Enhanced Use Case Map Traversal Semantics

Jason Kealey and Daniel Amyot

SITE, University of Ottawa, Canada
jkealey@shade.ca, damyot@site.uottawa.ca

Abstract. The Use Case Map (UCM) notation enables the use of graphical scenarios to model grey-box views of a system's operational requirements and behaviour, in context. The scenario traversal mechanism is the most popular UCM analysis technique but its current tool support in UCMNav is limited and hard to use, and the high-coupling of its features makes it difficult to maintain and evolve. This paper presents major enhancements to the recent jUCMNav Eclipse plug-in consisting of a new scenario traversal semantics accompanied by enhanced trace transformations to Message Sequence Charts. In addition, this paper identifies a set of semantic variation points which lay the groundwork for notational clarifications and user-defined semantic profiles.

1 Introduction

The *Use Case Map* (UCM) notation [5] is a part of the proposal for ITU-T's User Requirements Notation (URN) [1, 10]. UCMs visually model operational scenarios cutting through a system's component structure, providing a high-level, grey-box view of system behaviour in context. Because of their visual nature and apparent simplicity, UCMs are quickly understood by many stakeholders. Furthermore, UCMs are useful in various development phases such as requirements modelling and analysis, test case generation, performance modeling, and business process modelling, and this in numerous application domains[1].

Among the techniques used to analyze and transform UCM models, the *scenario traversal mechanism* is likely the most popular and best supported one. This mechanism essentially provides an operational semantics for UCMs based on an execution environment. By providing an initial context, called *scenario definition*, the traversal mechanism determines which scenario paths of the UCM model will be followed, until no progress is possible. There are many typical applications of such traversal semantics, including:

- **Model understanding and scenario visualization**: complex UCM models involve many paths and diagrams that invoke one another. The traversal can highlight which scenario paths are followed in a given context (e.g., see Figures 1 and 2). In addition, the traversed paths can be visualized in a linear form, e.g. by transforming them to Message Sequence Charts (MSC) [9], hence avoiding the need to flip back and forth through many diagrams.

[1] The UCM Virtual Library, http://www.UseCaseMaps.org/pub/, contains a collection of over 140 papers and theses illustrating these topics.

- **Model analysis**: scenario definitions act like test cases for the UCM model itself and enable the detection of unexpected behaviour (deadlocks, races, interactions, etc.) as well as the regression testing of evolving models.
- **Test goal generation**: once validated, the scenarios extracted via the traversal mechanism can serve as a basis for design/implementation-level test goals, e.g., in MSC, UCM sequence diagrams, or TTCN-3 format.
- **MDA-like transformations**: the traversal mechanism can use platform-dependent information sources on top of UCM models and scenario definitions in order to generate partial design models (e.g., in MSC or UML).

UCMNav is a UCM modelling, analysis, and transformation tool developed over the past decade. Though it includes a scenario traversal mechanism [2, 3] and transformation procedures to various target languages (including MSCs [13]), it suffers from major limitations and usability, extensibility, and maintainability issues. jUCMNav, its Eclipse-based successor, is a complete re-implementation of the modelling tool which now supports URN in its entirety, i.e. UCM combined with the Goal-oriented Requirements Language (GRL) [16].

This paper introduces major analysis enhancements to jUCMNav by providing an extensible scenario traversal mechanism accompanied by trace transformations to MSCs. The new scenario traversal engine supports a more complex data model in addition to being designed for extensibility. Furthermore, this paper identifies a set of semantic variation points for which the behaviour is unclear in UCMs as well as potential alignment with common workflow patterns, laying the groundwork for notational clarifications and user-defined semantic profiles.

Section 2 introduces an example UCM model featuring an active scenario, setting the stage for the introduction of the new scenario traversal semantics in Section 3. Section 4 describes the new scenario traversal listener infrastructure which is used by the three-step MSC generation algorithm. Section 5 discusses related work and summarizes UCM semantic variation points; clearing up semantic issues is a necessary step towards future enhancements to the notation and jUCMNav. Section 6 finally presents conclusions and future work.

2 An Example Use Case Map Model with Scenarios

An example is used here to illustrate parts of the UCM notation and typical usage. It also emphasizes some of the complexities of the traversal mechanism and limitations of the current UCM notation. The interested reader can access more comprehensive tutorial material online[2]. Although the scenario traversal mechanism is only explained in Section 3, note that both figures in this section highlight a particular scenario in a different color (i.e., the active scenario).

Our sample UCM model describes an online e-commerce front-end to a warehouse selling physical products. The company's business processes do not allow it to show real-time product availability on its website; because this process is

[2] See `http://www.UseCaseMaps.org` and `http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/JUCMNavTutorials`.

manual, an unfortunate web customer can order a product that is not available in the warehouse. Should this occur, the web store will inform the user that his order includes back-ordered products. Consequently, the user will either decide to wait for the product to become available, cancel the back-ordered items, or cancel the order completely.

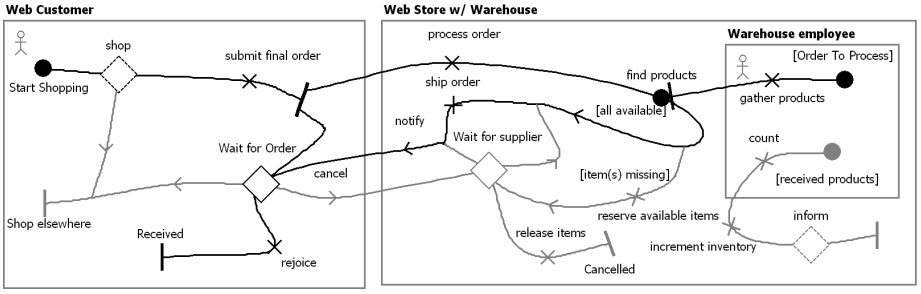

Fig. 1. Example Use Case Map with an active scenario

Figure 1 describes the top level process which can be read from the *start points* (filled circles) following the paths until *end points* (bars) are reached. Along the way, various path elements are encountered such as *responsibilities* (shown as X's), *Or-forks* (mutually exclusive alternatives), *Or-joins* (path merging), and *stubs* (diamonds). *Condition labels* on alternatives and pre/post-conditions are shown between square brackets (the logical conditions themselves are formalized using a data model). Stubs contain sub-maps (called *plug-ins*) and can either be static or dynamic (dotted outline). The former contains only one plug-in whereas the latter offers different possibilities and the one that is used is selected by the traversal engine depending on the stub's selection policy.

Figure 2 describes the Wait for Order plug-in map and illustrates a few other UCM constructs. The *And-fork* introduces the concept of concurrency in Use Case Maps and its counterpart (not used here), the And-join, is used for synchronizing paths. Fork and joins can be used independently and do not need to be well nested. Looping paths are also allowed. Another way to model synchronization in UCMs, which we use here, is to make use of the *waiting place* (filled circle on path) or the *timer* (clock icon). Although both path nodes block until a connected path arrives, the timeout has the added capability of following a *timeout path* (zigzag symbol) if the connected path never arrives. Section 5 will provide insights on how the UCM notation could be enhanced with additional workflow patterns to improve the readability and precision of this plug-in map.

Conceptually, our example scenario highlights the primary use case where the customer orders items which are in stock. The warehouse employee receives the order, gathers the products and ships them off to the customer.

In [2, 3, 13], scenario definitions are composed of a list of start points that are triggered in a given context (a set of variable initializations), and possibly

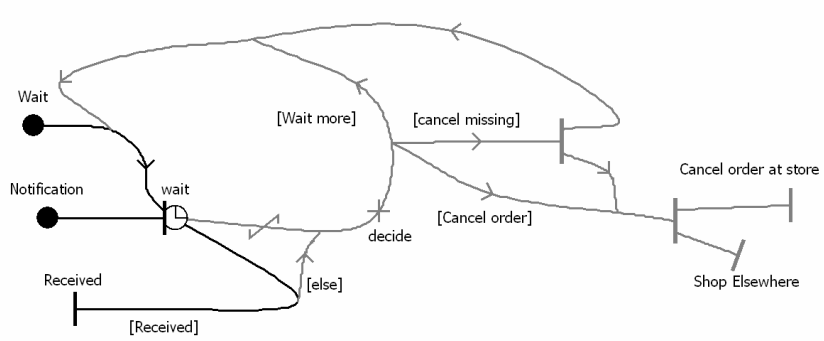

Fig. 2. Wait for Order plug-in map

with post-conditions required to be fulfilled at the end of the execution. Responsibilities are allowed to change the values of the variables. Once executed, the traversal results are visualized as a coloured path over the original UCM.

The UCMNav implementation of this mechanism suffers from many limitations. The only data type allowed is Boolean. Conditions and assignments are described using a very simple action language with a non-standard syntax. The same start points cannot be triggered multiple times. Scenario definitions are not reusable, leading to scalability and management issues. The traversal is combined with a trace linearization algorithm (for MSC generation) that is prone to errors; hence it is difficult to debug and maintain. The traversal is rigid, without semantic variation points and without tolerance for errors (it often blocks if something unexpected happens or is not initialized properly). Only one scenario can be run at a time. Finally, UCMNav has a dependency on external tools for MSC generation (e.g., UCMExporter [3]) and visualization (e.g., Telelogic Tau), hence hindering usability. The following sections will address these issues and discuss the new solution implemented in jUCMNav.

3 New Traversal Semantics

3.1 Data Model and Operators

This section describes the data model and operators now available in jUCMNav. jUCMNav supports Boolean variables, integer variables, and variables of user-defined enumeration types whereas UCMNav only supported Boolean variables. All variables are global in scope, which is more appropriate for requirements (targeted by URN) than implementation. jUCMNav supports more operators to work with these data types and although they remain very simple, the set supported by jUCMNav greatly improves expressiveness for conditions:

– **Integers {..., -2, -1, 0, 1, 2, ...}**: support for comparison (equals, not equals, greater than, less than, greater or equal to, less or equal to) and arithmetic operators (additive inverse, addition, subtraction, multiplication);

– **Booleans {true, false}**: support for comparison (equals, not equals) and logical (not, and, or, xor, implies) operators;
– **User-defined enumerations { {INITIAL, ACTIVE}, ... }**: support for definition and for comparison (equals, not equals) operators.

The concrete grammar is omitted here for simplicity but it should be noted that it does support the SDL syntax for these data types and operators[3]. The action language used to modify variables in responsibilities supports assignments and if-else statements on top of the operators listed above. The pseudo-code parser was automatically generated from a grammar in Backus-Naur Form (BNF) using JavaCC/JJTree [11]. Integers were not added with the intension of supporting complex mathematical computations; their main use in UCMs is to better support loop constructs which were previously represented using complicated sets of Boolean variables. As for enumerations, they are well adapted to Or-forks or stub selection conditions that have multiple possible branches.

3.2 Metamodel Enhancements

The URN metamodel (implemented in jUCMNav [16]) was enhanced to support scenario definitions. The relevant portion is presented as a class diagram in Figure 3. A UCM model can contain a set of scenario groups which, in turn, contain scenario definitions. A particular scenario definition is represented as:

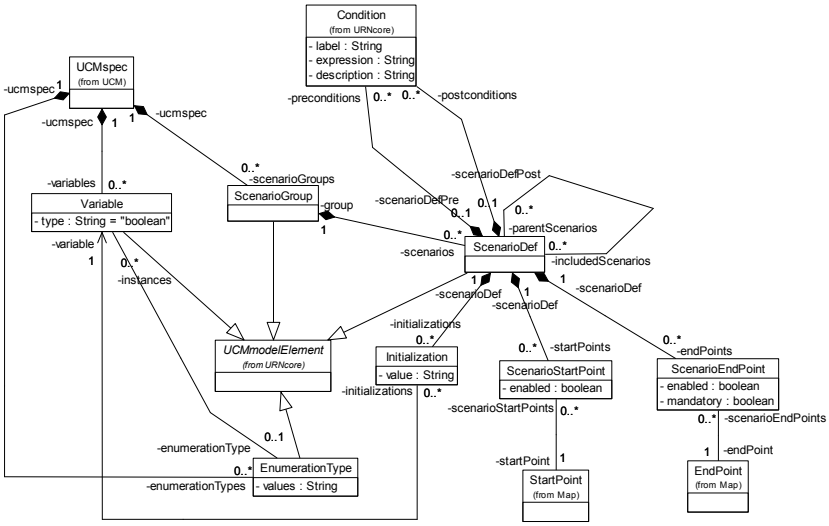

Fig. 3. UCM scenarios metamodel

[3] See http://jucmnav.softwareengineering.ca/twiki/bin/view/ProjetSEG/HelpOnLine

- An ordered list of scenario start points, where duplications are allowed;
- A set of variable initializations;
- A set of scenario post-conditions (logical conditions expressed using the language described in the previous section);
- A set of scenario preconditions;
- An ordered list of scenario end points that must be reached during execution;
- An ordered list of included scenarios (for reusability and management).

Only the first three of these elements were supported in UCMNav. Our contribution to this model is the support of additional elements that make UCM scenarios closer to test cases. The UCM modeller can now define where the traversal should end, to facilitate model verification. Furthermore, scenario inclusion now allows a modeller to reuse existing scenario definitions and incrementally build the test suite. In particular, default variable initializations can be defined in one central location and overridden if necessary in including scenarios. This is also useful when new variables are added to an evolving UCM model. All preconditions, post-conditions, start points, and end points are also included but cannot be overridden in the parent scenario: the union of these elements is always executed before the parent scenario's elements.

3.3 Architecture and Algorithm Overview

This section presents the scenario traversal engine's high-level architecture and algorithms. jUCMNav provides a default semantic interpretation of the various constructs according to common understanding and the draft standard, but as will be presented in Section 5, there are many issues that are up for clarification.

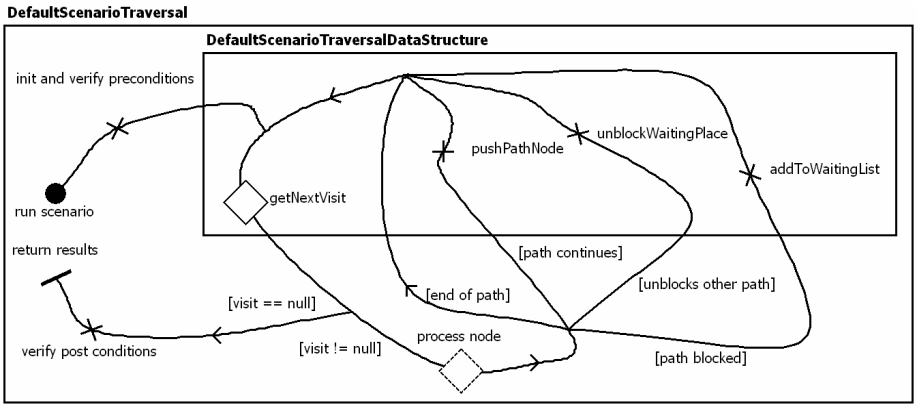

Fig. 4. Default Scenario Traversal

The default scenario traversal algorithm is architecturally separated in two: DefaultScenarioTraversal and DefaultScenarioTraversalDataStructure, as shown in

the UCM of Figure 4. The former defines the flow of control in the traversal algorithm and how each UCM path node should be processed, according to the default semantics of each path node. The latter encapsulates data structures such as the stack of path nodes that have to be processed and the waiting list (a queue of path nodes that cannot be processed at this time). By using a stack and a blocked node list, one defines a depth-first traversal algorithm. A breadth-first implementation could be trivially added by simply changing the stack to a queue inside the DefaultScenarioTraversalDataStructure.

The core of the traversal engine is represented here as the process node dynamic stub. Each path node is treated differently according to its type, related node connections, and related conditions. Section 5 will detail the ones that are particularly challenging. The traversal engine's second most important responsibility is to decide on the path node that should be executed next, which is defined in the getNextVisit stub and refined by the plug-in map shown in Figure 5.

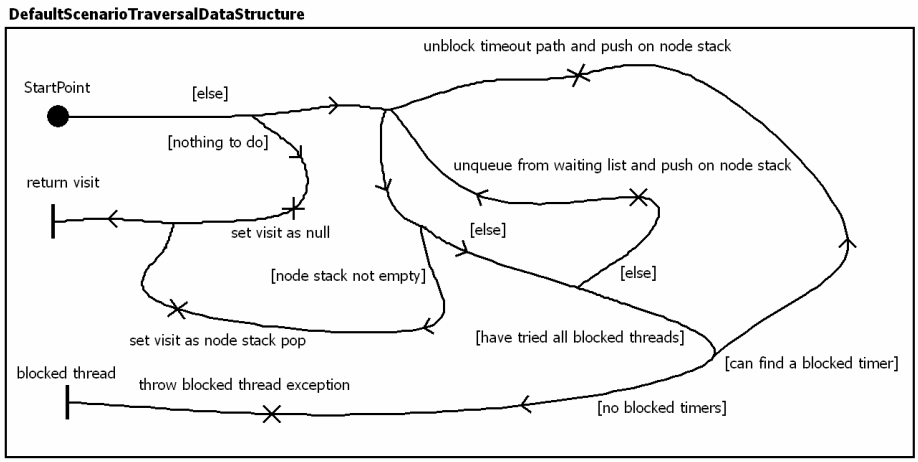

Fig. 5. Get Next Visit plug-in map

The default scenario traversal data structure uses a node stack and a waiting list internally: this depth first behaviour respects the general traversal behaviour presented in [2] that was implemented in UCMNav. Because concurrency is involved, it does keep track of pseudo-threads but the implementation itself remains single-threaded. As maintainability is one of the goals of this implementation, simplicity is key and multi-threading is left as future work.

jUCMNav's infrastructure opens the door for the creation of new algorithms by any modeller: our scenario traversal framework uses a low coupling strategy and the *chain of responsibility* design pattern [7], enabling tool builders to override the default traversal algorithm via a plug-in to jUCMNav.

3.4 Validation Methodology

To validate the correctness of the implementation, jUCMNav's set of unit tests was augmented significantly. For the parser aspects only, over one hundred tests were created, to verify the correctness of the BNF grammar given to the parser generator. Furthermore, another fifty tests that make use of the new scenario features were created to cover the base traversal cases. The automated tests focus on the low-level aspects of the scenario generation, for the most part. As for checking the high-level behaviour, the MSC export plug-in presented in Section 4 was used for that purpose. Although primarily implemented as a way to visualize scenario execution using a widespread notation, the exported files provide a manual mechanism to double-check scenario traversals.

4 New Scenario Export Mechanism

4.1 Scenario Metamodel and Existing Tool Support

The generation of MSCs from jUCMNav execution traces has been planned since its inception. In parallel to jUCMNav's creation, initiated two years ago, a team of undergraduate students created an Eclipse-based MSC Viewer that reads the scenario files generated by the old UCMNav and converted by a filter called UCMExporter [3]. This is no longer necessary with jUCMNav, which uses the MSC Viewer metamodel to describe the result of traversals.

jUCMNav exports the XML serialization of the metamodel depicted in Figures 6 and 7 using the Eclipse Modeling Framework (EMF) [6]. The metamodel is heavily inspired by the XML DTD used by UCMExporter [3]. It contains concepts for groups of scenarios, component definitions and instances, and events, conditions, and messages ordered in sequences or in parallel (no choice as alternatives are resolved by the traversal semantics). The different types of events correspond to UCM path nodes traversed during execution.

jUCMNav creates a model instance using the information collected during a scenario traversal and serializes the result to a file. Working on this instance, another tool could generate an MSC (for use in Telelogic Tau, for example), UML sequence diagrams or TTCN-3 test skeletons.

4.2 Architecture and Algorithm Highlights

The transformation of UCM scenario execution traces to MSCs is implemented as a jUCMNav export plug-in (available via Eclipse's standard Export menu). The export mechanism creates a `.jucmscenarios` file, which can be loaded by the MSC viewer. The MSC generator itself is implemented as a simple listener to the scenario traversal algorithm. This approach decouples the traversal from the export mechanism which in turn increases the maintainability of the application as a whole; a vast improvement over the older tools.

Simply put, the export plug-in executes selected UCM scenario definitions with the MSC generator listening to the various notifications and iteratively

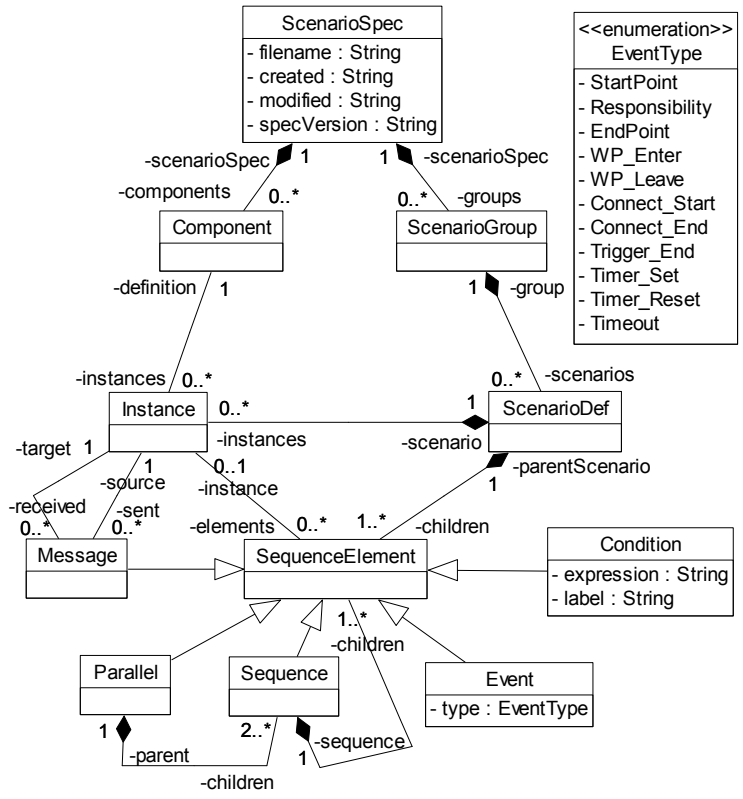

Fig. 6. Exported UCM scenario metamodel (1/2)

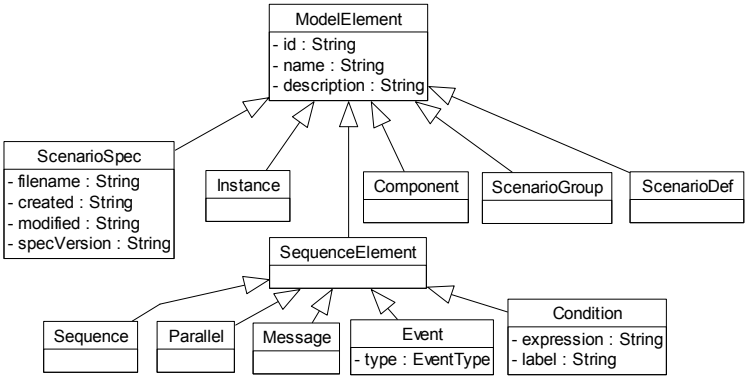

Fig. 7. Exported UCM scenario metamodel (2/2) - Model elements

building the scenarios. Generating an instance of the ScenarioSpec metamodel from the scenario traversal is actually a three-step process, which cleverly re-uses jUCMNav's internal structure and scenario traversal mechanism: 1) generate a flat UCM while traversing; 2) make it well-formed; 3) export XML scenario.

First, the traversal listener generates a new URN model that represents a *flat* view of the scenario execution (a partial order without any Or-forks, Or-joins, or stubs). jUCMNav's auto-layout feature can be used to render the UCM diagram. This greatly improves debugging capabilities and this transformation has become a feature in its own right.

Algorithmically, the flatting process maps incoming notifications to the creation of a particular element in the target map. By re-using the same internal commands used by the UCM editor when a user builds a diagram, the new target diagram is thus syntactically valid. Each executed scenario is represented in its own map and the original scenario definitions are cloned and can be re-executed verbatim on the generated URN model. An example of such a flat scenario, generated from our example model, can be seen in Figure 8 (left).

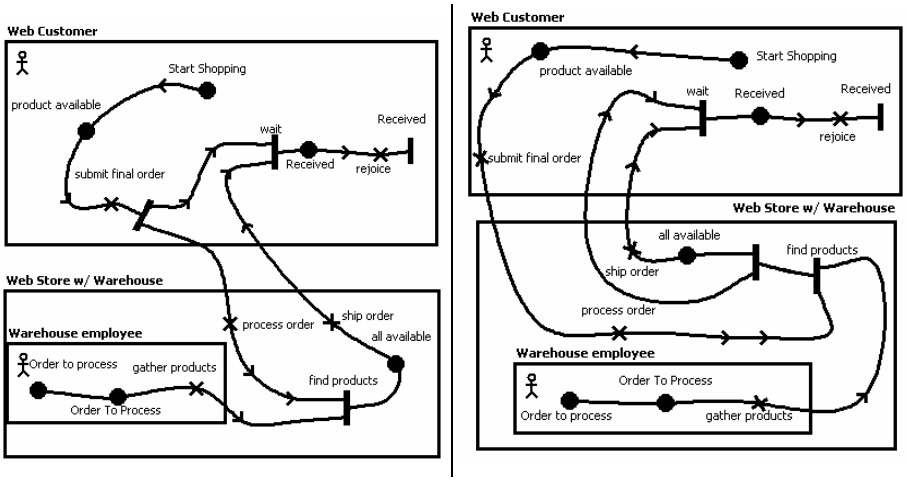

Fig. 8. Flattened scenario and its well-nested version

Second, the generated URN model is checked for well-nestedness according to the definition in [2]. If it is not well-nested (e.g., left part of Figure 8), the model is transformed and additional concurrency constraints are imposed to ensure that it can be expressed in a linear form (such as an MSC or UML sequence diagram). One well-nested version of this scenario can be seen in Figure 8 (right). Here, the direction arrows carrying plug-in traversal events (as metadata) are constrained to be executed after *process order* to make the result well-nested.

Third, the ScenarioSpec instance is built by traversing the well-nested URN model. The main complexity here is the synthesis of synchronization messages necessary to ensure causality across multiple instances. The inferred messages are

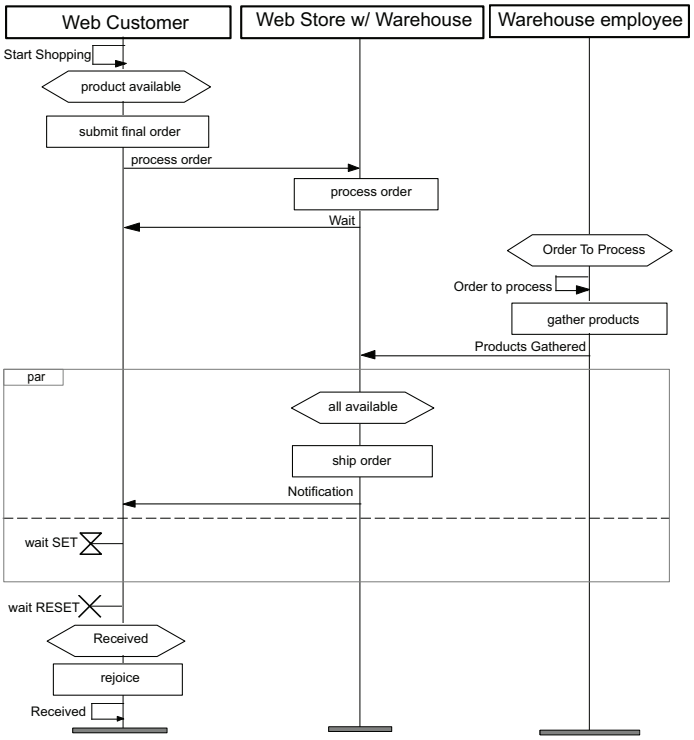

Fig. 9. Message Sequence Chart exported from Figure 8

based on the techniques presented in [3]. Additional simplifications were possible in our implementation due to a less restrictive architecture (UCMExporter was implemented as XML transformations) and because jUCMNav's query framework was re-used. This output can then be visualized in the MSC Viewer that is packaged with jUCMNav, as seen in Figure 9.

In this transformation, UCM components are converted to MSC instances, UCM start/end points to self messages, UCM responsibilities to MSC actions, UCM timers to MSC timer set/reset/timeout events, and UCM conditions to MSC conditions. Explicit UCM concurrency (And-forks) is shown using MSC *par* inline statements. The inter-instance messages are synthesized to preserve the UCM causality, and names are created automatically according to the context.

5 Discussion

5.1 Return on Contributions

Most of UCMNav's limitations presented in Section 2 and related to the previous work in [2, 3, 13] have been addressed in our extension to jUCMNav. More complex and usable data types are available, together with an action language whose

concrete syntax is compatible with SDL. Scenario definitions now support post-conditions, expected end points, and start points that can be triggered multiple times. Scenarios can be included in other scenarios, hence improving management and scalability. Visual scenario highlight is supported; traversed paths and elements are shown in a different colour and offer a hit count indicating the number of times they were traversed. Multiple scenarios can be executed, enabling coverage analysis of a set of test scenarios. The traversal, linearization, and MSC generation are entirely decoupled, and intermediate UCM representations (with scenarios) can be exported, enabling other types of analysis and transformations. The traversal can be guided by user preferences (e.g., for the required degree of determinism at choice points), and the algorithms can be overridden by external plug-ins. Various errors and warnings are reported in the standard Eclipse way, and double-clicking them brings the focus on the problematic model element. jUCMNav also includes its own MSC viewer, which supports common features such as zooming, scrolling, outlines, and diagram export.

5.2 Related Work

He *et al.* used MSCs generated from a UCM model to synthesize a SDL model for a simple telephony application [8]. They recommended improvements to the UCM traversal semantics and MSC generation that have been addressed here. In particular, the synthetic message names generated for MSCs are used in a consistent way across scenarios, the MSCs now include conditions expressing the selection of plug-in maps in dynamic stubs, and the UCM notation now distinguishes between actors (environment) and system components.

UML 2.x activity diagrams share commonalities with UCMs and the type of transformation discussed here could be applicable to the generation of sequence diagrams from activity diagrams. Störrle surveyed several transformations from activity diagrams to different semantic domains [17]. Some are done formally using denotational semantics, some are informal by examples, and others (similar to our approach) are done by algorithm/interpreter. Liang *et al.* surveyed other synthesis approaches for different notations [12]. In contrast with many of these approaches, ours handles path selection based on control variables of different types, scenario models that are hierarchical and/or not well-formed, submodels with multiple input/output segments, and complex component structures.

Bisgaard Lassen *et al.* proposed an approach to generate process descriptions (at the level of UML activity diagrams, Petri Nets, YAWL, or BPEL) from MSCs [4]. In essence, their transformation is the reverse of ours and could easily be adapted to cover UCMs as a target notation. Combining both approaches could enable a round-trip transformation process.

5.3 Analysis of Semantic Variation Points

During the traversal, many Use Case Map concepts could be interpreted in different ways. The semantic variations listed in Table 1 are related to the typical interpretation of the notation in its current form. Conceptually, the traversal

pushes tokens along UCM paths. Most of these semantic variation points have a natural solution, often related to the initial implementation in UCMNav. Because no variation points were documented, the implementation was not questioned or re-evaluated. However, given jUCMNav's extensibility and the availability of the Eclipse environment (including the standard *Problems* view), more power can be given to the modeller in terms of precisely defining the expected behaviour and resulting warnings and errors.

Table 1. Identified semantic variations

#	Element	Semantic Variation	Main Alternatives
1	Start Point	Precondition is false	Abort, Pause, Continue
2	Or-Fork Dynamic Stub	Multiple branches evaluate to true	Abort, Pick one (deterministic or not), Follow all
3	Or-Fork Dynamic Stub	No branches evaluate to true	Abort, Pause, follow random
4	And-Join	Not all in-branches arrived	Abort, Continue
5	And-Join	Do they have memory?	Yes, No, Hybrid
6	Multiple	Can block simultaneous instances of a node?	Yes, No
7	Timer	Both continuation and timeout path are enabled	Pick continuation, Pick timeout, Or-Fork Behaviour
8	Timer / Wait	Do they have memory?	Yes, No
9	Timer / Wait	Connected path arrives at unblocked node	Error, Pause, Continue
10	Stub	No plug-in exists	Error, Continue
11	End Point	Multiple different out bindings to fire	All in parallel, All in sequence, Pick one
12	End Point	Same out binding should be fired multiple times	All in parallel, All in sequence, Once
13	End Point	Post-condition is false	Abort, Pause, Continue
14	Scenario	How are start points launched?	In parallel, In sequence, Mixed

The choices made by the default scenario traversal (Table 2) were mainly motivated by intuitiveness and simplicity of implementation; the hardest decisions are related to leaving a plug-in map while dealing with concurrency. jUCMNav now supports user-defined preferences for several traversal semantic variations, and the door is opened to additional options in the formalization of the UCM notation. Interestingly enough, a recent paper [14] evaluates the UCM notation in terms of its expressiveness compared to other workflow notations. The paper identifies a set of workflow patterns [18] that are not currently well supported by plain UCM constructs. Although many of these patterns can still be modeled using a combination of constructs, they introduce contrived solutions such as in Figure 2. The relationships between these workflow patterns and the traversal semantic variation points we identified are presented in Table 3.

In summary, there are three main modifications to the UCM notation and its tooling to inherently support a wider breadth of workflow patterns, such as those above. Because these changes are tightly coupled to the traversal algorithm, they provide good insight on the semantic clarifications that are required in UCMs.

– First, two slight notational changes should be made to both Or-forks and And-forks. Conditions should be added on the And-fork branches thus combining the concepts of alternatives and concurrency; this would greatly simplify Figure 2. Such a feature would also cover non-exclusive Or-forks with multiple `true` branches (semantic variation 2).
– Second, stubs and plug-in bindings should be enhanced to support the execution of multiple concurrent plug-ins and synchronization. This would clear up the main issues brought up by semantic variations 11 and 12 while at the same time greatly increase the expressiveness of Use Case Maps.
– Finally, to properly support these workflow patterns and to clear up issues concerning blocked elements (semantic variations 5, 6 and 8), modifications should be made to the UCM traversal engine in order to support plug-in and component instances. Currently, each stub shares the same global plug-in instance, which does have its benefits in terms of simplicity, but greatly limits expressiveness. These changes may require variable instances local to particular components and plug-ins, as well.

Table 2. Default scenario traversal choices

#	Semantic Variation	Default Implementation
1	Precondition is false	Abort or pause, user-defined preference
2	Multiple branches evaluate to true	Pick one, user-defined preference
3	No branches evaluate to true	Abort or pause, user-defined preference
4	Not all in-branches arrive	Abort
5	Do and-joins have memory?	Hybrid (will soon change to yes)
6	Can block simultaneous instances of a node?	No
7	Both continuation and timeout path are true	Pick continuation
8	Do timers and waiting places have memory?	No
9	Connected path arrives at unblocked node	Error (race condition)
10	No plug-in exists	Continue only if stub has one *in* and one *out*
11	Multiple different out bindings to fire	All in parallel
12	Same out binding should be fired multiple times	Only fire exit path once
13	Post-condition is false	Abort
14	How are start points launched?	In sequence

6 Conclusions and Future Work

Providing an operational traversal semantics for the UCM notation and enabling transformations to MSCs present interesting challenges. In this paper, we have re-engineered and greatly enhanced the pre-existing scenario traversal mechanism, created a scenario traversal listener infrastructure, and identified relevant

Table 3. Relationships between workflow patterns and semantic variations

Workflow Pattern	Solution	Related to
Multiple choice	Or-Fork with concurrent branches, And-Fork with conditions on branches Dynamic Stub with concurrent plug-ins	SV: 2
Synchronizing merge	Synchronizing Dynamic Stub with concurrent plug-ins	SV: 6, 12
N-out-of-M join	Synchronizing (n/m) Dynamic Stub with concurrent plug-ins	SV: 4, 6
Discriminator	Special case (1/m) of above	SV: 4, 6
Multiple Instances without synchronization	Use of replication factor on component or plug-in binding	SV: 2, 6, 11, 12
Multiple Instances with a priori design time knowledge	Synchronizing static stub with replication factor as fixed number	SV: 2, 6, 11, 12
Multiple Instances with a priori runtime knowledge	Synchronizing static stub with replication factor as variable expression	SV: 2, 6, 11, 12
Interleaved parallel routing	Component bindings and specialized traversal	SV: 14
Deferred choice	Specialized traversal	SV: 2, 3
Milestone	Specialized traversal	SV: 2, 3

semantic variation points that will have an impact on the future of the notation and in particular on the traversal semantics. Our enhancements to the jUCMNav tool represent an important step towards a feature-rich, usable, powerful, and maintainable framework to support research and applications based on URN.

The first challenge we foresee is the implementation of the enhancements to the UCM notation described in Section 5.3. By directly supporting a wider variety of workflow patterns, the notation's expressiveness will be greatly enhanced. Once UCM's core is strengthened, the second challenge will be to reinforce URN's characteristic advantage in the niche of early requirement engineering notations: integrated support of goals (via GRL) and scenarios (via UCM) in one model [16]. Having the GRL goal model impact the scenario traversal mechanism (and vice versa) are forthcoming enhancements. Finally, aspect-oriented extensions to URN have recently been proposed [15] and will likely benefit from a good integration with the traversal semantics and MSC generation.

Acknowledgments. This research was supported by the Natural Sciences and Engineering Research Council of Canada, through its programs of Discovery Grants, Strategic Grants, and Postgraduate Scholarships. We are grateful to J.-F. Roy, E. Tremblay, J.-P. Daigle, J. McManus, and G. Mussbacher for various contributions to the jUCMNav tool, and to A. Boyko, T. Boyko, M. Kovalenkov, and T. Abumohammad for their MSC Viewer plug-in.

References

1. Amyot, D. and Mussbacher, G: URN: Towards a New Standard for the Visual Description of Requirements. In E. Sherratt (Ed.): Telecommunications and be-

yond: The Broader Applicability of SDL and MSC (SAM 2002). Lecture Notes in Computer Science 2599, Springer 2003, 21–37.
2. Amyot, D., Cho, D.Y., He X., and He, Y.: Generating Scenarios from Use Case Map Specifications. Third International Conference on Quality Software (QSIC'03), Dallas, USA, November 2003, 108-115.
3. Amyot, D., Echihabi, A., and He, Y.: UCMExporter: Supporting Scenario Transformations from Use Case Maps. In: NOuvelles TEchnologies de la RÉpartition (NOTERE'04), Sadia, Morocco, June 2004, 390–405.
4. Bisgaard Lassen, K., van Dongen, B.F., and van der Aalst, W.M.P.: Translating Message Sequence Charts to other Process Languages using Process Mining. BETA Working Paper Series, WP 207, Dept. Technology Management, Technische Universiteit Eindhoven, The Nederlands, March 2007.
`http://ga1717.tm.tue.nl/wiki/publications/beta_207`
5. Buhr, R.J.A.: Use Case Maps as Architectural Entities for Complex Systems. IEEE Transactions on Software Engineering, 24(12), August 1998, 1131–1155.
6. Eclipse: Eclipse Modeling Framework (EMF), `http://www.eclipse.org/emf/`
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, USA, 1995.
8. He, Y., Amyot, D., and Williams, A.: Synthesizing SDL from Use Case Maps: An Experiment. In R. Reed and J. Reed (Eds), System Design, 11th International SDL Forum (SDL'03), Stuttgart, Germany, July 2003. Lecture Notes in Computer Science 2708, Springer 2003, 117–136.
9. ITU-T: Recommendation Z.120 (04/04) Message Sequence Chart (MSC). Geneva, Switzerland, 2004.
10. ITU-T: Recommendation Z.150, User Requirements Notation (URN) – Language Requirements and Framework. Geneva, Switzerland, 2003.
11. java.net: JavaCC^{TM}: JJTree Reference Documentation. 2007.
`https://javacc.dev.java.net/doc/JJTree.html`
12. Liang, H., Dingel, J., and Diskin, Z.: A Comparative Survey of Scenario-based to State-based Model Synthesis. 5th Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), May 2006, ACM Press, 5–12.
13. Miga, A., Amyot, D., Bordeleau, F., Cameron, D., and Woodside M.: Deriving Message Sequence Charts from Use Case Maps Scenario Specifications. In R. Reed and J. Reed (Eds), Meeting UML - Tenth SDL Forum (SDL'01), Copenhagen, June 2001. Lecture Notes in Computer Science 2078, Springer 2001, 268–287.
14. Mussbacher, G.: Evolving Use Case Maps as a Scenario and Workflow Description Language. 10th Workshop on Requirements Engineering (WER'07), Toronto, Canada, May 2007.
15. Mussbacher, G., Amyot, D., and Weiss, M.: Visualizing Early Aspects with Use Case Maps. To appear in: LNCS Journal on Transactions on Aspect-Oriented Software Development, Springer, July 2007.
16. Roy, J.-F., Kealey, J. and Amyot, D.: Towards Integrated Tool Support for the User Requirements Notation. In R. Gotzhein, R. Reed (Eds.) SAM 2006: Language Profiles - Fifth Workshop on System Analysis and Modelling, Kaiserslautern, Germany, May 2006. Lecture Notes in Computer Science 4320, 198–215.
17. Störrle, H.: Semantics of Control-Flow in UML 2.0 Activities. In: Bottoni, P., Hundhausen, C., Levialdi, S., und Tortora, G. (Eds.), Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Rome, Italy, 2004. IEEE Computer Society, 235–242.
18. Workflow Patterns website, 2007. `http://www.workflowpatterns.com`