A Tool Suite for Multi-Paradigm Specification[1]

Lynne Blair[†], Trevor Jones, Gordon Blair

Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.
[†]Corresponding author's email: lb@comp.lancs.ac.uk. Fax: +44 (0)1524 593608

Abstract. In this paper, we present a tool suite developed to facilitate the use of a multi-paradigm specification technique. By adopting such a technique, different aspects (or components) of a system can be specified using different formal languages. Using FC2 as our common file format, we can load in different (partial) specifications, compose them and then view the result in textual or graphical format, simulate the composed behaviour, and model check a given logic formula against our (composed) system.

1. Introduction

The tool suite that we present in this paper allows the composition of (partial-) specifications that have been written in a number of different formal paradigms such as process algebra (e.g. LOTOS and CCS), (real-time) temporal logic and (timed) automata. We anticipate that these component specifications will have come from a variety of sources, including other tools such as Autograph and FCTOOLS [4], Eucalyptus [5], Lite [6] and UPPAAL [1]. We thus provide an environment in which our tool can be integrated with others such as these. We have also developed and incorporated a tool to convert temporal logic formulae into automata, so that logic specifications can be composed with other aspects of our system. Used this way, temporal logic is a powerful specification technique and not simply a language for model checking.

2. A Multi-Paradigm Approach

2.1 Why Multi-Paradigm Specification?

Formal specification techniques provide the ability to specify the behaviour of a system in a language that has clear, concise and unambiguous semantics. However, it is well recognised within the formal methods community, and is not at all surprising, that different languages have their own individual strengths and weaknesses. As pointed out in [7], "there will never be a universal FDT (formal description technique) any more than there will never be a universal programming language".

[1] FASE'99 - Fundamental Approaches to Software Engineering, Amsterdam, 22-26 March 1999

J.-P. Finance (Ed.): FASE'99, LNCS 1577, pp. 234-238, 1999.
© Springer-Verlag Berlin Heidelberg 1999

As systems, and their corresponding specifications, get larger and more complex, it becomes more difficult to find a single formal language that is suitable for specifying the entire system. Instead, our approach allows the best features of several different specification techniques to be exploited. The different specifications can then be brought together using composition rules built on a common semantics.

2.2 Underlying Theory

We use timed labelled transition systems as our common semantics, from which we define timed automata as a higher level semantic model. Then, from each of the different languages mentioned above, we define a mapping onto timed automata. Since we do not have space to present the theory that lies behind our semantics and mappings, we point the interested reader to [3]. Our rules for the composition of different specifications are based on the usual interpretation of (selective) parallel composition; any action in one component specification that occurs in the selected set of synchronising actions must wait to synchronise with a matching action from the other component specification. In contrast, any actions not in the set of synchronising actions may proceed independently. These rules are presented formally in [3].

3. The Composer Tool

3.1 Composition

The user interface to our tool is displayed in figure B.1. As can be seen, input to the tool takes the form of “fc2” files or “aut” files. The FC2 format (see [4]) is used as our common file format. Although we have identified a few problems with this format (see [2]), its use is fairly widespread. Consequently, we can integrate our tool with other tools as mentioned above. For convenience, we also allow the input of “aut” files from Eucalyptus, although we then provide a conversion to FC2.

Composition can either be achieved automatically, through the recognition and matching of syntactically equivalent actions and variables, or it can be done *explicitly*. The latter provides a safer and more flexible composition in which any action (resp. variable) in one automaton can be composed with any action (resp. variable) from another. For example, “send” and “put” may be used in different component specifications for the same action. The inverse may also be true, i.e. different actions (resp. variables) may have (perhaps unintentionally) been given the same name. Explicit composition allows you to distinguish between these.

Note that to carry out further compositions, or perform model checking, a “recomposition” flag should be set at this point (set as default). However, to explore the new automaton in Autograph, the flag must be deselected (see [2] for justification). Clicking “Next” now completes the composition and saves the result to an “fc2” file; a window also appears giving the number of states and transitions in the new automaton. Note that for convenience we also automatically produce another file (“temp.aut”) which can easily be loaded into Eucalyptus for quick viewing.

3.2 Simulation

The simulator allows us to step through the system one action at a time by selecting available actions from the user interface (see figure B.2). Once an action has been selected, the simulator progresses to the next state and a new set of possible actions appear. We can also choose to idle in a particular state by incrementing the appropriate variable on the left side of the interface. Note that this may have the effect of disabling certain actions, e.g. when a variable exceeds the bound of a guarded action.

Steps that have been taken through a simulation are recorded in two formats. Firstly, a textual log is provided in the centre of the user interface; this can also be saved to file. Secondly, a graphical trace of (abbreviated) events is displayed beneath the simulator window. Labelled nodes represent the states visited and selected actions are represented by transitions between nodes. The intensity of colour increases with repeated visits to a particular state. During a simulation, we may also end up in one of the special states, labelled TRUE or FALSE. These occur if we have composed an automaton derived from logic formulae, or if we are simulating the results of model checking (see below). Intuitively, TRUE represents a valid path whilst FALSE represents an invalid path. Both of these states are effectively terminal states: once reached, you always stay in this state. However, note that in these states any action is always possible; hence they do not correspond to deadlock. The use of temporal logic as one of our specification techniques, as well as the model checking process, make it useful to visualise such states in our simulator.

3.3 Generating Automata from Temporal Logic

By selecting the “Logic Tool” from the main user interface, the user is able to enter a logic formula (representing part of the system specification) and convert it into an automaton. The user interface to this tool is shown in figure B.3. In this way, the translated formula can be composed with other component specifications. The logic we use is a linear time temporal logic, details of which can be found in [3]. Details of the translation into automata can also be found in this reference. Note however that the current version of the tool only supports an untimed logic, although derivation rules to convert real-time logic into timed automata have been defined. Ongoing work is addressing the issue of urgency/ maximal progress in automata and also exploring the use of a branching time temporal logic framework.

3.4 Model Checking

With model checking, we want to prove that a logic formula is valid with respect to a given system. The process we use to achieve this is to initially construct an automaton for the *negated* formula. This is straightforward through the use of our logic tool (also incorporated into the model checker). We then need to form the *cross product* of the overall system with the negated formula. The rules for this are very similar to those of composition; the difference is that we do not permit the independent progression of

automata – if one does an action, they must both do that action. This produces a (possibly empty) automaton. Since we have negated the formula, any path occurring in the resulting automaton shows that the formula is invalid over the given system. These paths can be explored using our simulator. Conversely, if the formula is valid over the system, the resulting automaton will be empty. As with the logic tool described above, we currently only support untimed model checking.

4. Summary

In this paper, we have described a tool suite to provide support for a multi-paradigm specification environment. We achieve this by using timed labelled transition systems as our underlying common semantic model and use timed automata as a realisation of this model. The composition of automata lies at the heart of our tool, which also includes a simulator, a logic tool for the generation of automata from temporal logic and a model checker. The tool has been designed to allow the integration with a number of other tools. Further details on our work can be obtained via the web (see http://www.comp.lancs.ac.uk/computing/users/lb/v-qos.html) or by email.

Acknowledgements

This work was carried out under the V-QoS project with the financial support of EPSRC (GR/L28890), and in collaboration with the University of Kent at Canterbury.

References

1. J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, C. Weise: "New Generation of UPPAAL", In Proceedings of the International Workshop on Software Tools for Technology Transfer, Aalborg, Denmark, 12-13 July, 1998.
2. L. Blair, T. Jones, "A note on some problems with the FC2 format", web document: http://www.comp.lancs.ac.uk/computing/users/lb/Composer/fc2notes.html, 1998.
3. L. Blair, G.S. Blair, "Composition in Multi-paradigm Specification Techniques", To appear at the 3rd International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'99), 15-18 February, Florence, Italy, 1999.
4. A. Bouali, A. Ressouche, V. Roy, R. de Simone, "The FCTOOLS User Manual (Version 1.0)", April 1996, see http://www.inria.fr/meije/verification/doc.html.
5. H. Garavel, "An Overview of the Eucalyptus Toolbox", Proceedings of the International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia), pp 76-88, June 1996, see http://www.inrialpes.fr/vasy/Publications/Garavel-96.html.
6. Lite: LOTOS Integrated Tool Environment, Tele-Informatics and Open Systems Group, University of Twente, The Netherlands, see http://wwwtios.cs.utwente.nl/lotos/lite/.
7. K.J. Turner (editor), "Formal Description Techniques", Proceedings of the 1st International Conference on Formal Description Techniques (FORTE'88), Elsevier Science, North-Holland, Amsterdam, 1990.

Appendix A: Platform Requirements

The Composer tool has been written in Java™, and can thus be used with all major platforms. In order to integrate our tool with Autograph, Eucalyptus and UPPAAL, we have used a LINUX platform for PCs.

Appendix B: Graphical User Interfaces from the Composer Tool

The following three figures show the graphical user interfaces for the main Composer tool (figure B.1), the simulator (figure B.2) and the logic tool (figure B.3).

Figure B.1

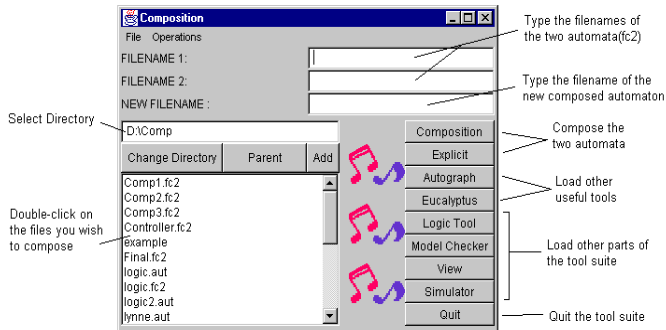

Figure B.2

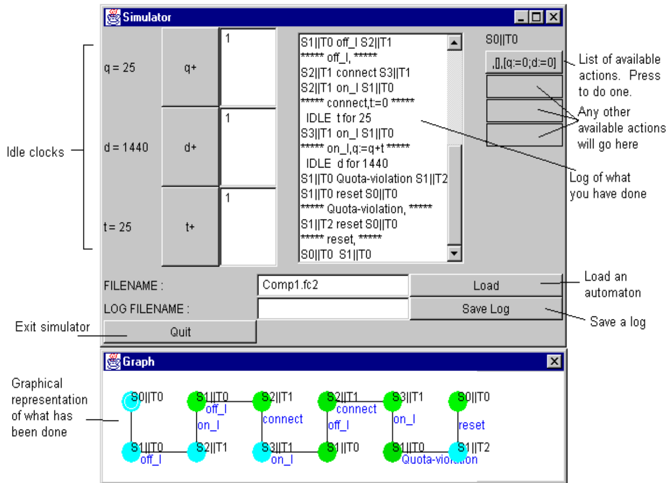

Figure B.3