ECLIPSE Feature Logic Analysis

Gregory W. Bond, Franjo Ivančić, Nils Klarlund, Richard Trefler
{bond, trefler, klarlund}@research.att.com
ivancic@gradient.cis.upenn.edu

Abstract—**ECLIPSE is a virtual telecommunications network based on IP. It is the result of an ongoing research project at AT&T Labs – Research that is investigating next-generation telecom service architectures. The ECLIPSE Statecharts language was developed to simplify feature (service) development, for example call waiting, by supporting a smooth transition from design to implementation and by supporting automated semantic analysis. The modular nature of ECLIPSE features necessitates that they utilize well-defined protocols for communicating with one another. If an individual feature fails to obey the protocol then it is likely that subscribers to the feature will be unable to complete calls. This paper describes a tool that uses the Mocha model checking tool to analyze ECLIPSE feature modules to ensure that they satisfy the specified protocols.**

Keywords— **DFC, Distributed Feature Composition, telecom services, voice over IP, VoIP, UML Statecharts, Mocha, model checking, Java**

I. Introduction

ECLIPSE is a virtual telecommunications network based on IP. It is the result of an ongoing research project at AT&T Labs – Research that is investigating next-generation telecom service architectures. The ECLIPSE network is intended to support multimedia telecommunication services involving voice, video, and text in seamless composition. The ECLIPSE network is designed to be device-independent to accommodate today's range of soft and hard devices such as cable phones, Microsoft Netmeeting^{TM}, AOL Instant Messenger^{TM}, as well as multiple external networks such as the public switched telephone network (PSTN). ECLIPSE provides a framework for rapid development and deployment of telecom services. It also provides a framework for managing "feature interaction," a problem that has hampered customization and rapid innovation of services in traditional telephony. ECLIPSE is an instance of Jackson and Zave's Distributed Feature Composition (DFC) virtual architecture [1]. DFC provides a framework for exposing and managing feature (service) interactions in multi-party, multi-feature (service) and multi-media "calls" in telecom networks. ECLIPSE implements DFC in an IP setting.

The ECLIPSE Statecharts language, hereafter referred to as "ECLIPSE Statecharts", was developed to meet the needs of ECLIPSE feature developers. Before ECLIPSE Statecharts was developed, ECLIPSE features were implemented using a general programming language (Java). It became clear early on that using a general programming language was inadequate for this purpose since it was easy to introduce faults into the feature logic, even for the simplest of features. For example, developers would forget to account for possible feature states, they would neglect to account for messages that might be received from the feature's environment, and they would respond incorrectly to messages received from the environment. For a more complicated feature like call waiting, which involves multiple parties, the problems were worse since the number of states and possible interleavings of messages exchanged with the environment were much greater. ECLIPSE Statecharts was designed to address these problems.

G.W. Bond, N. Klarlund and R. Trefler are located at AT&T Labs – Research, Florham Park, NJ, USA; F. Ivančić is located at University of Pennsylvania, Philadelphia, PA, USA

ECLIPSE Statecharts is a customized version of the Unified Modeling Language (UML) Statecharts behavioral description language [2], [3]. The UML Statecharts language, hereafter referred to as "UML Statecharts", is a graphical language for describing hierarchically structured state machines. Since it is a graphical language based on state machines it is well suited for describing the high level behavior of system structures. The language supports hierarchically structured state machines so it is possible to describe complex behavior with simple diagrams. The language also supports a number of concepts that are useful for describing timed, reactive systems, for example concurrent state machines, timed transitions, and a number of inter-object and inter-state machine communication mechanisms. In addition to being a powerful behavioral description language, UML Statecharts is part of the Object Modeling Group's UML standard for object-oriented system modeling. For this reason, a growing number of tools are available or in development to support the language.

As a design language UML Statecharts might have sufficed for describing the high level behavior of ECLIPSE features. However, by incorporating a number of ECLIPSE concepts into the language, it is possible to formally translate an ECLIPSE feature design to an implementation. Indeed, experience has shown that a feature described with ECLIPSE Statecharts needs very few additional implementation details.

Since ECLIPSE Statecharts are based on finite state machines they are suitable for automated analysis. The ability to analyze ECLIPSE feature logic is desirable for a number of reasons. A consequence of the underlying architecture of ECLIPSE is that the failure of any feature module involved in a call ("feature box" in DFC) can cause the entire call to fail. Moreover, since the ECLIPSE architecture is open we expect third-parties to develop features for use in ECLIPSE networks. For these reasons it is important to ensure that each feature module deployed in an ECLIPSE network satisfy certain minimal integrity constraints.

One way to ensure that these constraints are met is to use run-time monitoring of individual features. This approach, which is still used in parts of the current ECLIPSE network, imposes run-time overhead which we would prefer to avoid. A complementary approach is to analyze feature logic prior to its deployment in the ECLIPSE network. Using this approach, features that satisfy the constraints no longer require run-time monitoring.

In the current ECLIPSE system, ECLIPSE Statecharts exist as a set of Java classes (i.e. state classes, transition classes and an interpreter class). The Java compiler is used to perform syntax checking and type checking. However, the compiler cannot

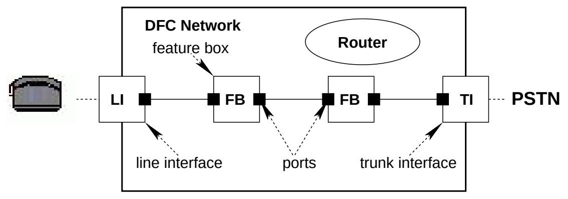

Fig. 1
A DFC NETWORK.

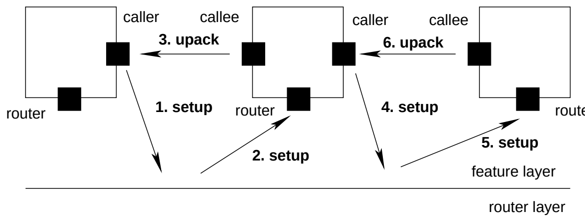

Fig. 2
CONSTRUCTING A USAGE IN DFC.

detect domain-specific semantic errors. To do this we have developed a tool to perform simple static analysis functions similar to the C language's lint tool, as well as to perform more complex model checking tasks on the code to ensure that a feature interacts correctly with its environment. The actual model checking task is performed using the Mocha model checking tool [4], [5].

II. DFC

The DFC architecture is an instance of the "pipes and filters" architectural design pattern. As shown in Figure 1, a DFC network consists of instances of a small number of component classes: Line Interface (LI) boxes which connect a single device to a DFC network, for example, a black phone; Trunk Interface (TI) boxes which connect another network to a DFC network e.g. the PSTN; Feature boxes which implement feature logic e.g. call waiting.

When a call is initiated from an LI or TI box, the router finds the destination LI or TI and then finds the feature boxes that are to be inserted based on user subscription data and precedence rules. Figure 2 shows how boxes establish connections between each other by exchanging DFC messages between peer ports according to a protocol defined as part of DFC. The overall graph of boxes that is constructed over the course of a call is called a usage.

III. INTER-PORT MESSAGING

In ECLIPSE, the behavior of an individual feature box is defined using an ECLIPSE Statechart. A feature box can communicate with its environment only via its ports which are connected to ports on peer boxes. A box's Statechart defines how the box reacts to messages it receives on its ports. The actions performed by a box in response to a message may include sending messages out its ports. Message exchange between peer ports is asynchronous and each port has its own message queue for incoming messages. This form of messaging is a refinement of one form of messaging specified by UML Statecharts.

In ECLIPSE Statecharts, as in UML Statecharts, transitions have labels of the form: $event[guard]/action$, where each label component is optional. Events are message receive operations on a port, guards are arbitrary boolean expressions, and actions are arbitrary expressions, which often include send operations on ports. A transition is enabled (fireable) if there is a message available in the specified port's queue and the guard evaluates to true. In ECLIPSE Statecharts we use the following notational short forms (borrowed from the CSP language[6]): $port!message$ to send, and $port?message$ to receive.

For the reader familiar with UML Statecharts, you should note that, unlike UML Statecharts where each object possesses a single queue for incoming asynchronous messages, ECLIPSE feature boxes potentially possess multiple queues: one for each port associated with the box.

IV. PORT PROTOCOLS

In order to support feature logic modularity in the context of a pipes and filters architecture, DFC requires that box ports obey well-defined protocols. These protocols ensure that a box can insert itself into a usage as it is being constructed, and remove itself from a usage when the usage is torn down. Once a box is inserted into a usage, a box is able to effect changes on the signaling and media associated with the "call" via its port connections.

There are four classes of box ports defined by DFC: router ports, which receive messages from a router, caller ports, which are only able to initiate connections to peer boxes; callee ports, which are only able to receive connections from peer boxes; and dual ports, which can behave as either a caller or callee port for the lifetime of a connection with a peer box.

A box programmer is responsible for ensuring that these protocols are correctly implemented for each port employed by a box. Typically a box uses a number of ports. For example, the ECLIPSE Statechart defining the feature logic for the call waiting feature, shown in Figure 3, employs 4 ports: a router port (labeled 'box'), two dual ports (labeled 'dual1' and 'dual2') and a callee port (labeled 'callee'). This Statechart utilizes the UML Statecharts notions of nested state machines and history pseudostate, as well as semantic refinements to UML Statecharts involving transition priority based on message class and nesting level, and port aliases. Port aliases permit indirectly specifying the identity of a port, analagous to how a pointer indirectly specifies the identity of a variable. Three port aliases are used in the call waiting feature: 'sub', 'conn', and 'wait'. These aliases are used to refer to the roles that actual ports play at any time during the feature's execution. For example, the roles of the ports representing the connected participant ('conn') and the waiting participant ('wait') are exchanged when the subscriber ('sub') signals the feature with a flash-hook.

V. FEATURE LOGIC ANALYSIS

Feature logic analysis addresses the following two questions:

- Did the programmer of the feature box consider all possible input messages that the environment—the peers associated with the feature box—can send to the feature box?
- Does the feature box output only those messages to its environmental peers that are expected by those peers?

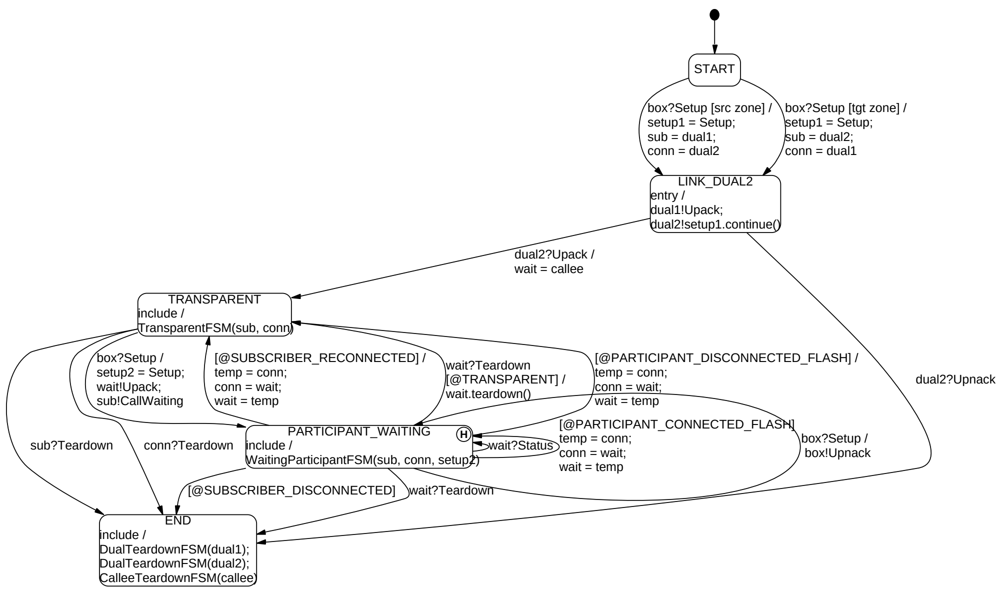

Fig. 3

THE CALL WAITING FEATURE LOGIC

Performing this analysis is a two-step process. The first step consists of translating the feature logic expressed as ECLIPSE Statecharts code into a model suitable for use by the Mocha model checking tool. The analysis of the model using Mocha is performed in the second step.

Model checking ECLIPSE feature code takes place in a test environment set up by ECLIPSE2Mocha within the modeling language framework of Mocha, called Reactive Modules (RM). That is, given a feature box B, ECLIPSE2Mocha translates B into RM and combines B with the RM versions of the standardized environmental peer entities with which B expects to communicate. Finally, ECLIPSE2Mocha adds a distinguished *bad* state to the RM model of B and its peers. The RM test environment behaves exactly like B combined with its peers except in the case when either B sends a message to a peer which the peer cannot accept or a peer sends a message to B which B cannot accept. In either case the test environment transits to the *bad* state.

Model-checking then consists of checking whether there is an execution of the test environment from the initial state of B to the *bad* state. This check can be easily embedded in the temporal logic which Mocha uses to evaluate RM models.

VI. TRANSLATION

The translation of ECLIPSE Statecharts feature logic code to a RM model consists of the series of steps shown in Figure 4.

The feature logic code – written in a subset of Java – is first parsed. The next step identifies the Eclipse Statecharts instructions, such as `addState` or `addTransition` and produces an abstract model of the code expressed as a hierarchical state machine. The hierarchical state machine is flattened, and then port aliases are instantiated. Some preliminary checks are performed on the resulting model and then a model is output in RM. These steps are explained in more detail in the following sections.

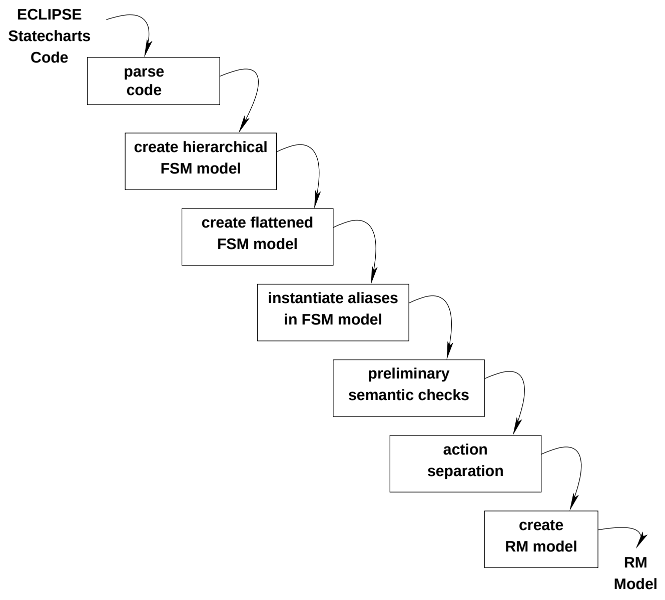

Fig. 4

TRANSLATION OF ECLIPSE FEATURE LOGIC TO A MOCHA MODEL

A. Parsing and Creating the Hierarchical FSM Model

The ECLIPSE Statecharts language is implemented in Java. To define an ECLIPSE Statechart a programmer subclasses the main FSM class and, in this class's constructor, creates instances of state and transition objects and invokes methods to add them to the FSM. The parent FSM class, the transition classes and the state classes comprise the ECLIPSE Statecharts language and its interpreter.

Programmers can define action methods for transition or state instances that will be executed by the FSM interpreter when a transition fires or when a state is entered or exited. Similarly, programmers can define guard methods for transitions that will be invoked by the FSM interpreter to determine if a transition is enabled.

The parsing step parses the Java code that defines an ECLIPSE Statechart. The parser grammar is customized to recognize the declarations of transitions, states, and their associated action or guard methods, as well as declarations of ports associated with the box. The resulting parse tree contains the elements of the Java code that are necessary for constructing the hierarchical FSM model.

B. Flattening

Since Mocha's RM language does not support the notion of hierarchical state machines, these are flattened to simplify translation to the RM model. Also, the semantics of Eclipse Statecharts are easily expressed in the flattened state machine. This is particularly true for the transition priority rules used in Eclipse Statecharts. Another reason for favoring a flattened state machine is that checking other properties not checked by Mocha, like "Are all possible status messages in this state covered?", is much easier in a flattened state machine.

It should be noted that the flattening phase does not increase the number of states in the state machine. On the contrary, it may actually reduce the number of states in the state machine. However, the flattening phase normally increases the number of transitions. Furthermore, the number of states will increase exponentially relative the to original program size when state machines are hierarchically nested.

C. Instantiation

Eclipse Statecharts permit the use of port aliases—that is, variables that range over the ports of the feature box. Mocha's RM does not directly support variable aliasing so the instantiation step explicitly instantiates occurrences of port aliases with their possible values. Instantiation results in adding conditions to transition guards. In general, instantiation will also increase the number of transitions.

D. Preliminary Semantic Checks

In this step we check certain semantic properties that are easily checked in the flattened state machine. Currently, we are checking whether a state that accepts a specific status message on a given port also takes care of all other possible status messages on that port. This is a nice by-product of flattening the state machine, because certain properties can be checked easily in a state-by-state fashion.

E. Action Separation

The RM model is not able to directly express the case of a feature box sending more than one message to the same peer during one transition. If such behavior is detected, the sending actions in the same transition are separated from each other by introducing a so-called micro-state μ. By introducing a sequence of micro-states, each with exactly one incoming transition and one outgoing transition, we handle the fact of sending a sequence of messages to the same peer. So, for example, if an action sends n messages to the same peer, we introduce $n-1$ new micro-states $\mu_{i,i=1,\ldots,n-1}$.

VII. Creating the Reactive Modules Model

The final translation step shown in Figure 4 creates the RM model for the Mocha model checker. In order to understand the mapping from the flattened FSM model to the RM model, it is necessary to provide some background information on Mocha and the RM language itself.

A. The Model Checker Mocha

Model checking is emerging as a practical tool for automated debugging of embedded software. In model checking, a high-level description of a system is compared against a logical correctness requirement to discover inconsistencies. Since model checking is based on exhaustive state-space exploration, and the size of the state space of a design grows exponentially with the size of the description, scalability remains a challenge. The model checker Mocha is based on the idea of exploiting modular design structure during model checking. Instead of manipulating unstructured state-transition graphs, it supports the hierarchical modeling framework of Reactive Modules.

The language Reactive Modules is a modeling and analysis language for heterogeneous concurrent systems with synchronous and asynchronous components. This is accomplished by the notion of time rounds. As a modeling language it supports high-level, partial system descriptions, rapid prototyping, and simulation. As an analysis language it allows the specification of requirements either in temporal logic or as abstract modules. Finally, as a language for concurrent systems, it allows a modular description of the interactions among the components of a system.

The behavior (executions) of a reactive system can be visualized in a message sequence charts (MSC) like fashion by using the simulator. To run the simulator, the user selects a module and the submodules/variables to be traced. For each selected variable, a vertical line shows its evolution in time. The value of a variable is displayed only when it changes. The same format is used to display the counter-examples generated by the model checkers during failed verification attempts. The simulator can be used either in automatic or in manual mode.

Mocha allows the specification of requirements in a rich temporal logic called alternating temporal logic (ATL). By far the most common requirements are invariants, and thus it is of utmost importance to implement invariant checking efficiently. With this in mind, Mocha provides both fine-tuned enumerative and symbolic state search routines for invariant checking.

B. The RM Model

The flattened state machine is translated into a single RM module. The environmental peers are instances of predefined RM modules. The combination of these RM modules constitutes the RM model that is used for model checking.

The operational semantics of the state machine are explicitly translated into the RM model. The communication between a box's Statechart and its environmental peers is accomplished using the following sub-round structure of one RM time round:

1. First a peer is chosen to send at most one message and update its internal state.

2. The feature box model will receive either no message or exactly one message from one of its peers. If it does not receive a message, it will not do anything. If it receives a message, it will update its internal state, and it might also send messages to its peers. Note that the use of micro-states constrains a feature box model to send at most one message to any peer.
3. The peers will receive the messages from the feature box. If a peer receives a message from the feature box, it updates its internal state.

Whenever a peer receives a message that it does not have an explicit transition for, the peer model fires an implicit transition into a special "bad state", that indicates the feature box is incorrect. Similarly, whenever the box model receives a message that it does not have a transition for, the box model fires an implicit transition into a bad state.

The approach to modeling we have adopted avoids explicitly modeling the queues between the feature box and its environment. Therefore, a message that is sent from a peer (and potentially changes the internal state of the peer) has to be handled immediately by the feature box model. In reality a feature box might actually enqueue a peer message for a while before it looks at it. So to avoid flagging an error in the feature box because it received a message that it was not expecting at this point (the programmer merely decided not to bother with this peer in this particular state), we have to ensure that the peer does not send this message in the first place. Therefore, we have introduced enabling flags that signal a peer whether the feature box model is ready to accept any message the peer might want to send at a given time.

We can avoid modeling the queues involved in the real system by modeling them implicitly in the environment of the feature box. The peers are allowed to skip a round where they are supposed to send a message, which basically models the fact that the previous message has not arrived at the peer yet. The possibility to skip a message can also be interpreted as a delay of the message from the peer to the feature box. After careful consideration of the environmental models it is clear that all possible message sequences that the environment in the original setting might send can be sent in the RM model.

The last step of our translation is the output of a RM model of the feature box. The peers have predefined models that are based on the state machines shown in Figures 5, 6, 7, and 8. If a feature box sends out instances of status message subclasses, rather than just instances of the parent status message class, it is necessary to add transitions into the peer models. Consider the case that a feature box sends out the status messages subclass $m_1, \ldots, m_n$ to its peers. For each transition in the peer model that is labeled with "`?status`", we will include a transition for each message $m_{i,i=1,\ldots,n}$ with the label "`?`m_i".

As mentioned before, we translate the flattened state machine as one RM module. The RM model maintains a variable called `currentState` that ranges over all states, and keeps track of the state that the flattened state machine is in. Each transition is translated to one update rule in the model. An additional rule covers the case that no message arrives from the environment in a time round. This rule ensures that the state of the state machine does not change. If a message arrives, but no update rule is applicable, then we will enter a "dead state" and flag

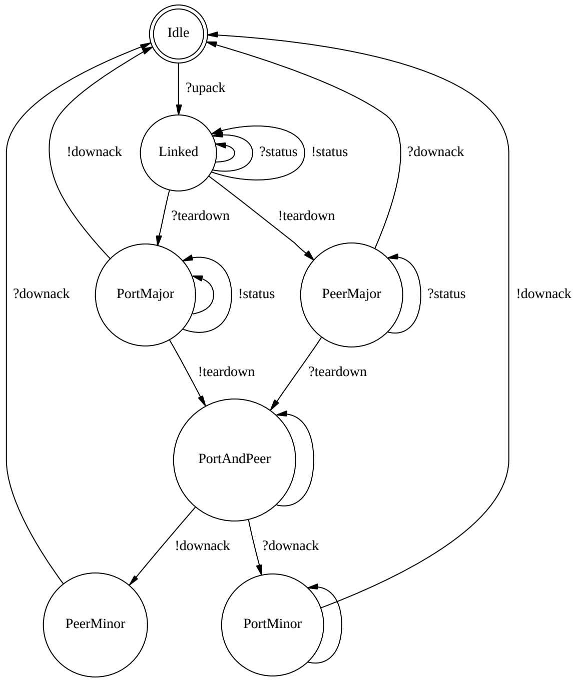

Fig. 5
DFC CALLEE PORT PEER PROTOCOL

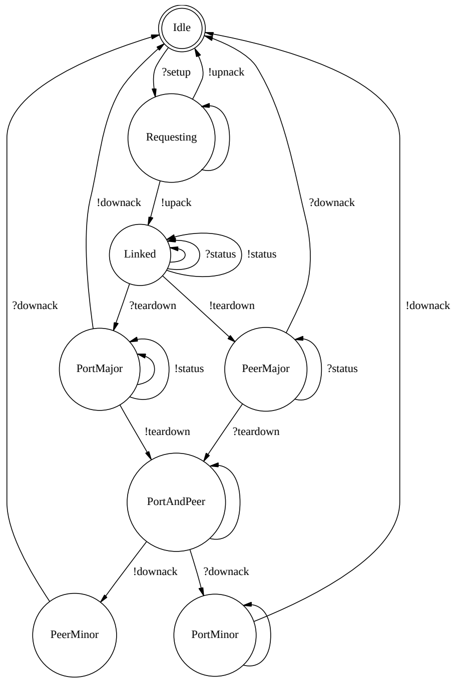

Fig. 6
DFC CALLER PORT PEER PROTOCOL

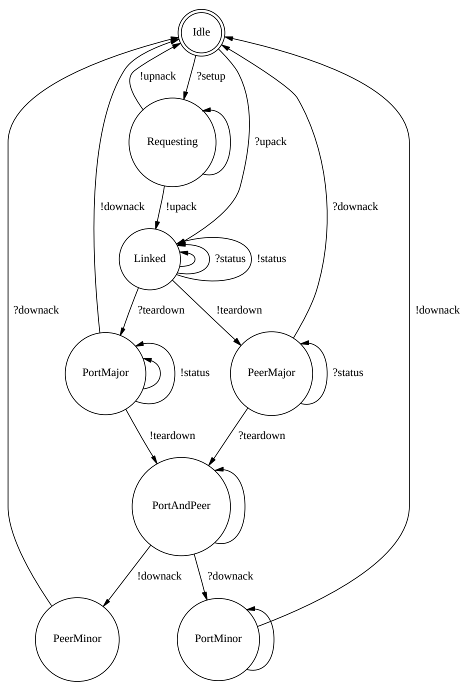

Fig. 7
DFC DUAL PORT PEER PROTOCOL

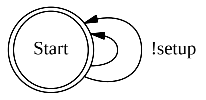

Fig. 8
DFC ROUTER PORT PEER PROTOCOL

an error. This covers one of the problems we are looking for, because it basically means that the programmer of the feature box has forgotten to take account for this particular message.

Figure 9 shows a sample transition in the flattened state machine. Its source state is A, and its destination is state B. It is enabled if the guard *alias* $= p \wedge g$ is true, and if the message m from the peer of port p arrives at the feature box. If this transition is taken, then the actions $p_1!m_1$, $p_2!m_2$, *alias* $= p_3$, are executed.

We translate this transition into RM in the following manner:

```
currentState = A & signalFromP? &
messageFromP' = m & alias = p & g ->
signalToP1!  ; messageToP1' := m1 ;
signalToP2!  ; messageToP2' := m2 ;
alias' := p3 ;
currentState' := B ;
```

When we send a message to a peer, we update the corresponding value, but we also have to make sure that it is realized that we

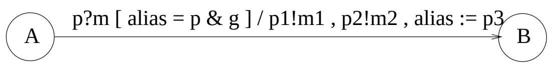

Fig. 9
TRANSITION FROM A TO B IN THE FLATTENED STATE MACHINE

updated the value. We therefore issue a Mocha event by saying `signalToP1!`. To check whether there has been a message send from P in this round, we check the corresponding event by `signalFromP?`. To receive the message that was sent in this round, we have to ask for `messageFromP'` instead of `messageFromP`, which holds the value of the previous time round.

VIII. MODEL-CHECKING ECLIPSE STATECHARTS

In Figure 10, we give an example of an ECLIPSE Statecharts feature box we have analyzed. This feature behaves like a buffer after it has been set up between a left and right neighboring box. Upon proper initialization according to the DFC protocol, the box is in state 'linked', where it reads messages from its right hand neighbor on the calleePort and sends them to its left hand neighbor on it callerPort and vice versa. The states 'linked', 'transparent1' and 'transparent2' explain this behavior. Here, we have assumed that the messages are atomic; in reality, the messages contain values that are temporarily stored in the feature box. The remaining states are necessitated by the DFC protocol.

If Mocha does determine that the state *bad* is reachable from the initial state, a debugging mechanism in Mocha is available to reconstruct the sequence of events leading to the bad state. In Figure 11 we show a screen shot of Mocha displaying such a trace. The error was generated by altering the program in Figure 10 so that the feature box sends two consecutive teardown messages: we changed the 'unlink5' to 'unlink6' transition so that a teardown message is placed on the callerPort instead of a downack. The trace, which is shown only partially, reflects that error by taking a path in the protocol that produces two teardown messages.

During the programming in Java of this trivial feature box, we introduced several little errors as typically happens, mostly due to misspellings. All but one were caught by the parser of the ECLIPSE2Mocha tool. (Some would also have been caught by the Java compiler.) The one that was not caught was discovered through model checking. The model checker approved of the Java code, but even a positive answer must be taken with a grain of salt. For example, in our setting the model checker does not check for liveness properties like "does the feature box always eventually acknowledge a teardown request?". Thus, it is a reasonable sanity check to willfully introduce errors in the feature box program that is purported to be correct. When we did this, we discovered that the program sometimes, unexpectedly, still was passed by the model checker. As a result, we discovered a misspelling of a method name that issues a message to a port. This Java error would not have been caught by a compiler since the erroneous name appeared in the initializer for an object of an anonymous, inner class.

When we originally programmed the call waiting box in Figure 3, we struggled with three insidious programming errors, all of which we later presented to our tool. They were all correctly identified through error traces.

A. Correctness of analysis

To give a complete account of what the correctness of our analysis is would be a huge task. For example, we would need a formal description of the semantics of ECLIPSE Statecharts,

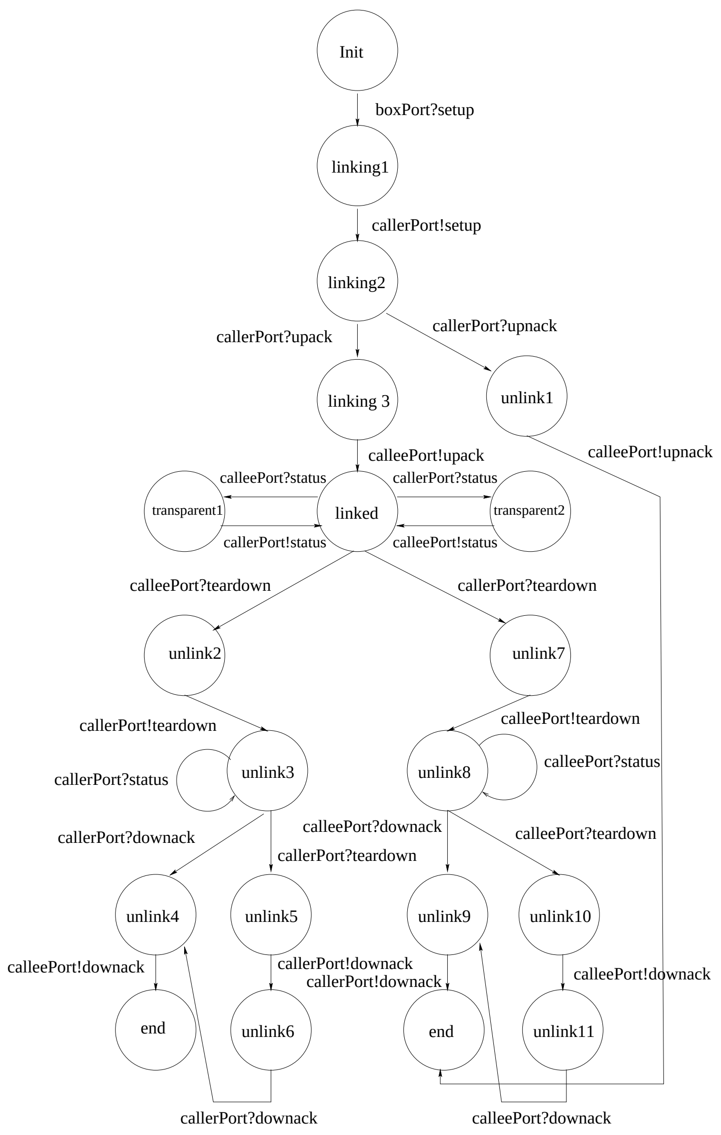

Fig. 10

TRANSPARENT FEATURE BOX

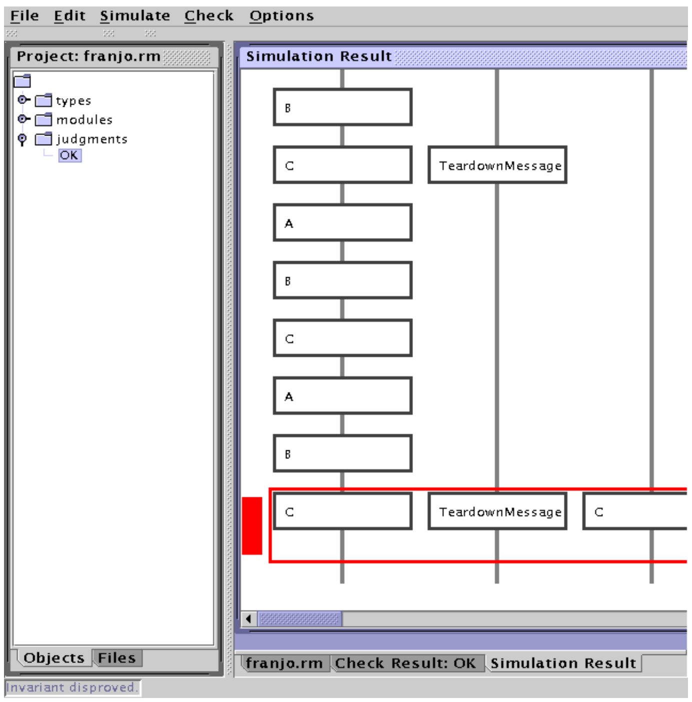

Fig. 11

MOCHA ERROR TRACE

the semantics of the translation, and the semantics of the Mocha language. Also, we would need to carefully explain the abstractions that are inherent to our analysis. Instead, we will give an informal statement that reflects our belief that we have correctly implemented the ECLIPSE Statecharts semantics through the translation to Mocha. Thus, we will have to relate errors found during runtime to errors discovered by our tool. Our concept of error is that of the Section V: an error occurs if either the environment or the feature box is unable to process a message. We say that there is an error in the ECLIPSE Statecharts feature if there exists an environment that follows the peer protocols and for which the composite system may enter a situation where a port is unable to process a message. Note that such a situation is characterized by a trace (history) of communication events. In general, traces involve buffering of messages—something that our Mocha model does not accommodate. Therefore, we say that a trace is synchronous if the event following a send message is the accept of the message. Moreover, we assume that the only communication events that involve the feature box are those that are recognized as such by the ECLIPSE Statecharts parser (thus, communication events invoked through auxiliary method definitions are not allowed). Also, we assume that all Boolean guards on message transitions are true. Then, we believe the following to be true.

- (Soundness) Any error found in the ECLIPSE test environment by the model checking procedure is an error in the ECLIPSE Statecharts feature.
- (Completeness under synchronization assumption) Assume that an ECLIPSE Statecharts feature exhibits an error in a synchronous trace. Then, the ECLIPSE Statecharts will not pass the model checking procedure.

In a more advanced tool, it would sometimes be relatively straightforward to analyze Boolean guards if they involve local variables. The issue of analyzing queued system is generally undecidable, since queues tend to resemble tapes of Turing machines.

IX. RELATED WORK

Research is currently very active and diverse in the area of model checking Statecharts. Space does not permit us to provide a comprehensive overview of this activity. Instead, we will address the current work most closely related to our own. Similar to our own work, [7], [8], [9], [10] have developed approaches to model checking properties of systems defined in various Statechart dialects. Only one of these approaches ([9]) addresses queued, asynchronous messaging between Statechart objects—similar to the way that an ECLIPSE feature box interacts with its environmental peers. However, in their approach they assume bounded queues between the environment and an object.

Instead of arbitrarily bounding queue length, our approach abstracts away the queues by exploiting properties of the environmental peer protocols and the semantics of the Mocha RM modeling language. Although the approach we use is not general enough to be applied to all possible environmental behavior, it is suitable for the environmental behavior defined by the DFC architecture.

In practice, the customized parser that we built for Java programs turned out to be very useful by itself for writing ECLIPSE Statecharts. The parsing step has revealed domain-specific semantic errors in finite state machines that the Java compiler deems to be error-free. Moreover, our experience with this tool validates the use of statically-checked constraints that formalizes software architectures. Several general tools for expressing such architectural constraints on code have been proposed; see [11] for references and the description of CoffeeStrainer, a tool for checking Java programs.

X. Conclusions and Future Work

We have built a tool, ECLIPSE2Mocha, for analyzing the communication behavior of an ECLIPSE feature and its immediate environment. ECLIPSE2Mocha is capable of detecting subtle semantic errors of ECLIPSE feature code and using the Mocha reporting features ECLIPSE2Mocha is well suited as a debugging aide for ECLIPSE features. Our translation of ECLIPSE feature code to RM code relies on a crucial abstraction – namely, the modeling of asynchronous communication via unbounded queues by synchronous communication. However, because we restrict the types of properties analyzed, errors detected by ECLIPSE2Mocha can be translated into errors of the ECLIPSE feature code.

For the future we see several interesting directions to take this work. Firstly, we see a need for an intermediate language between the Java code of ECLIPSE Statecharts and RM. Such an intermediate language would make the use of other analysis tools far easier and remove the need for a direct mapping between ECLIPSE Statecharts and RM. Secondly, we would like to explore the use of model checkers of hierarchical models [12] to avoid the flattening phase currently used by ECLIPSE2Mocha. Thirdly, we are interested in incorporating ECLIPSE2Mocha and the use of Mocha directly within a domain specific compiler for ECLIPSE Statecharts.

Finally, we would like to enhance the class of properties checked. This can be done by enlarging the type of environmental entities used in the analysis and by more faithfully modeling the unbounded queues and asynchronous communication of ECLIPSE. These enhancements would allow us to check significantly more feature interaction properties.

Acknowledgments

The authors would like to thank the other members of the ECLIPSE project at AT&T Labs – Research for their feedback during the development of ECLIPSE Statecharts and the analysis tool: Eric Cheung, Andrew Forrest, Michael Jackson, Hal Purdy, Chris Ramming, Xiaotao Wu (Columbia University) and Pamela Zave.

References

[1] Michael Jackson and Pamela Zave, "Distributed feature composition: a virtual architecture for telecommunications services," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 831–847, Oct. 1998.
[2] David Harel and Eran Gery, "Executable object modelling with Statecharts," *IEEE Computer*, July 1997.
[3] Object Management Group, *OMG Unified Modeling Language Specification, version 1.3*, Object Management Group, June 1999, Available at `ftp://ftp.omg.org/pub/docs/ad/99-06-08.pdf`.
[4] Rajeev Alur, Thomas A. Henzinger, F.Y.C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran, "Mocha: Modularity in model checking," in *Proceedings of the Tenth International Conference on Computer-aided Verification (CAV)*. 1998, number 1427 in Lecture Notes in Computer Science, pp. 521–525, Springer-Verlag.
[5] L. de Alfaro, R. Alur, R. Grosu, T. Henzinger, M. Kang, R. Majumdar, F. Mang, C. Meyer-Kirsch, and B.Y. Wang, *Mocha: Exploiting Modularity in Model Checking*, August 2000, Available at `http://www-cad.eecs.berkeley.edu/~mocha`.
[6] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.
[7] S. Gnesi, D. Latella, and M. Massink, "Model checking UML Statechart diagrams using JACK," in *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*, 1999, pp. 46–55.
[8] E. Mikk, Y. Lakhnech, M. Siegel, and G.J. Holzmann, "Implementing Statecharts in PROMELA/SPIN," in *Proceedings of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, 1998, pp. 90–101.
[9] J. Lilius and I.P. Paltor, "vUML: a tool for verifying UML models," in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, 1999, pp. 255–258.
[10] Chonlawit Banphawatthanarak and Bruce H. Krogh, "Verification of stateflow diagrams using SMV: sf2smv 2.0," Tech. Rep., Dept. of Electrical and Computer Engineering, Carnegie Mellon University, June 2000, Available at `http://www.ece.cmu.edu/~krogh`.
[11] B. Bokowski, "Statically-checked constraints on the definition and use of types in Java," in *Proceedings of ESEC/FSE'99*, 1999, vol. 1687 of *LNCS*, pp. 355–375.
[12] R. Alur and M. Yannakakis, "Model checking of hierarchical state machines," in *Proceedings of the Sixth ACM Symposium on the Foundations of Software Engineering*, 1998, pp. 175–188, Available at `http://www.cis.upenn.edu/~alur`.