LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios[i]

Sebastian Uchitel, Robert Chatley, Jeff Kramer and Jeff Magee

Department of Computing, Imperial College
[su2 | rbc | jk | jnm]@doc.ic.ac.uk

Abstract. We present a tool that supports the elaboration of behaviour models and scenario-based specification by providing scenario editing, behaviour model synthesis, and model checking for implied scenarios.

Introduction

The design of concurrent systems is a complex task prone to subtle errors. Behaviour modelling has proved to be successful in helping uncover flaws at design time; however, it has not had a widespread impact on practitioners because model construction remains a difficult task and because the benefits of behaviour analysis appear at the end of the model construction effort. In contrast, scenario-based specifications have a wide acceptance in industry and are well suited for developing first approximations of intended behaviour; however, they are still maturing with respect to rigorous semantics and analysis tools. This paper presents a tool for supporting a process for elaborating system models and specifications that exploits the potential benefits of behaviour modelling and scenario-based specifications yet ameliorates their shortcomings.

From a novice user's perspective, the tool allows creating message sequence chart (MSC) specifications [1], obtaining feedback on scenarios that are missing, and adding them as positive or negative scenarios. The theory behind the feedback is that of implied scenarios [5, 6]. They signal aspects of the MSC specification that need further elaboration due to the concurrent nature of the specified system and the partial nature of the specification. From an advanced user's perspective, the tool also allows accessing the behaviour models that are synthesised for implied scenario detection. These models are the basis for reasoning on system design. Behaviour models can be viewed, edited, analysed and animated using the tool.

The Elaboration Process

Scenario notations such as MSCs (see Figure 1) describe two distinct aspects of a system. Firstly, they depict a series of examples of what constitutes acceptable system behaviour. These examples consist of sequences of messages – called traces – that system components are expected to send each other. Secondly, scenario notations outline the high-level architecture of the system. Scenarios depict with vertical arrows, or instances, which system components are involved in providing the intended system behaviour. They also describe component interfaces because they illustrate which messages are being sent and received by each component. By architecture we mean the system components and their interfaces.

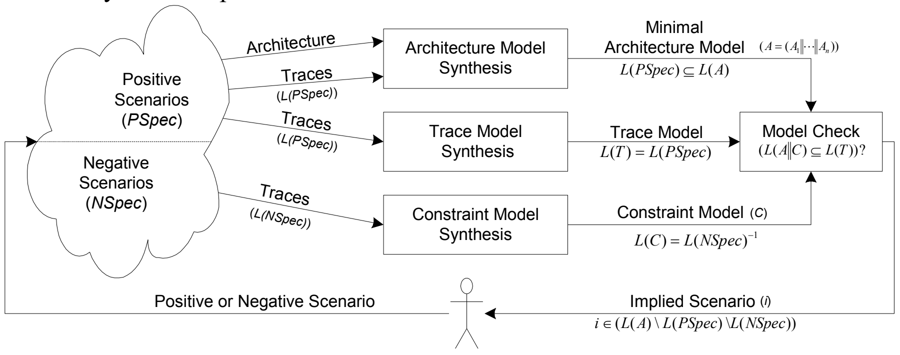
Positive Scenarios (PSpec)
Negative Scenarios (NSpec)
Architecture
Traces (L(PSpec))
Traces (L(PSpec))
Traces (L(NSpec))
Architecture Model Synthesis
Trace Model Synthesis
Constraint Model Synthesis
Minimal Architecture Model $(A = (A_1 \| \cdots \| A_n))$
$L(PSpec) \subseteq L(A)$
Trace Model
$L(T) = L(PSpec)$
Constraint Model (C)
$L(C) = L(NSpec)^{-1}$
Model Check $(L(A \| C) \subseteq L(T))?$
Implied Scenario (i)
$i \in (L(A) \setminus L(PSpec) \setminus L(NSpec))$
Positive or Negative Scenario

Fig. 1. An overview of the elaboration process

Implied scenarios indicate gaps in a scenario-based specification that are the result of specifying the behaviour of a system from a global perspective yet expecting it to be provided by independent entities with a local system view. If the architecture does not provide components with a rich enough local view of what is happening at a system level, they may not be able to enforce the intended system behaviour. Effectively, what may occur is that each component may, from its local perspective, believe that it is behaving correctly, yet from a system perspective the behaviour may not be what is intended. These additional system traces resulting from the inadequate local view of components are called implied scenarios.

The process (depicted in Figure 1) starts with a MSC specification comprised of two parts [6]: a positive part (*PSpec*) that describes the intended system behaviour and a negative part (*NSpec*) that describes undesired behaviour. The positive part of the MSC specification is defined in terms of basic and high-level MSCs [1]. The negative part is using an adapted version of basic MSCs [6]. The semantics of the MSC language determines a set of traces for each part: *L*(*PSpec*) and *L*(*NSpec*). Note that we interpret messages as synchronous hand-shaking communication, and also adopt weak sequential composition for the semantics of high-level MSCs.

From the MSC specification we synthesise three different behaviour models in the form of labelled transition systems (LTS) where transitions are labelled with the messages that components send each other. Using the positive system traces, *L*(*PSpec*), and architectural information, the architecture model synthesis builds the architecture model (*A*) as the parallel composition of components models $A_1,\ldots,A_n$ where the alphabet of A_i coincides with the interface of component *i*. Thus, we have $A=(A_1\|\ldots\|A_n)$ where $\|$ is the LTS parallel composition operator, and *L*(*PSpec*) is a subset of *tr*(*A*) where *L(A)* is the set of maximal traces exhibited by LTS A. In addition *A* can be proven to be the minimal model (with respect to trace inclusion) that complies with the MSC architecture and that includes all the specified traces

(L(*PSpec*)). Thus, maximal traces in $L(A)\backslash L(PSpec)$ are implied scenarios. The second behaviour model we build is the trace model (T). This model is built from the set of positive system traces (L(*PSpec*)), ignoring the specified architecture, such that $L(T)=L(PSpec)$ (assuming L(*PSpec*) is a regular language). The third behaviour model we build is the constraint model (C). This is built from the set of negative system traces (L(*NSpec*)) so that it captures the complement of the traces the system should not exhibit: $L(C)=L(NSpec)^{-1}$.

We are interested in maximal traces that are exhibited by the architecture model (A), have not been specified in the positive part of the MSC specification (L(*PSpec*)) and have not already been explicitly prohibited in the negative part of the MSC specification (L(*NSpec*)). Thus, we are interested in the following set of traces: $(L(A)\backslash L(NSpec))\backslash L(PSpec)$. These traces can be detected by model-checking $L(T)\subseteq L(A\|C)$. If the inclusion does not hold, an implied scenario is generated as a counter-example. Note that for non-regular positive MSC specifications [3], $L(T) \subseteq L(PSpec)$ holds rather than $L(T)=L(PSpec)$. However, implied scenarios can still be detected with the same inclusion test.

According to whether users consider the implied scenario intended or undesired system behaviour, the positive or negative part of the scenario specification is updated. By repeating the process it is possible to iteratively elaborate scenario-based specifications and behaviour models

In principle, the elaboration process based cannot be guaranteed to converge to a state where there are no more implied scenarios to be validated. This is reasonable as, on each acceptance of an implied scenario, stakeholders could keep on introducing new functionality in the form of traces that the underlying architecture model could not perform, including the addition of new message labels or even components. However, supposing that at some point all the positive behaviour of the system has been captured, it may be possible to converge to a stable specification by rejecting the rest of the implied scenarios. To support convergence we have introduced an expressive negative scenario notation [6], which in our experience, assuming L(*PSpec*) is regular, suffices to make the elaboration process converge. However, this remains an open question that we intend to investigate in future work.

As mentioned earlier, four artefacts are produced as a result of the incremental elaboration process. The *first* is a MSC specification that has been evolved from its original form to cover further aspects of the concurrent nature of the system being described, including possible functional aspects. The *second* is the architecture model, which preserves the system architecture and provides the specified positive behaviour; however, it may exhibit additional unspecified behaviour. The *third* is the trace model, which captures precisely the traces specified as positive behaviour. The *fourth* is the constraint model, which captures the properties that the architecture model should comply with if it is to avoid the negative scenarios and provide only the specified system behaviour.

The architecture model provides the basis for modelling and reasoning about system design. In fact it is the model that needs to be further developed through architectural and design decisions as designers move towards implementation. The constraint model aids the design process as it models the properties that the architecture model must satisfy.

An important observation regarding the convergence of the elaboration process is that even if the iterative process is cut off before converging, it still allows the elaboration of the initial MSC specification into a more complete system description, and produces three behaviour models that help developers reason about the design of the system.

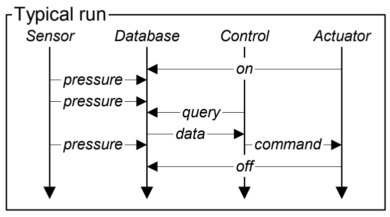

Fig. 2. A (basic) message sequence chart.

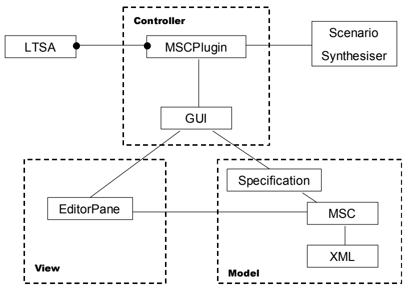

Fig. 3. Architecture of the MSC editor plug-in.

The Tool Implementation

The tool was implemented in Java as an extension of the Labelled Transition System Analyser (LTSA) [2]. LTSA uses the FSP process calculus [2] to specify behaviour models. From the FSP description, the tool generates a LTS model. The user can animate the LTS by stepping through the sequences of actions it models, and model-check the LTS for various properties, including deadlock freedom, safety and progress properties. The MSC extension builds on this introducing a graphical editor for MSCs and by generating an FSP specification from a scenario description. FSP code is generated for the architecture, trace and constraint models described previously. LTSA model checking capabilities (safety and deadlock freedom checks) are then used to detect implied scenarios.

The architecture of LTSA was adapted so that extra modules can easily be plugged in and out without having to recompile the code for the core system. All of the code associated with a plug-in module is archived into a jar file and is dynamically loaded by the core LTSA using techniques based on reflection. The core code then interrogates the plug-ins to find out whether they require extra graphical components (for instance menus, toolbars, display panels etc) to be added to user interface.

The basic architecture of the MSC editor plug-in is shown in Figure 3. Circles on the line connecting LTSA and the MSC plug-in denote the loose coupling that exists between them. The architecture is based on the familiar Model-View-Controller pattern. The underlying representation of the model is an XML tree. This allows us to easily load and save the data to a file. Other parts of the tool access and manipulate the model through objects representing the Specification, BMSCs, HMSCs, Instances etc, rather than exposing the underlying XML.

The view component generates and displays message sequence charts that can be edited. The view requests data from the model and rebuilds its display every time that it is notified that the model has changed. The GUI controller listens for events caused by user interaction and sends updates to the model when the user edits one of the diagrams. Then the view is informed and it rebuilds the display based on the updated model. The controller will pass the model (Specification) to the ScenarioSynthesiser that generates the textual FSP description of the scenario model. The core of LTSA is then called to check the model. From the information that LTSA returns, the plug-in can build a message sequence chart describing any implied scenario that may be present in the model, and pass it to the GUI to display to the user. LTSA also generates a state model that can be displayed as a set of state transition diagrams.

Conclusions and Future Work

The LTSA-MSC tool [4] has been used successfully on a number of medium-sized case studies. It has proven to support the elaboration of scenario-based specifications and behaviour models in a practical and cost-effective manner. As part of the STATUS project we are taking a scenario-based approach to modelling and simulating software systems with a view to assessing them for usability early in the design process. We hope in the future to extend the tool with a graphical front end that can mock up the proposed user interface for the system.

References

[1] ITU, *Message Sequence Charts*, International Telecommunications Union. Telecommunication Standardisation Sector, Recommendation Z.120, 1996.
[2] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*. New York: John Wiley & Sons Ltd., 1999.
[3] R. Morin, *On Regular Message Sequence Chart Languages and Relationships to Mazurkiewicz Trace Theory* in International Conference on the Foundations of Software Science and Computation Structure (FOSSACS'01), Genova, 2001.
[4] S. Uchitel, *LTSA-MSC Tool*. Department of Computing, Imperial College, 2001.
[5] S. Uchitel, J. Kramer, and J. Magee, *Detecting Implied Scenarios in Message Sequence Chart Specifications* in Joint 8th European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'01), Vienna, 2001.
[6] S. Uchitel, J. Kramer, and J. Magee, *Negative Scenarios for Implied Scenario Elicitation* in 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'02), Charleston, 2002.

[i] Partially supported by STATUS ESPIRIT project (IST-2001-32298).