MTSA: Eclipse support for Modal Transition Systems construction, analysis and elaboration. *

Nicolás D'Ippolito
Departamento de Computación, FCEyN
Universidad of Buenos Aires
ndippolito@dc.uba.ar

Dario Fishbein, Howard Foster
Computing
Imperial College London
{fdario,hf1}@doc.ic.ac.uk

Sebastian Uchitel
Departamento de Computación, FCEyN
Universidad of Buenos Aires
suchitel@dc.uba.ar

ABSTRACT

In this paper we detail the design and implementation of an Eclipse plug-in that supports construction, analysis and elaboration of Modal Transition Systems. The plug-in supports construction of MTS using FSP process algebra and synthetising from scenarios and FLTL safety properties. These models could be animated and model checked. The MTSA-Eclipse plug-in is publicly available at: http://lafhis.dc.uba.ar/ suchitel/MTSA.html.

Categories and Subject Descriptors

2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

DESIGN

Keywords

Eclipse, MTS, Synthesis, FLTL, Scenarios

1. INTRODUCTION

The requirements and design of software systems are amenable to analysis through the construction of behaviour models, that is, formal operational descriptions of the intended system behaviour. This corresponds to the traditional engineering approach to construction of complex systems. The major advantage of using models is that they can be studied to increase confidence in the adequacy of the product to be built. In particular, behaviour models used to describe software systems can be analysed and mechanically checked for properties in order to detect design errors early in the development process and allow cheaper fixes.

Although behaviour modelling and analysis have been shown to be successful in uncovering subtle design errors [2], the adoption of such technologies by practitioners has been slow. This is in part due to a mismatch between most widely adopted software development techniques and a fundamental characteristic of traditional behaviour models. On one hand, as part of the essence of widely used iterative and incremental software development processes, the available system descriptions tend to be of a partial nature leaving some aspects of the desired behaviour undefined until a more advanced stage of the process is reached. On the other hand, traditional behaviour models such as labelled transition systems (LTS) [8] and statecharts [10], are assumed to be complete descriptions up to a fixed level of abstraction: the existence of a transition models behaviour the system is expected to exhibit, while the absense of a transitions means that the system prohibits a particular behaviour. In summary, traditional behaviour models describe required and prohibited behaviour but they cannot support partial or undefined behaviour yet when the advantages of constructing models are more rewarding the complete system description is not available.

Our approach adopts Modal Transition Systems (MTS) [11] as the basis for describing partial system behavior. MTS are a natural extension to LTS, which have been proven to be successful for modeling and analyzing the behavior of systems. Systems are modeled as a set of components or sub-systems that communicate and synchronize to provide system level behavior. Each component is described as a transition system where labels on transitions represent an interaction of the component with the environment. In MTS, each transition can be either *required* or *maybe*. The later means that it is not yet certain if the interaction modelled by the transition is required or prohibited in the final system. An MTS with no maybe-transitions is a model that is fully defined up to its alphabet, and hence corresponds to an LTS. MTS models come equipped with a definition of refinement that captures the notion of "more defined than" or "more information than". A refinement step corresponds *intuitively* to removing a maybe transition or replacing it with a required one. Refinement can be shown to preserve temporal properties [1, 4], hence by refining an MTS we are guaranteed that all properties that where true (false) in the partial model will continue to be so in the refinement. The result of applying refinemet of a model M until no *maybe* transitions left in it, is called an *implementation* of M, in fact, refinement can be thought of as narrowing the number of acceptable implementations of a partial behaviour description provided by an MTS.

We have implemented the Modal Transition Systems Analy-

*Funded by IBM Eclipse Innovation Award 2006

ser (MTSA), a tool that supports building and model checking MTS, as a plugin to the eclipse framework. We expect the tool to be used in conjunction with other plug-ins that aid the modelling, analysis and development of software systems. In this way, we expect software developers to move towards implementations with greater ease and confidence. The tool embodies theory and algorithms for analysing, composing and modifying MTS developed by us (e.g. [1, 14]) and others (e.g. [11]). Our plugin MTSA, which builds on LTSA [12] (a tool for analysis of LTS), allows the users to input MTS in a textual language which includes a number of novel composition, modifier and synthesis operators, and then visualise a graphical representation of the resulting MTSs. In addition, the tool supports MTS validation by providing animation and Fluent Linear Temporal Logic model checking features.

In the remainder of the paper we present the main features of the tool and exemplify their use for building, elaborating and analysing partial behaviour models of software systems. We then describe the tool architecture and its integration into eclipse. We conclude with a discussion on future work.

2. USING MTSA ECLIPSE PLUG-IN

In this section we discuss the main features of the plugin. In Section 2.1 we show how models are input into MTSA and how the MTS models can be visualised. We then describe the construction of complex behaviour models explaining the various composition operations provided (2.2) and the synthesis techniques from scenarios and properties suppported by the tool (2.3). Finally we discuss support for model validation (2.4).

2.1 Writing and Viewing Simple Models

MTSA provides an *FSP Editor* in which MTS models are described using an extension of the language Finite State Processes, a textual language used originally to describe LTS [12]. Figure 1 shows *FSP Editor* with an example of FSP code that models a light switch. The code declares a `LIGHT` process using action prefix (`->`) and choice (`|`) to model its expected behaviour. Actions terminating with `?` represent actions that may be provided in the final implementation of the switch, but are not guaranteed to be provided. The model describes a switch which is guaranteeed to exhibit an alternating `on` and `off` behaviour, but for which it is not yet known if it can be switched off (resp. on), if the light is already off (resp. on).

MTSA supports two ways of compiling FSP code into MTS. The first is using the *Outline View* and the other one is using the toolbar associated with the *FSP Editor*. These views are depicted in Figure 8. If the user wants to compile from the toolbar, she must set focus of the plugin on the *FSP Editor* to enable the toolbar, on the other hand the user can compile from the outline view selecting the model to compile from the tree view in the *Compositions* node.

After the MTS compilation process ends, the user can see the result as a graphical representation of the model in the *MTS Draw* view. In this view the user is able to manipulate the model to enlarge or shrink the picture. Figure 2 shows the graphical representation obtained after compiling the *Light* model of the example shown in Figure 1.

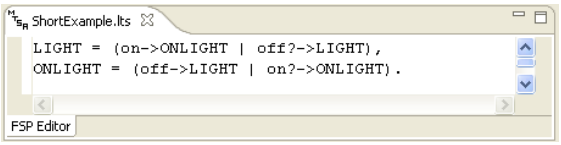

Figure 1: Light model - Editor View

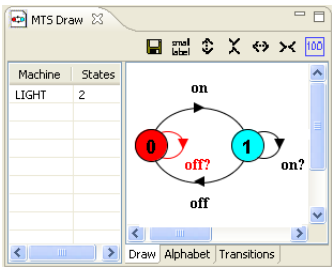

Figure 2: Light model - Draw View

2.2 Composing models

MTSA allows construction of complex behaviour models *compositionally*. The tool provides a number of different composition mechanisms. Firstly, it provides parallel composition CSP-style [5] to build models that describe the result of executing (the models of two *different*) components concurrently in an asynchronous fashion yet synchronising on shared labels. In addition to parallel composition, which has been the main focus of traditional approaches to behaviour modelling, we provide an alternative composition mechanism, namely model merging: In the context of model elaboration, we are interested in composing two partial descriptions of the *same* component to obtain a model that is more comprehensive than either of the original partial descriptions. Merging two models should return an MTS that is their least common refinement. In other words, the model that is as partial as possible while guaranteeing the required and proscribed behaviour of the models being merged. MTSA currently provides three different operatos for merging: `++` guarantees to find the least common refinement for models with the same communicating alphabet whether this exists, and provides one minimal common refinements when multiple incomparable minimal common refinementes exist.

For these cases, the operators `+cr` and `+ca` are models used currently for research purposes to approximate from above and below the set of minimal common refinements [15]. Figure 8, depicts the merge `++` of MTS depicted in Figures 4 and 7.

2.3 Model Synthesis

Although compositional operators aid the construction of complex MTS, building the models to be composed remains a difficult, labour-intensive task that requires considerable expertise. To alleviate this problem, MTSA supports automated synthesis of behavioural models from declarative requirements specifications (e.g., [13, 7]) and from synthesis from scenarios and use cases (e.g., [10, 16, 9]). The details of this work are presented in [14].

2.3.1 *Synthesis from Constraints*

Properties can be thought of as statements that prune the space of acceptable behaviours of the system to be. A behaviour model synthesized from properties should characterize all possible behaviours that do not violate the properties. Such a model provides an *upper bound* on all the behaviours that the system will actually provide, once implemented.

A webmail system is required to enforce legal and private access to the emails it stores. These requirements can be formalized in FLTL [3] as properties. Legal access requires the User be *Registered* if it is to be *LoggedIn* (i.e. □(*LoggedIn* ⇒ *Registered*)). Private access requires that the User be *LoggedIn* if she is to receive e-mail from the Server (*sendMsg*) formalised in FLTL as □(*sendMsg*⇒ *LoggedIn*). *Registered* and *LoggedIn* are fluents that change value according to the occurrence of events (see top half of *Editor View* in Figure 3). A User is *Registered* once she has been *enabled* and not yet *disabled*. A User is *LoggedIn* once she has been *authenticate*d and not yet done a *logout* nor been *disabled*. An additional requirement, *LogoutsAreAckd*, specifies that users should be sent an acknowledgment on logout, (i.e □(*logout* ⇒ **X** *logoutMsg*)).

Declaring safety properties like the above using the *constraint* keyword (see *Editor View* in Figure 3) the user can build automatically MTS that describe all the behaviour the system could provide without violating the property. The Figure 4 shows a model synthetized from *logoutsAreAckd* constraint. The MTS synthesised from a FLTL safety property is guaranteed to characterise all non-deadlocking LTS that satisfy the property [14].

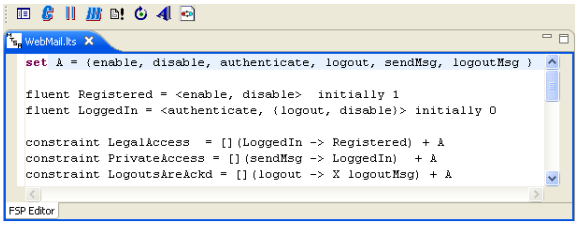

Figure 3: Web Mail Properties - MTS Editor View

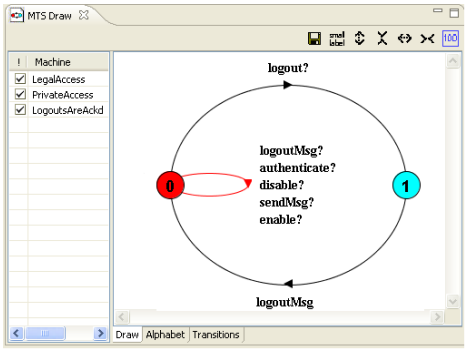

Figure 4: Synthesis from logoutsAreAckd property

2.3.2 *Synthesis from Scenarios*

In their simplest, and widely used form, scenarios are existential statements: they provide examples of the intended system behaviour; in other words, sequences of interactions that the system is expected to exhibit. By synthesizing behaviour models from scenarios, it is possible to support early analysis, validation, and incremental elaboration of behaviour models. A behaviour model synthesized from scenarios should provide a *lower bound* from which to identify the behaviours that the system will provide but that have not been explicitly captured by the scenarios.

Figure 5 provides some examples of the intended system behaviour using a standard message sequence chart notation [6]. The scenario describes a repetition (the outer rep box) of a choice (the inner alt box) between two sequences of actions: (1) a User requests authentication from the Server which then sends a number of messages; after that, the User logs out and receives a logout message. (2) an Admin disables the User during user activities, effectively expelling the latter from the system. An example of a sequence of events required by *authenticate, sendMsg, disable, logoutMsg, . . .*

Figure 6 shows the FSP code that describes the sequence chart behiavour. The code declares a `WebMailScenario` process and uses an `abstract` prefix. The process, without the abstract prefix, captures the required behaviour described in the scenarios. This part of the FSP declaration is generated automatically using standard scenario to LTS techniques. The `abstract` keyword modifies the process producing an MTS that characterises all LTS models that preserve the `WebMailScenario` model behaviour (i.e. including LTS that exhibit more behaviour than that described in the scenarios). The resulting MTS for the scenario in Figure 5 is depicted in Figure 7.

Note that the models synthesised from properties and scenarios can be composed using the merge operation as discussed above. The merge for the MTS of Figures 4 and 7 is depicted in Figure 8 and combines in one model the lower and upper bounds to the intended system behaviour as provided by the scenarios and properties of the webmail system.

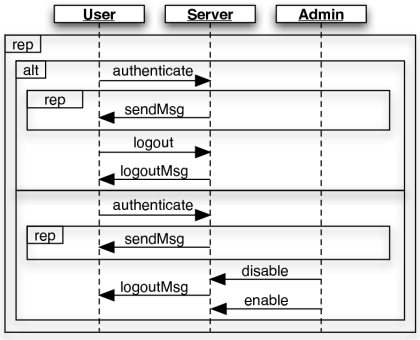

Figure 5: Web Mail Scenario Specification

2.4 Model Validation

We have discussed inputting, visualising, composing and synthesizing models in MTSA, we discuss suport for model validation. MTSA supports FLTL and deadlock-freedom model checking in addition to model animation.

```
abstract WebMailScenario = (authenticate->SEND_MESSAGE),
SEND_MESSAGE = (sendMsg->SEND_MESSAGE
       | disable->logoutMsg->enable->WebMailScenario
       | logout->logoutMsg->WebMailScenario).
```

Figure 6: Web Mail Scenario FSP

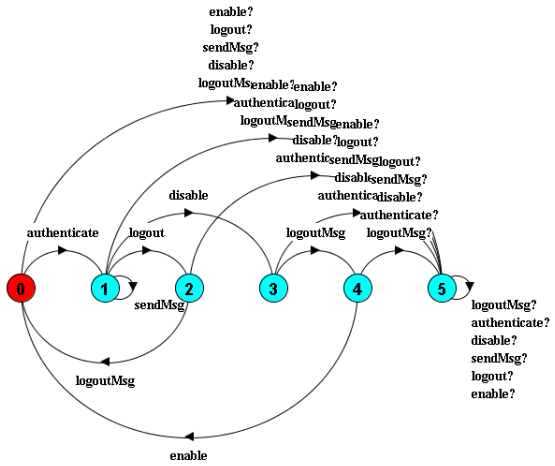

Figure 7: Web Mail Abstract Model

2.4.1 Model Checking

Model checking is an automated technique for verifying that a system satisfies a set of required properties. In the case of MTSA it supports model checking of MTS models, that is, to verify if a property is satisfied by all deadlock free implementations of a MTS model. The result of checking a formula ϕ against a model M has three possible outcomes: *true*, *false* y *maybe*. The result is *true* if and only if all implementations of M satisfy the formula ϕ, *false* if and only if all implementations of M do not satisfy the formula ϕ, and *maybe* otherwise. MTSA implements model checking as, essentially, two model checks over LTS derived from the MTS under analysis [15]. The LTS model checks are implemented using the core classes of the LTSA tool [12].

In MTSA Plugin properties to be checked are defined in FSP using the `assert` keyword as in `assert Logout = ([](X logout -> LoggedIn))`. In order to check a property, the user must right click it in the tree view of the *Outline View* (see Figure 8). The *Output View* shows the operations that are being executed during MTSA property check, as an example, the output of the *Logout* property check is shown in Figure 8.

2.4.2 Model Animation

MTSA supports model animation, this allows users to walk through the behaviour exhibited by an MTS and validate model. MTSA has an *Animator View* that it is accessible from the *animator* button in the toolbar of the plugin or simply making the *Animator View* visible. The user is displayed the MTS being animated, with its current state in red. In addition, a window with the communicating alphabet of the MTS is displayed. Enabled actions on the current state are depicted with checked boxes. The animation proceeds with the user selecting one of the enabled transitions. The MTS model is updated and the new set of enabled actions is displayed. Figure 8 depicts an ongoing animation.

3. THE MTSA ECLIPSE PLUG-IN

Using the Eclipse framework opens the potential to link the tool with a network of other Eclipse plug-in contributions and aims to simplify the number of different, bespoke tools used in software engineering as a whole. There were several reasons why we sought to leverage the Eclipse Integrated Development Environment (IDE) for our work and develop an IDE based tool rather than extending the previously standalone MTSA tool. Firstly, a growing number of editors have been released to support a number of different languages and specifications (for example, Java and C#) irrespective of actual technology deployment environment. Our approach required an IDE which was flexible to multiple editors and views working closely together. Secondly, the notion of providing extension points promotes contributing our plug-in not only to increase the number of available plug-ins, but also work closely with other contributors to enhance the overall engineering experience by plug-ins working together.

Indeed, the migration of LTSA (MTSA is based on the core of the LTSA tool [12]) to the Eclipse environment consisted in rebuilding the model, view and controller pattern using the Eclipse Plug-in development environment (PDE). There were a few issues related to moving from a standalone LTSA to an extended plug-in version of the MTSA tool. Our aim was to provide a consistent and expandable mechanism to support cross editing and view updates. As changes occur to a document, we reflect this in any associated views (i.e when a user types a FSP in the *Editor View* then the *Draw View* and *Output View* must reflect these changes). To improve the performance of MTSA we implemented threaded jobs in translation, synthesis, compilation and process analysis. Long running jobs should not restrict other work being undertaken, and should provide continuous feedback to the user.

The FSP Editor extends the MultiPageEditorPart class of the PDE. Extensions to support calling the appropriate editor are easily configurable in the plugin.xml deployment config file. The editor content is scanned on an 'input rest' (i.e. after there is a delay in user editing interactions) and upon document restore or save actions to provide useful editor functions, such as syntax highlighting. A full parsing of source is however, performed on compilation of the FSP source, whereby an outline view content is updated with a breakdown of an FSP document. This includes a list of compositions (such as specified composite processes) and a list of basic processes (possibly synthesised from scenarios or properties). Another useful PDE feature we built was the core Page sub-part of a MultiPageEditor to support views with multiple tabs (such as in the MTS Draw View to show graphical MTS models, their alphabets and FSP representation - see Figure 8).

4. CONCLUSIONS AND FUTURE WORK

We have described the MTSA plug in for Eclipse, which supports a variety of modelling and analysis features aimed at

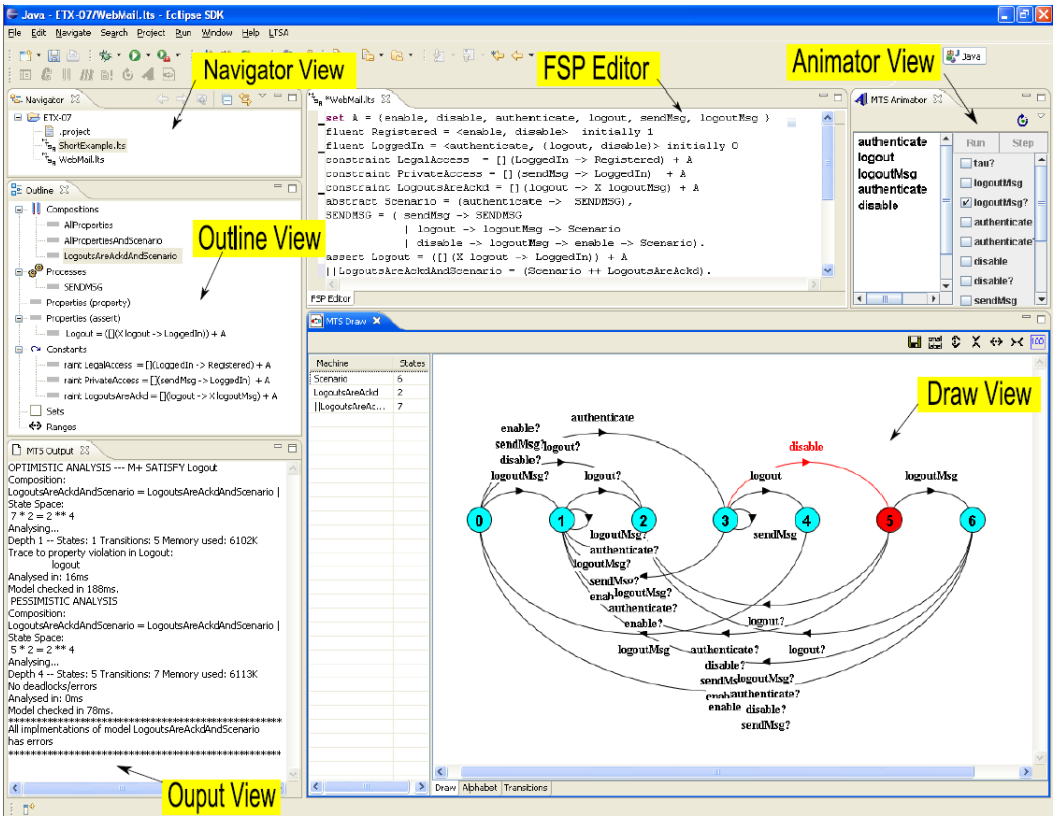

Figure 8: MTSA Plugin Full View

assisting developers in producing better systems. We aim to integrate our work with existing plugins in order to allow for a more cohesive environment for iterative software development practices. In particular, we aim to integrate with plugins that support development of graphical models such as hierarchical statecharts and scenario descriptions now that we have synthesis procedures from these languages.

5. REFERENCES

[1] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. "Multi-Valued Symbolic Model-Checking". *ACM TOSEM*, 12(4), October 2003.

[2] E. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643, December 1996.

[3] D. Giannakopoulou and J. Magee. "Fluent Model Checking for Event-Based Systems". In *ESEC/FSE'03*, 2003.

[4] D. Harel, H. Kugler, and A. Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements". In *FMs in Soft. and Systems Modeling*, pages 309–324, 2005.

[5] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall, New York, 1985.

[6] ITU. Recommendation z.120: Message sequence charts. *ITU*, 2000.

[7] R. Kazhamiakin, M. Pistore, and M. Roveri. "Formal Verification of Requirements using SPIN: A Case Study on Web Services". In *SEFM'04*, 2004.

[8] R. Keller. "Formal Verification of Parallel Programs". *CACM*, 19(7):371–384, 1976.

[9] K. Koskimies and E. MŁkinen. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7):643–658, 1994.

[10] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *Distributed and Parallel Embedded Systems*, 1999.

[11] K. Larsen and B. Thomsen. "A Modal Process Logic". In *LICS'88*, pages 203–210, 1988.

[12] J. Magee and J. Kramer. *"Concurrency - State Models and Java Programs"*. John Wiley, 1999.

[13] C. Ponsard, P. Massonet, A. Rifaut, J. Molderez, A. van Lamsweerde, and H. T. Van. "Early Verification and Validation of Mission-Critical Systems". In *FMICS'04*, 2004.

[14] S. Uchitel, G. Brunet, and M. Chechik. "Behaviour Model Synthesis from Properties and Scenarios". In *ICSE'07*, 2007.

[15] S. Uchitel and M. Chechik. "Merging Partial Behavioural Models". In *FSE'04*, pages 43–52, 2004.

[16] S. Uchitel, J. Kramer, and J. Magee. "Incremental Elaboration of Scenario-Based Specifications and Behaviour Models using Implied Scenarios". *ACM TOSEM*, 13(1), 2004.