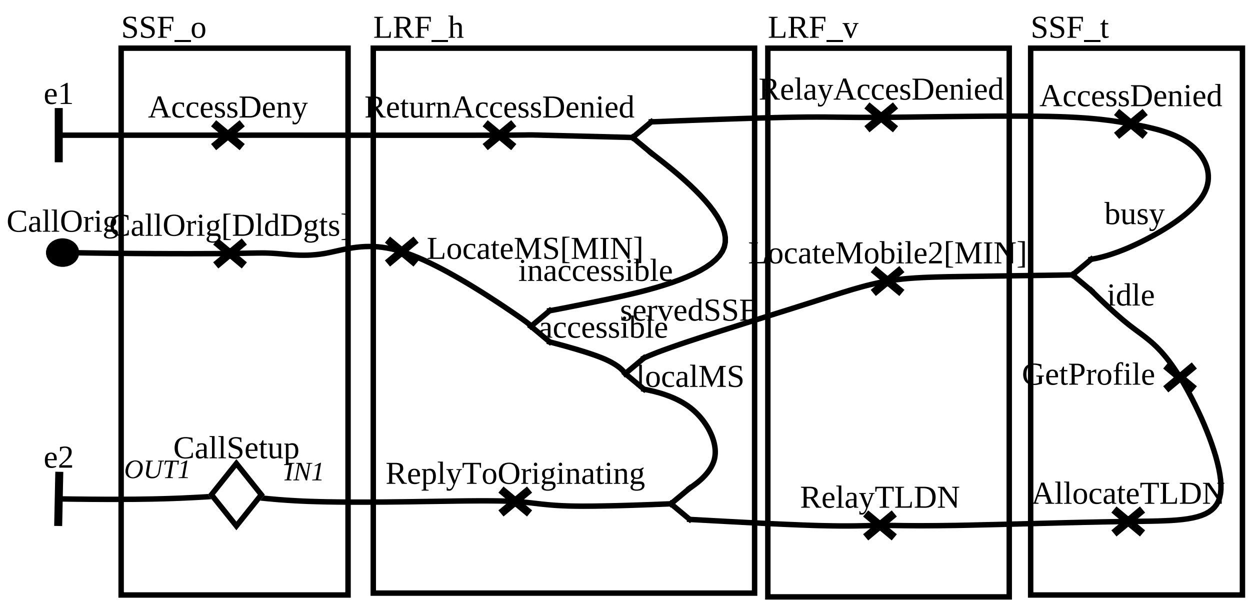

Figure 7-1: UCMNav root map for the WIN call delivery example.

The WIN example was not supplied with any additional documentation for this testing exercise. Therefore the call delivery scenario was simply inferred from the UCM model. 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 timeout 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.

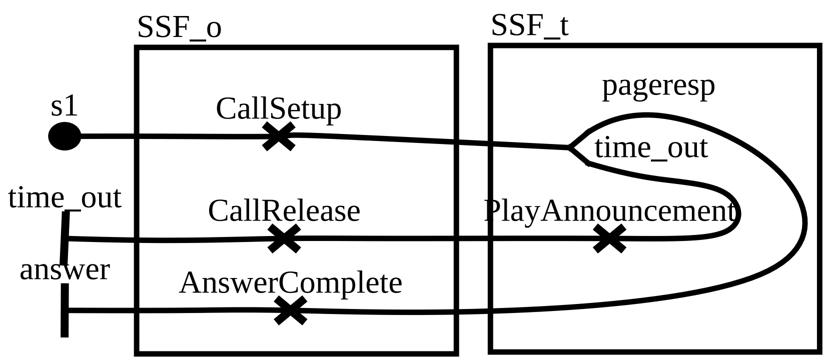

Figure 7-2: UCMNav plug-in map for the CallSetup stub shown in Figure 7-1 for the WIN call connection example.

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 the original UCM.

7.2. Example Of A 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 resource for 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.

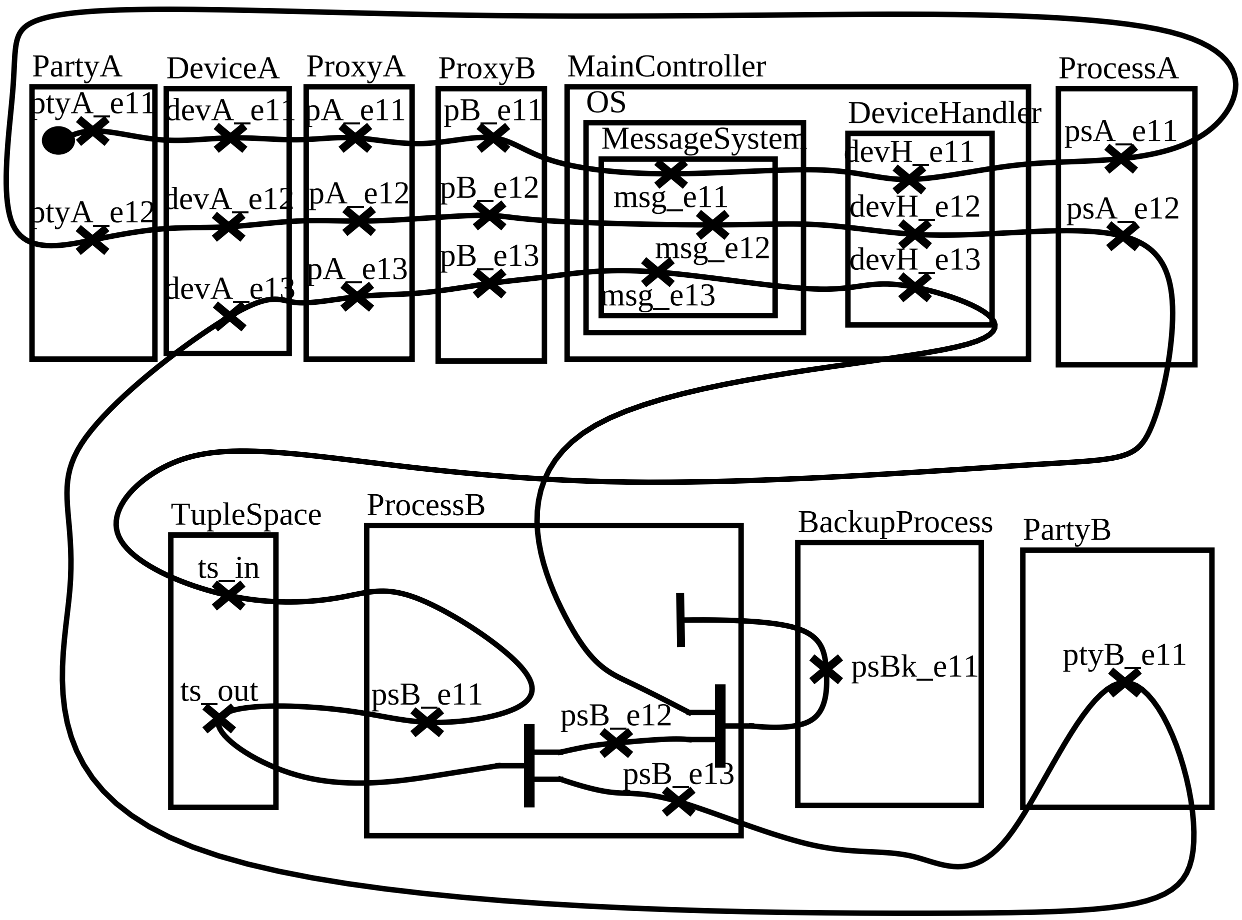

Figure 7-5: UCMNav root map for the distributed hand-off protocol example.

The protocol described in this model is used to coordinate a hand-off 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 *** and is used as a means to coordinate the communication between the other system participants.

The scenario illustrated in Figure 7-5 illustrates the transfer 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*. A *ts_in* responsibility is performed in the

TupleSpace before *ProcessB* is reached. After *ProcessB* performs some operation, the *TupleSpace* performs a *ts_out* responsibility and returns to *ProcessB*. The execution then splits off into two parallel paths at *ProcessB*. One of the parallel paths traverses *PartyB*, *DeviceA*, *ProxyA*, *ProxyB*, the *MessageSystem* and *DeviceHandler* contained in the *MainController*, before returning to *ProcessB*. The other parallel branch remains in *ProcessB* and executes some responsibility. The two parallel paths are then rejoined into a single path and a call is made to the *BackupProcess*.

7.2.1. Distributed Hand-Off Protocol LQN Conversion Resuslts

In order to generate a UCM2LQN output file that could be read by the LQN tools, it was necessary to modifiy 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 provide a solution for 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 numberous asynchronous, synchronous, and forwarding calls and replies that take place in the example.

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 thefunctioningt 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.

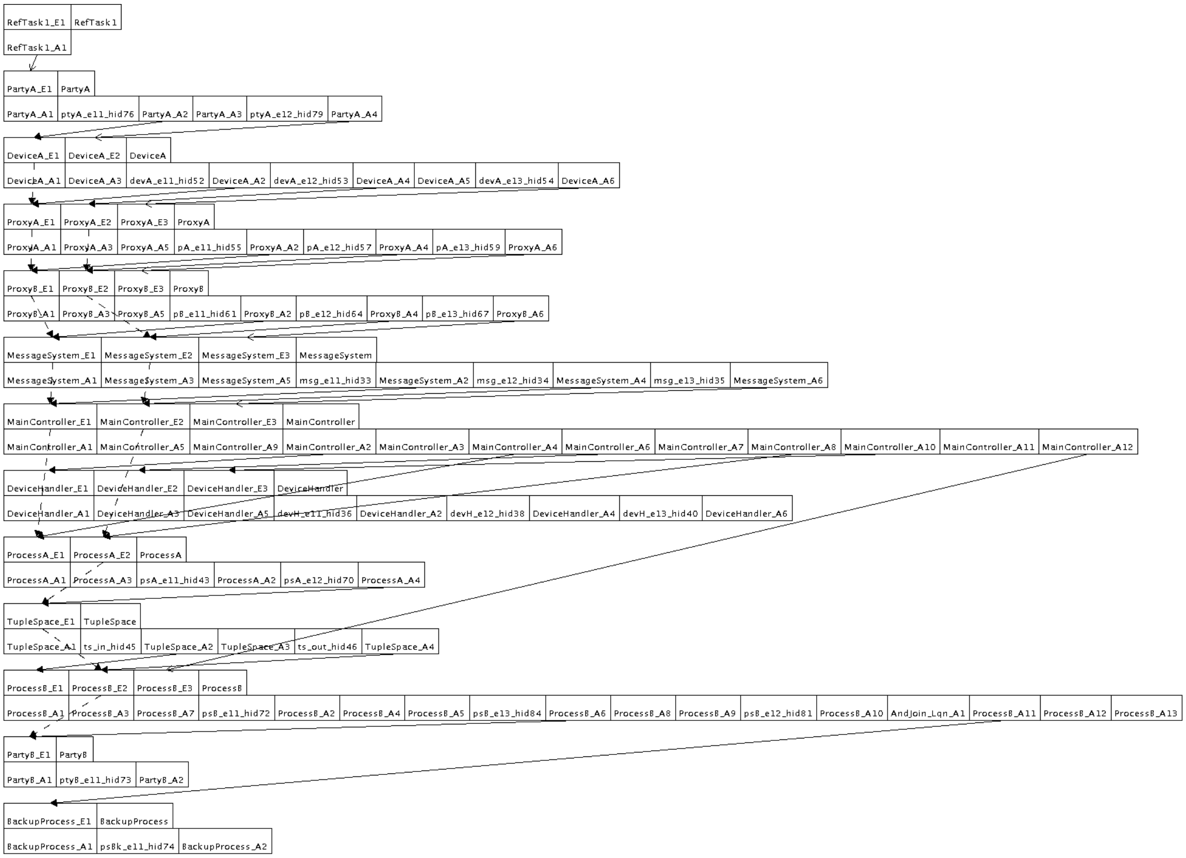

Figure 7-6: LQN for the distributed hand-off protocol generated by the UCM2LQN converter from the UCM shown in Figure 7-5. (output from jLqnDef)

Chapter 8 - Conclusions

8.1. Contributions

The major 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 sover 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 crossinge, and creating the apropriate 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 need to pay attention the naming 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 due to the same limitation of LQNS in solving distributed forks and joins.

8.2.1. Converter Limitations

8.3. Future Research

Chapter 9 - References

[1]

[2] Craig Scratchley, "Evaluation and Diagnosis of Concurrency Architectures", Report OCIEE-00-07, PhD thesis, Carleton University, Ottawa, Sept 2000, pp 92-125

[3] C. Scratchley, C. M. Woodside, "Evaluating Concurrency Options in Software Specifications", Proc 7th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, Md., October 1999, pp 330 - 338

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

[5]