IMPLEMENTING AND VERIFYING THE SAFETY OF THE TRANSACTOR MODEL

By

Brian Boodman

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____
Carlos Varela, Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

April 2008
(For Graduation May 2008)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The transactor model is an extension of the actor model designed to tolerate failures in distributed systems. Transactors can provide guarantees about consistency of a distributed system's state in the face of message loss and temporary failures of computing nodes. The model introduces dependency information and a two-phase checkpointing protocol. The added dependency information enables transactors to track the interdependencies caused by communications between actors, making it possible to ensure that the state of the distributed program as a whole remains globally consistent. This thesis discusses the use of three tools in order to test and prove the safety of the *transactor* model. We used Maude rewrite systems as a tool to test the model behavior and to discover problems with the model. During this stage, we discovered a safety bug and proposed changes to fix it. We then used the Athena proof verification system to show that the updated model is safe. Finally, we used the Salsa actor programming language as a basis for a higher-level transactor-based prototype programming language.

First, we developed a prototype implementation of the transactor model using Maude. Maude's underlying rewriting rules system is well-suited towards developing an executable operational semantics for concurrent programming models. The implementation was used to test example programs and check the transactor model's safety. This prototype was in fact used to discover a safety error.

Subsequently, we wrote a formal proof in the Athena language. As a multi-sorted first order logic system, Athena provides an effective means of representing the transactor model's correctness properties. Because Athena proofs are computer-checkable, they are more reliable than traditional proofs. Further, Athena permits the use of automated theorem proving, allowing us to skip tedious steps which would otherwise unnecessarily complicate the proof's readability.

Finally, we developed a coordination language using Salsa and Java. The language provides a practical demonstration of the use of the transactor model and shows some of the potential issues in creating an effective implementation of the model.

1. Background

1.1 Overview

Distributed computing over the internet allows processing to take place where resources, including memory, processing power, and data may not be available on a single, local machine. There are many web services and grid systems which make use of distributed computing in order to leverage the resources and information available on separate machines. It is useful to find some way of modeling such distributed systems in order to create tools to make use of distributed systems. However, modeling such distributed systems is not always sufficient as real systems may experience failures, resulting in them being unable to complete their computations. Thus, it is also necessary to provide a model which accounts for such failures. Using such a model makes it possible to formally analyze the behavior of systems built on such a model to ensure that they fulfill the important requirement of being able to somehow recover from failures. A model makes it possible to provide guarantees that such recoveries are possible within some set of constraints.

1.1.1 Types of Failures

1.1.1.1 Permanent Failures

Permanent failures are failures which result in data being lost forever. The most intuitive form of permanent failure is an irrecoverable hardware failure such as a a failed hard drive. However, distributed systems may also experience permanent failures if a machine is removed from the distributed network and never put back into the network. By definition, it is not possible to recover state information from machines which have permanently failed. However, the use of redundancy can make it possible to recover from permanent failures by loading the lost information from another machine.

1.1.1.2 Transient Failures

After a transient failure, data may be lost, but the machine which generated the data can simply run the processes again, restoring the lost data. This is not possible after a permanent failure because the machine will never run the processes again. Temporary node failures occur when a particular machine on the distributed system fails. This could be caused by a computer crash or may be triggered programmatically by the programmer. Unlike with permanent failures, in a temporary node failure the failed machine will restart, recovering the most recent backup. Machines which depended on the failed node will also need to recover from previous backups.

1.1.2 Message Loss

Message losses may be caused by network errors or may be the result of a machine being disconnected from the network. Unless the machine which sent a message experiences a transient or permanent failure, a machine can recover from message loss by resending any lost messages.

1.1.3 Globally Consistent State

Of paramount importance in a distributed system is maintaining a globally consistent state. A global state becomes inconsistent if a machine on the network experiences a transient failure while some other machine which depends on it does not fail. In this situation, the machine which did not experience a failure will contain state information which is based on the assumption that the machine which provided it with related data did not fail. For example, let us suppose some bank is using a distributed account system with a machine for accounts for two people, `Bob` and `George`. If `Bob` sends a message to `George` saying, “I have sent you 500 dollars” and then experiences a transient failure, the system will no longer be consistent as `George` will have received the 500 dollars but `Bob` will not have sent it. So, a globally consistent state is one in which none of the interdependencies between machines results in inconsistencies between the states of these machines. Note also that one can not only consider whether the entire distributed system is safe but also can consider a subset of the machines on the system as being globally consistent.

1.2 Transactor Model of Reliable Distributed Computing

1.2.1 Overview of the Transactor Model

The transactor model [5] is an extension of the Actor model [1]. The actor model is a programming model for computing with distributed state. Actor primitives encapsulate state information and run concurrently both internally and with other actors. Actors are able to communicate with other actors via asynchronous message-passing (messages may arrive in any order). Actors may spawn other actors and can migrate to different locations.

The transactor model makes specific guarantees about the global state of the system. It is intended to ensure that the distributed system maintains a globally consistent state in the face of transient failures and message losses. The model does not handle permanent failures. Message losses can never result in a globally inconsistent state, since a machine will not be dependent upon another machine until after it receives the message that machine sent. While at any given time the state of the distributed system may contain machines which are not consistent with one another, the transactor model keeps track of the interdependencies between these machines, ensuring that such machines will roll back to a previous state if necessary in order to maintain consistency. Thus, the system will move from globally consistent state to globally consistent state.

Further, the transactor model ensures that the distributed system will be *safe*. We define safety as a guarantee that if there is an execution trace going from one global system state to another, where both are globally consistent, then that the execution trace could be simulated without the node failures. Because the solution with these failures could have been reached without failures, users can treat transactors as though node failures do not happen.

1.2.2 Semantic Domains

1.2.2.1 Histories

Transactors use a two-phase commit protocol. A transactor is either *stable* or *volatile*. A transactor becomes stable via a local operation, and does not lose this status unless it fails or checkpoints, at which point it becomes volatile again, as in Figure 1.1. A stable transactor cannot change its state and thus cannot gain new dependencies. Further, it cannot fail unless a transactor it depends on is volatile. This is according to

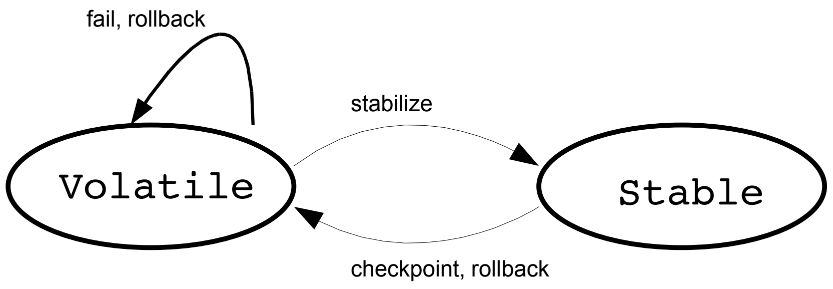

Figure 1.1: Cycle of the histories for a transactor.

the model; in an implementation of the transactor model, a failure would be hidden as the transactor would merely reload its state from a backup (which could be created when it became stable). A stable transactor can checkpoint if every transactor it depends on is also stable. A transactor history indicates if the transactor is volatile or stable and encodes the sequence of checkpoints and rollbacks the transactor has gone through. Note that a transactor can only checkpoint if the transactor and all of its direct and indirect dependencies are stable.

1.2.2.2 Worldviews

The transactor model is a specialization of actors and inherits all the constructs from the actor model, however, it explicitly represents state to enable fine-grained dependencies. In addition, transactors and messages have an additional dependency component, called a *worldview*. The worldview is a 3-tuple (γ, δ, ρ). It is composed of a *history map* γ, *dependence graph* δ, and a *root set* ρ. The dependence graph is a directed graph of dependencies and may be unconnected. The dependence graph stores only the transactor names, but these names are associated with the histories stored in the history map component. The root set is a set of transactor names which is also associated with the histories stored in the history map component. A transactor's root set can be interpreted as being potential dependencies. Transactors are added to the root set when new messages are received. If a transactor changes its volatile state in response to a received message, the transactors within the root set will become actual dependencies and will be connected to the transactor's own name within its dependency graph. In this way, a transactor gains new dependencies when it changes state in response to received messages.

An illustrative example is a stateless proxy transactor. Let us suppose some proxy transactor `TProxy` handles received messages by sending out new messages, but has no state. Because it would never set its own state, it would never gain any new dependencies. Transactors receiving messages from the stateless transactor and then setting their state would be dependent upon the proxy transactor, since a transactor always introduces itself as a dependency when sending a message. However, such dependencies would never prevent a transactor from checkpointing, since the proxy does not depend on anything and has no state. Note that this assumes that when the proxy transactor is created, it is initialized by becoming stable, which provides an explicit guarantee that it will not change its state (and that it cannot roll back).

Note that the transactor model discussed in the POPL paper [5] did not make use of worldviews as they are presented here. The previous model was updated to account for a safety error as discussed in Chapter 3.

1.2.2.3 State

The dependence graph and history map are maintained between messages and are used to track existing dependencies. As transactors retain this information between messages, a transactor may know about some dependencies of other transactors, even if it does not itself depend on them. The root set, on the other hand, is used to introduce new dependencies. Thus, a message's root set would indicate which transactors the message itself depends on. If a transactor performed a `setstate` operation after receiving a message, these dependencies would be added as new dependencies. Because dependencies are used to indicate that a transactor's state depends on other transactors' states, a transactor will not add these new dependencies if it does not set its state. Further, it will remove them upon receiving a new message.

1.2.3 Operational Semantics

1.2.3.1 Receiving Messages

Messages include the worldview of the transactor which sent the message. The recipient of a message may roll back if the message's worldview shows that something the recipient depends on has rolled back. If this does not occur, then the transactor may reject

[pure] *Evaluate pure redex.*

$$\frac{e \longrightarrow_\lambda e' \quad e \in \mathcal{E}_P^{rdx}}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, e\,], s\, ;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[pure]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, e'\,], s\, ;\, \gamma, \delta, \rho\rangle]}$$

[new] *Create new transactor.*

$$\begin{array}{ll}\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathbf{trans}\ b'\ \mathbf{init}\ s'\ \mathbf{snart}\,], s\, ;\, \gamma, \delta, \rho\rangle] & \\ \quad \xrightarrow[\text{[new]}]{t} \quad \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, t'\,], s\, ;\, \gamma', \delta \cup \delta', \rho \cup \{t'\}\rangle] & t' \text{fresh} \\ \qquad\qquad [t' \mapsto \langle b', \mathsf{nil}\, ;\, \mathbf{ready}, s'\, ;\, \gamma', \delta' \cup \delta, \emptyset\rangle] & \gamma' = \gamma[t' \mapsto \mathbf{H}_0] \\ & \delta' = t' \leftarrow (\rho \cup \{t\})\end{array}$$

[snd] *Send message. Include transactor in message dependencies.*

$$\begin{array}{l}\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathbf{send}\ v_m\ \mathbf{to}\ t'\,], s\, ;\, \gamma, \delta, \rho\rangle] \\ \quad \xrightarrow[\text{[snd]}]{t} \quad \mu \uplus \{t' \Leftarrow \langle v_m\, ;\, \gamma, \delta, \rho \cup \{t\}\rangle\} \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathsf{nil}\,], s\, ;\, \gamma, \delta, \rho\rangle]\end{array}$$

[rcv1] *Message dependences not invalidated by transactor, and viceversa: process message normally.*

$$\frac{\neg((\gamma_m \downarrow \rho_m) \rtimes \gamma') \qquad \neg(\gamma(t) \nrightarrow \gamma'(t))}{\begin{array}{l}(\mu \uplus \{t \Leftarrow \langle v_m\, ;\, \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathbf{ready}, s\, ;\, \gamma, \delta, \_\rangle] \\ \quad \xrightarrow[\text{[rcv1]}]{t} \quad \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, (b\ v_m), s\, ;\, \gamma', \delta', \rho'\rangle]\end{array}} \qquad (\gamma', \delta', \rho') = (\gamma, \delta, \_) \oplus (\gamma_m, \delta_m, \rho_m)$$

[get] *Retrieve state.*

$$\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathbf{getstate}\,], s\, ;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[get]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, s\,], s\, ;\, \gamma, \delta, \rho\rangle]$$

[set1] *Transactor is volatile: setting state succeeds.*

$$\frac{\neg\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathbf{setstate}(s)\,], \_\, ;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[set1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathsf{true}\,], s\, ;\, \gamma, \delta \cup (t \leftarrow \rho), \rho\rangle]}$$

[self] *Yields reference to own name.*

$$\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, \mathbf{self}\,], s\, ;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[self]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\, ;\, \mathcal{R}[\, t\,], s\, ;\, \gamma, \delta, \rho\rangle]$$

Figure 1.2: Transition rules encoding basic actor semantics.

the message without rolling back if the worldview within the message and the worldview of the transactor show that the message was generated by a transactor which has since rolled back. Otherwise, the message will be accepted. In that case, the transactor will update its worldview based on the message's worldview.

1.2.3.2 Transactor Configuration Transition Rules

The operational semantics of the transactor model is defined as a labeled transition sytem between transactor configurations based on the actor model. The transition rules inherited from the actor model are shown in Figure 1.2. The transitions introduced in the transactor model are shown in Figures 1.6, 1.3, and 1.9, with the last modeling spontaneous message loss and node failures.

[rol2] *Transactor is volatile and persistent: rollback reverts state to contents of persistent state saved by last checkpoint.*

$$\frac{\surd(\gamma(t)) \qquad \neg\Diamond(\gamma(t)) \qquad \gamma(t) \looparrowright h'}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{rollback}, \_\ ;\ \gamma, \_, \_\rangle] \ \xrightarrow[\text{[rol2]}]{t}\ \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s_{\surd}\ ;\ [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[rol3] *Transactor is volatile and ephemeral: rollback causes transactor to be annihilated.*

$$\frac{\neg\surd(\gamma(t)) \qquad \neg\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle \_, \_\ ;\ \mathbf{rollback}, \_\ ;\ \gamma, \_, \_\rangle] \ \xrightarrow[\text{[rol3]}]{t}\ \mu \parallel (\theta \setminus t)}$$

[rcv2] *Message dependences invalidated by those of transactor but not vice-versa: discard message.*

$$\frac{((\gamma_m \downarrow \rho_m) \rtimes \gamma') \qquad \neg(\gamma(t) \looparrowright \gamma'(t))}{\begin{array}{l}(\mu \uplus \{t \Leftarrow \langle \_\ ;\ \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s\ ;\ \gamma, \delta, \_\rangle] \\ \quad \xrightarrow[\text{[rcv2]}]{t}\ \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s\ ;\ \gamma', \delta', \emptyset\rangle]\end{array}} \qquad (\gamma', \delta', \_) = (\gamma, \delta, \_) \oplus (\gamma_m, \delta_m, \rho_m)$$

[rcv3] *Transactor dependences invalidated by message and transactor is persistent: transactor rolls back, message remains.*

$$\frac{\gamma(t) \looparrowright \gamma'(t) \qquad \surd(\gamma(t))}{\begin{array}{l}(\mu \uplus \{t \Leftarrow \langle v_m\ ;\ \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, \_\ ;\ \gamma, \delta, \_\rangle] \\ \quad \xrightarrow[\text{[rcv3]}]{t}\ (\mu \uplus \{t \Leftarrow \langle v_m\ ;\ \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s_{\surd}\ ;\ [t \mapsto \gamma'(t)], \emptyset, \emptyset\rangle]\end{array}}$$

$$(\gamma', \_, \_) = (\gamma, \delta, \_) \oplus (\gamma_m, \delta_m, \rho_m)$$

[rcv4] *Transactor dependences invalidated by message and transactor is ephemeral: transactor is annihilated.*

$$\frac{\gamma(t) \looparrowright \gamma'(t) \qquad \neg\surd(\gamma(t))}{(\mu \uplus \{t \Leftarrow \langle \_\ ;\ \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle \_, \_\ ;\ \mathbf{ready}, \_\ ;\ \gamma, \delta, \_\rangle] \ \xrightarrow[\text{[rcv4]}]{t}\ \mu \parallel (\theta \setminus t)}$$

$$(\gamma', \_, \_) = (\gamma, \delta, \_) \oplus (\gamma_m, \delta_m, \rho_m)$$

Figure 1.3: Transition rules for programmatic rollback and consistency management.

1.2.4 The τ-Calculus

The τ-calculus shown in Figure 1.7 is used in order to represent the *persistent behavior* of individual transactors. A transactor's persistent behavior is a list of message handlers used in order to respond to individual messages. Syntactic sugar is introduced in Figure 1.10.

The $\mathbf{msgcase}$ construct yields a lambda abstraction whose body processes incoming messages. Messages are assumed to take the form of a vector of parameters, the first of which is an atom that constitutes a *message name*. The $\mathbf{msgcase}$ body tests the value of the incoming message and processes the other message arguments appropriately; messages that are not understood are ignored.

The $\mathbf{declstate}$ construct declares names for a transactor's state, which is presumed to consist of a vector of elements. This construct does not "expand" into a core τ-calculus expression, instead, it simply defines a static name scope for subsequent references of the

```
Input:  $(\gamma_t, \delta_t, \_),\ (\gamma_m, \delta_m, \rho_m)$
Output: $(\gamma', \delta', \rho')$

Step 1: let $G = (V, E)$, where
            $V = \{\langle t, h\rangle \mid \gamma_t(t) = h \vee \gamma_m(t) = h\}$
            $E = \{\langle t_1, h_1\rangle \leftarrow \langle t_2, h_2\rangle \mid (t_1 \leftarrow t_2 \in \delta_t \ \wedge\ \gamma_t(t_1) = h_1 \ \wedge\ \gamma_t(t_2) = h_2)$
                        $\vee(t_1 \leftarrow t_2 \in \delta_m \wedge\ \gamma_m(t_1) = h_1 \wedge\ \gamma_m(t_2) = h_2)\}$

Step 2: while $\exists\langle t, h_1\rangle, \langle t, h_2\rangle \in V$, s.t., $h_1 < h_2$ do
            if $h_1 \rightarrow_{\Diamond} h_2$ then
              $E = E \cup \{\langle t, h_2\rangle \leftarrow \langle t', h'\rangle | \langle t, h_1\rangle \leftarrow \langle t', h'\rangle \in E\}$
                  $\cup \{\langle t', h'\rangle \leftarrow \langle t, h_2\rangle | \langle t', h'\rangle \leftarrow \langle t, h_1\rangle \in E\}$
            else if $h_1 \rtimes h_2$ then
                $\forall t' | \langle t', h'\rangle \leftarrow \langle t, h_1\rangle \in E$, do
                    $V = V \cup \{\langle t', h''\rangle\}$, where $h' \looparrowright h''$.
            else if $h_1 \dashv h_2$ then
                $\forall t' | \langle t, h_1\rangle \lhd_E \langle t', h'\rangle$, do
                    $V = V \cup \{\langle t', h''\rangle\}$, where $h' \rightarrow_{\Diamond} h''$.
            $V = V - \{\langle t, h_1\rangle\}$
            $E = E - \{\langle t, h_1\rangle \leftarrow \_\} - \{\_ \leftarrow \langle t, h_1\rangle\}$

Step 3: let $(\gamma', \delta', \rho')$ be defined as
            $\gamma'(t) = h$                iff $\exists\langle t, h\rangle \in V$.
            $t_1 \leftarrow t_2 \in \delta'$            iff $\langle t_1, \_\rangle \leftarrow \langle t_2, \_\rangle \in E$.
            $\rho' = \rho_m - \{t \mid \gamma_m(t) \dashv \gamma'(t)\}$
```

Figure 1.4: The dependence worldview union algorithm: $(\gamma', \delta', \rho') = (\gamma_t, \delta_t, \_) \oplus (\gamma_m, \delta_m, \rho_m)$

form $!u$ and $u := e$, which expand into appropriate operations on the transactor's state vector.

1.2.5 Example Reference Cell Programs in the τ-calculus

Figure 1.11, Figure 1.12, and Figure 1.13 shows three examples of a *reference cell* in the τ-calculus. A reference cell is a simple data container which holds a value. It accepts two types of messages, `set(val)` and `get(customer)`, which are used to set and get the contents of the reference cell, respectively. Figure 1.11 shows an unreliable reference cell. This equivalent to a reference cell written using actors and does not handle

$$\begin{array}{ccll} \rightsquigarrow & \triangleq & (\looparrowright \cup \rightarrow_{\Diamond} \cup \rightarrow_{\surd}) & \text{("is succeeded by")} \\ \rtimes & \triangleq & [\rightarrow_{\Diamond}]\cdot \looparrowright \cdot \rightsquigarrow_{*} & \text{("is invalidated by")} \\ \dashv & \triangleq & [\rightarrow_{\Diamond}]\cdot \rightarrow_{\surd} \cdot \rightsquigarrow_{*} & \text{("is validated by")} \end{array}$$

Figure 1.5: History transition rules

[set2] *Transactor is stable: attempt to set state fails.*

$$\frac{\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathbf{setstate}(\_)\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[set2]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathsf{false}\,], s\,;\, \gamma, \delta, \rho\rangle]}$$

[sta1] *Transactor is volatile: stabilization causes it to become stable.*

$$\frac{\gamma(t) \rightarrow_{\Diamond} h'}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathbf{stabilize}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[sta1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathsf{nil}\,], s\,;\, \gamma[t \mapsto h'], \delta, \rho\rangle]}$$

[sta2] *Transactor currently stable:* **stabilize** *is a no-op.*

$$\frac{\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathbf{stabilize}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[sta2]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathsf{nil}\,], s\,;\, \gamma, \delta, \rho\rangle]}$$

[chk1] *Transactor is independent: checkpoint succeeds.*

$$\frac{\Diamond(t, \gamma, \delta) \qquad \gamma(t) \rightarrow_{\surd} h'}{\mu \parallel \theta[t \mapsto \langle b, \_\,;\, \mathbf{checkpoint}, s\,;\, \gamma, \delta, \_\rangle] \xrightarrow[\text{[chk1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s\,;\, \mathbf{ready}, s\,;\, [t \mapsto h'], \emptyset, \emptyset\rangle]}$$

[chk2] *Transactor is dependent or volatile:* **checkpoint** *simply behaves like* **ready**.

$$\frac{\neg\Diamond(t, \gamma, \delta)}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathbf{checkpoint}, s\,;\, \gamma, \delta, \_\rangle] \xrightarrow[\text{[chk2]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathbf{ready}, s\,;\, \gamma, \delta, \emptyset\rangle]}$$

[rol1] *Transactor is stable:* **rollback** *simply behaves like* **ready**.

$$\frac{\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathbf{rollback}, s\,;\, \gamma, \delta, \_\rangle] \xrightarrow[\text{[rol1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathbf{ready}, s\,;\, \gamma, \delta, \emptyset\rangle]}$$

[dep1] *Transactor is weakly independent: yields false.*

$$\frac{\tilde{\Diamond}(t, \gamma, \delta)}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathbf{dependent?}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[dep1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathsf{false}\,], s\,;\, \gamma, \delta, \rho\rangle]}$$

[dep2] *Transactor is dependent: yields true.*

$$\frac{\neg\tilde{\Diamond}(t, \gamma, \delta)}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathbf{dependent?}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[dep1]}]{t} \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\,;\, \mathcal{R}[\,\mathsf{true}\,], s\,;\, \gamma, \delta, \rho\rangle]}$$

[lose] *Message loss.*

$$(\mu \uplus \{m\}) \parallel \theta \xrightarrow[\text{[lose]}]{m} \mu \parallel \theta$$

Figure 1.6: Transition rules encoding basic transactor semantics.

$\mathcal{A}$	$=$	$\{\mathsf{true}, \mathsf{false}, \mathsf{nil}, \ldots\}$	*Atoms*
$\mathcal{N}$	$=$	$\{0, 1, 2, \ldots\}$	*Natural numbers*
$\mathcal{T}$	$=$	$\{t_1, t_2, t_3, \ldots\}$	*Transactor names*
$\mathcal{X}$	$=$	$\{x_1, x_2, x_3, \ldots\}$	*Variable names*
$\mathcal{F}$	$=$	$\{=, +, \ldots\}$	*Primitive operators*
$\mathcal{V}$	$::=$	*Values*	
		$\mathcal{A} \mid \mathcal{N} \mid \mathcal{T} \mid \mathcal{X}$	
	$\mid$	$\lambda\mathcal{X}.\, \mathcal{E}$	*Lambda abstraction*
	$\mid$	$\langle \mathcal{V}, \mathcal{V} \rangle$	*Pair constructor*
$\mathcal{E}_P$	$::=$	*Pure expressions*	
		$\mathcal{V}$	
	$\mid$	$(\mathcal{E}\ \mathcal{E})$	*Lambda application*
	$\mid$	$\mathbf{fst}(\mathcal{E})$	*First element of pair*
	$\mid$	$\mathbf{snd}(\mathcal{E})$	*Second element of pair*
	$\mid$	$\mathbf{if}\ \mathcal{E}\ \mathbf{then}\ \mathcal{E}\ \mathbf{else}\ \mathcal{E}\ \mathbf{fi}$	*Conditional*
	$\mid$	$\mathbf{letrec}\ \mathcal{X} = \mathcal{E}\ \mathbf{in}\ \mathcal{E}\ \mathbf{ni}$	*Recursive definition*
	$\mid$	$\mathcal{F}(\mathcal{E}, \ldots, \mathcal{E})$	*Primitive operator*
$\mathcal{E}_A$	$::=$	*Traditional actor constructs*	
		$\mathcal{E}_P$	
	$\mid$	$\mathbf{trans}\ \mathcal{E}\ \mathbf{init}\ \mathcal{E}\ \mathbf{snart}$	*New transactor*
	$\mid$	$\mathbf{send}\ \mathcal{E}\ \mathbf{to}\ \mathcal{E}$	*Message send*
	$\mid$	$\mathbf{ready}$	*Ready to receive message*
	$\mid$	$\mathbf{self}$	*Reference to own name*
	$\mid$	$\mathbf{setstate}(\mathcal{E})$	*Set transactor state*
	$\mid$	$\mathbf{getstate}$	*Retrieve transactor state*
$\mathcal{E}$	$::=$	*Transactor expressions*	
		$\mathcal{E}_A$	
	$\mid$	$\mathbf{checkpoint}$	*Make failure-resilient*
	$\mid$	$\mathbf{rollback}$	*Revert to prev. checkpt.*
	$\mid$	$\mathbf{stabilize}$	*Prevent state changes*
	$\mid$	$\mathbf{dependent}?$	*Test dependence*

Figure 1.7: The τ-calculus grammar

$\mathcal{W}$	$::=$	$\mathbf{V}(\mathcal{N}) \mid \mathbf{S}(\mathcal{N})$	*Volatility value*
$\mathcal{Y}$	$::=$	$\langle \mathcal{W}, [\mathcal{N}] \rangle$	*Transactor history*
$\mathcal{H}$	$=$	$\mathcal{T} \xrightarrow{f} \mathcal{Y}$	*History map*
$\mathcal{D}$	$::=$	$\mathcal{T} \leftarrow \mathcal{T}$	*Dependency*
$\mathcal{G}$	$=$	$\{\mathcal{D}\}$	*Dependence graph*
$\mathcal{R}$	$=$	$\{\mathcal{T}\}$	*Root set*
$\mathcal{S}$	$::=$	$\langle \mathcal{V}, \mathcal{V}\ ;\ \mathcal{E}, \mathcal{V}\ ;\ \mathcal{H}, \mathcal{G}, \mathcal{R} \rangle$	*Transactor*
$\mathcal{M}$	$::=$	$\mathcal{T} \Leftarrow \langle \mathcal{V}\ ;\ \mathcal{H}, \mathcal{G}, \mathcal{R} \rangle$	*Message*
Θ	$=$	$\mathcal{T} \xrightarrow{f} \mathcal{S}$	*Name service*
Ω	$=$	$\{\!\{\mathcal{M}\}\!\}$	*Network*
$\mathcal{K}$	$::=$	$\Omega \parallel \Theta$	*Transactor configuration*

Figure 1.8: Semantic domains.

[fl1] *Spontaneous failure of volatile, persistent transactor causes rollback.*

$$\frac{\surd(h) \qquad \neg\Diamond(h) \qquad h \looparrowright h'}{\mu \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \_, \_\ ;\ \delta[t \mapsto h], \delta_c, \rho \rangle] \quad \xrightarrow[\text{[fl1]}]{t} \quad \mu \parallel \theta[t \mapsto \langle b, s_{\surd}\ ;\ \mathbf{ready}, s_{\surd}\ ;\ [t \mapsto \langle h', \rangle], \emptyset, \emptyset \rangle]}$$

[fl2] *Spontaneous failure of volatile, ephemeral transactor causes it to be annihilated.*

$$\frac{\neg\surd(h) \qquad \neg\Diamond(h)}{\mu \parallel \theta[t \mapsto \langle \_, \mathsf{nil}\ ;\ \_, \_\ ;\ \delta[t \mapsto h], \_, \_ \rangle] \quad \xrightarrow[\text{[fl2]}]{t} \quad \mu \parallel \theta}$$

Figure 1.9: Transition rules modeling spontaneous failures.

failures. The persistent reference cell shown in Figure 1.12 is assumed to be handling stable, independent clients. It will stop being able to make progress if a dependent client sets its state, until it is informed that the client has become independent or has rolled back. Figure 1.13 shows a persistent reliable reference cell. It rolls back any time it receives a message from a transactor which is not stable and independent, thus never changing state in response to such messages.

1.3 Model Checking and Rewrite Systems with Maude

Maude [8] is a language which provides support for rewriting systems. The Maude language is used to create an executable implementation of the transactor semantics. The Maude language is an effective way to produce simple prototypes of the transactor model. Because of the way Maude's rewrite system works, transactor programs running under the Maude transactor implementation display all information, making it easy to examine exactly what the model is doing and to ensure that it is behaving as expected. Prototypes written in the Maude language were used to discover and illustrate safety bugs in previous versions of the model.

1.4 Proof Verification with Athena

Athena [3] is a multi-sorted first-order logic proof system which supports *Fitch* style natural deductions. This style of deduction is similar to the way people reason naturally (as opposed to sequent-oriented proof systems), making Athena proofs easier to read and write by those familiar with informal computer science proofs such as those that were initially used in the transactors POPL paper [5]. Support for multiple sorts makes Athena well-suited towards representing the semantic domains of transactors. Athena supports not only checking proofs but also supports automated theorem proving, useful in automating simple steps within a proof, thereby making such proofs far easier to read by avoiding tedious steps.

Athena is used to provide a more formal proof of the safety invariant. Because Athena proofs are computer-readable, it is not possible to inadvertently skip steps within a proof. This ensures that proofs do not contain underlying assumptions, something which was an issue in previous versions of safety proofs of the transactor model. As a result, such proofs provide more confidence than proofs written without Athena.

1.5 Programming Actor Systems with SALSA

The Salsa language [9] is built on top of Java and is written as an implementation of the actor model. Using Salsa as part of an implementation of transactors removes the need to re-implement the actor model, making it possible to produce a transactor language by coding only the new features of transactors, i.e. the dependency tracking and checkpointing systems. Further, because the Salsa language compiles to Java, integrating Salsa and Java code is very easy. Since keeping track of dependency information is done using sequential code, that part of the transactor language is implemented in Java rather than Salsa.

1.6 Roadmap of the Thesis

In Chapter 2, we introduce the Maude rewrite system. We discuss Maude's role as a tool for developing an implementation of the transactor model and Maude's role in discovering a bug in the safety of the transactor model as defined in the POPL paper [5].

In Chapter 3, we show the use of the Athena proof language in formalizing a proof that the updated transactor model presented in this chapter follows the safety invariant.

In Chapter 4, we discuss the use of Salsa and Java to develop a higher-level transactor programming language.

Finally, we conclude in Chapters 5 and 6 highlighting related work and describing potential future directions.

```
[vec1]   $\langle e_1, \ldots, e_n \rangle$          $\triangleq$  $\langle e_1, \langle \ldots, \langle e_n, nil \rangle \rangle \ldots \rangle, \quad n > 0$
[vec2]   $\langle\rangle$                            $\triangleq$  $\mathsf{nil}$
[vec3]   $\vec{x}$                                   $\triangleq$  $\langle x_1, \ldots, x_n \rangle$  *for some* $n \geq 0$
[seq]    $e_1; e_2$                                  $\triangleq$  $((\lambda x.\ e_2)\ e_1), \quad x \notin fv(e_2)$
[if1]    **if** $e_1$ **then** $e_2$ **fi**          $\triangleq$  **if** $e_1$ **then** $e_2$ **else** $\mathsf{nil}$ **fi**
[let1]   **let** $x = e_1$ **in** $e_2$ **ni**       $\triangleq$  $((\lambda x.\ e_2)\ e_1)$
[let2]   **let** $\langle x_1, \ldots, x_n \rangle = e_1$ **in**
           $e_2$
         **ni**
           $\triangleq$   **let** $x_1 =$ **fst**$(e_1)$ **in**
                            ...
                              **let** $x_n =$ **fst**(**snd**(...**snd**$(e_1)$...)) **in**
                                $e_2$
                              **ni**
                            ...
                          **ni**
[vabs]   $\lambda \vec{x}.\ e$
           $\triangleq$   $\lambda x'.$ **let** $\vec{x} = x'$ **in** $e$ **ni**, $\quad x' \notin fv(e)$
[msg1]   $\mathsf{msg}\ \vec{x}$                     $\triangleq$  $\langle \mathsf{msg}, \vec{x} \rangle$
[msg2]   **msgcase**
           $\mathsf{msg}_1\ \vec{x}_1 \ \Rightarrow\ e_1$
         | ...
         | $\mathsf{msg}_n\ \vec{x}_n \ \Rightarrow\ e_n$
         **esac**
           $\triangleq$   $\lambda \langle m, z' \rangle.$ (
                            **if** $m = \mathsf{msg}_1$ **then**
                              **let** $\vec{x}_1 = z'$ **in** $e_1$ **ni**
                            **else**
                              ...
                              **if** $m = \mathsf{msg}_n$ **then**
                                **let** $\vec{x}_n = z'$ **in** $e_n$ **ni**
                              **else**
                                **ready**
                              **fi**
                              ...
                            **fi**;
                            **ready** )
[sta1]   **declstate** $\langle u_1, \ldots, u_n \rangle$ **in** $e$ **etats**
           $\triangleq$   *declaration of names for* $n$ *elements of state*
[sta2]   $!u_i$
           $\triangleq$     **let** $\langle x_1, \ldots, x_i, \vec{z} \rangle =$ **getstate in** $x_i$ **ni**
                            $u_i$ *is the ith name declared in the closest*
                            *statically-enclosing* **declstate** *scope, of length* $n$, $n \geq i > 0$
[sta3]   $u_i := e$
           $\triangleq$     **setstate**$(\langle !u_1, \ldots, !u_{i-1}, e, !u_{i+1}, \ldots, !u_n \rangle)$
                            *where* $u_i$ *is the ith name declared in the closest*
                            *statically-enclosing* **declstate** *scope, of length* $n$, $n \geq i > 0$
```

Figure 1.10: Defined forms.

```
let cell = trans
   declstate ⟨contents⟩ in
      msgcase
         set⟨val⟩ =>
            contents := val
      |  get⟨customer⟩ =>
            (send data⟨!contents⟩ to customer)
      esac
   etats
init
   ⟨0⟩
snart
```

Figure 1.11: An unreliable reference cell

```
let pcell₁ = trans
   declstate ⟨contents⟩ in
      msgcase
         initialize⟨⟩ ⇒
            stabilize;
            checkpoint
      |  set⟨val⟩ ⇒
            contents := val;
            stabilize;
            checkpoint
      |  get⟨customer⟩ ⇒
            stabilize;
            (send data⟨!contents⟩
                  to customer);
            checkpoint
      esac
   etats
init
   ⟨0⟩
snart
```

Figure 1.12: A persistent reference cell which assumes stable, independent clients

```
let $pcell_2$ = trans
  declstate $\langle contents \rangle$ in
    msgcase
      initialize$\langle\rangle \Rightarrow$
        stabilize;
        checkpoint
    | set$\langle val \rangle \Rightarrow$
        $contents := val$;
        if dependent? then
          rollback
        else
          stabilize;
          checkpoint
        fi
    | get$\langle customer \rangle \Rightarrow$
        stabilize;
        (send data$\langle !contents \rangle$
             to $customer$);
        checkpoint
    esac
  etats
init
  $\langle 0 \rangle$
snart
```

Figure 1.13: A persistent reliable reference cell

2. Executable Operational Semantics

2.1 From τ-calculus to Maude Rewrite Systems

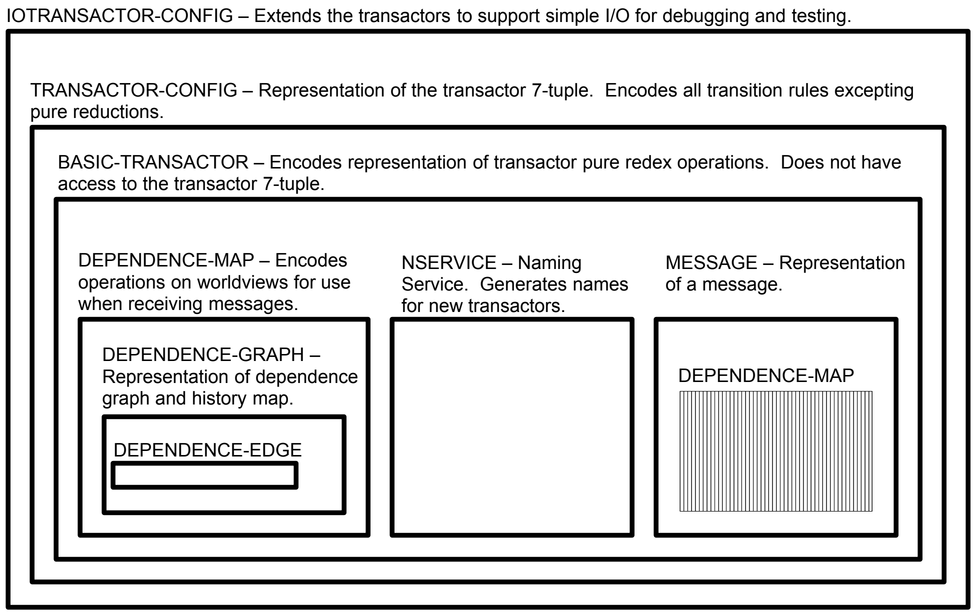
IOTRANSACTOR-CONFIG – Extends the transactors to support simple I/O for debugging and testing.
TRANSACTOR-CONFIG – Representation of the transactor 7-tuple. Encodes all transition rules excepting pure reductions.
BASIC-TRANSACTOR – Encodes representation of transactor pure redex operations. Does not have access to the transactor 7-tuple.
DEPENDENCE-MAP – Encodes operations on worldviews for use when receiving messages.
NSERVICE – Naming Service. Generates names for new transactors.
MESSAGE – Representation of a message.
DEPENDENCE-GRAPH – Representation of dependence graph and history map.
DEPENDENCE-MAP
DEPENDENCE-EDGE

Figure 2.1: Layout of the Maude implementation

Rewriting logic is particularly effective at representing concurrent changes and thus works well in representing models of distributed computing. Further, Maude's power as a logical language makes it effective at representing semantic domains through the use of sorts and logical functions. So, Maude is well-suited towards representing transactors. For example, consider the differences between the formal definition of relations on

```
*** Transactor is volatile: setting state succeeds.
crl[set1] : Universe(SetM|NS, t|- > Transactor(b, P1; R1[setstate(S2)], S1; (lk1, t|- > h), ds1, db1)) =>
  Universe(SetM|NS, t|- > Transactor(b, P1; R1[eval(b, true)], S2; (lk1, t|- > h), (ds1, rootlink(t, db1)), db1))
  if isVolatile(h) .

*** Transactor is stable: setting state fails.
crl[set2] : Universe(SetM|NS, t|- > Transactor(b, P1; R1[setstate(S2)], S1; (lk1, t|- > h), ds1, db1)) =>
  Universe(SetM|NS, t|- > Transactor(b, P1; R1[eval(b, false)], S1; (lk1, t|- > h), ds1, db1))
  if isStable(h).
```

Figure 2.2: Transactor-config module: Maude code for `setstate` transitions

```
op setstate(_) : $TState-> $ Redex [ctor format(ys os d d d)].
op setstate(_) : Redex-> Redex [ctor].
op didset : -> Handler [ctor].
op setstate($R$) = drawstate($R$, $didset$).
op eval($T$, draw($didset$, $I$)) = setstate(declstate(T2 $I$)).
```

Figure 2.3: Basic-transactor module: Maude code for `setstate` redex

```
pr DEPENDENCE - MAP.
pr QID.
sorts Message Data.
var $T$ : Qid.
var $M$ : DMap.
var $D$ : Data.
op $M_$ : Nat-> Data [ctor].
op _ < - < _; _, _, _ >: Qid Data DMap DGraph Set{Qid}-> Message [ctor].
```

Figure 2.4: Message module: Maude code for transactor messages

histories and the Maude definition of relations on histories shown on Table 2.1. There are only minor differences between the representations of the histories, such as using sN (successor of N) rather than $N + 1$.

The Maude implementation uses multiple modules to represent the different components of the transactor model. Figure 2.1 shows a diagram of how these modules are organized. At the highest level is the representation of the transactor configurations themselves. The `TRANSACTOR-CONFIG` module encodes the program as a whole. By design, the pure redex operations are encoded in a separate module, which the `TRANSACTOR-CONFIG` modules makes use of. For example, consider the the `setstate` command in Figure 2.2 and Figure 2.3. Whereas `TRANSACTOR-CONFIG` encodes the transition rules for the `setstate` command, `BASIC-TRANSACTOR` encodes the redex itself. This redex is used within the code for the transition rules. Because the `BASIC-TRANSACTOR` module, is separate, it is not possible for standard redexes to access the transactor configuration. In the same way, a programming language realization of the transactor model would not enable access to the transactor configurations, since these are handled by the language itself. Contained within the `BASIC-TRANSACTOR` module is a module for representing worldviews (`DEPENDENCE-MAP`), a module for representing the naming service(`NSERVICE`), and a module for representing mes-

Relation	History Operations (Semantics)			History Operations (Maude)		
(“rolls back to”)	$\langle \mathbf{V}(n), l_h \rangle$	$\looparrowright$	$\langle \mathbf{V}(n+1), l_h \rangle$	$< v(N), L >$	=>	$< v(sN), L >$
(“rolls back to”)	$\langle \mathbf{S}(n), l_h \rangle$	$\looparrowright$	$\langle \mathbf{V}(n+1), l_h \rangle$	$< s(N), L >$	=>	$< v(sN), L >$
(“stabilizes to”)	$\langle \mathbf{V}(n), l_h \rangle$	$\rightarrow_{\Diamond}$	$\langle \mathbf{S}(n), l_h \rangle$	$< v(N), L >$	=>	$< s(N), L >$
(“checkpoints to”)	$\langle \mathbf{S}(n), l_h \rangle$	$\rightarrow_{\surd}$	$\langle \mathbf{V}(0), n :: l_h \rangle$	$< s(N), L >$	=>	$< v(0), N.L >$

Table 2.1: Comparison of formal semantics of histories operations and Maude definitions of history operations

```
fmod DEPENDENCE − INFO is
  pr NAT.
  pr NAT − LIST.

  sort DepInfo.

  op < v(_), _ >: Nat NatList− > DepInfo.
  op < s(_), _ >: Nat NatList− > DepInfo.
```

Figure 2.5: Maude code for history representation.

sages(`MESSAGE`). The code for the messages is shown in 2.4. Since messages contain worldviews, the latter module is also composed of the `DEPENDENCE-MAP` module. Finally, the `IOTRANSACTOR-CONFIG` module is a wrapper which adds input and output functionality.

Differences between the Maude implementation of the model and the formal definition of the model are hidden through the use of Lex/Yacc, which are used to convert persistent behaviors (*i.e.*, programs) into the format understood by Maude. For example, compare Figure 2.6 to Figure 1.11. The Maude representation makes use of a functional module to represent the behavior. Individual message handlers are defined through the use of equations for each type of message. Of special note is that the Maude implementation treats each individual message handler as a separate equation with the behavior name as an argument, as opposed to the model which treats the behavior as having a set of message cases. This is done for implementation reasons, but can also be used to model behaviors which share some methods through inheritance (*e.g.*, if a particular message handler is a specialization of another). Finally, the Maude implementation provides a `TRcell` operation which can be used to generate a transactor which has the behavior defined within the module. Note that in Figure 2.6, the `TRcell(Q1,true)` function is not really useable as no default `'initialize` message handler is defined. Otherwise, it would be useful as a way to create a transactor instance that would be initialized as though it had just

```
fmod TRcell is
  pr BASIC − TRANSACTOR .
  pr INT .
  op contents : − > Int .
  eq contents  =  0 .

  eq ready [ eval('cell, msg('set.val))] =
    (V contents := (val)).
  eq ready [ eval('cell, msg('get.customer))] =
    (send msg('data.(!V contents)) to customer).

  var val : Impure .
  var customer : Impure .
  var Q1 : Qid .

  op TRcell(_, _) : Qid Bool − > NSEntry .
  eq TRcell(Q1, false) = Q1|− >
    Transactor(
      'cell, nill;
      ready, declstate(T2 0)
      Q1 |− > h0 @ Empty, Gempty, Gempty).
  eq TRcell(Q1, true) = Q1|− >
    Transactor(
      'cell, nill;
      ready[eval('cell, msg('initialize))], declstate(T2 0)
      Q1 |− > h0 @ Empty, Gempty, Gempty).
endfm
```

Figure 2.6: A basic reference cell in Maude (auto-generated)

received an `'initialize` message [1].

One can easily look at the Maude code in sections. In the first section, modules for making use of transactors and integers are loaded. This allows the `TRcell` module to make use of transactors and integer semantics. Immediately below this, a `contents` function is defined. In the τ-calculus, state is defined as a vector of elements[2]. The program assigns a number to each element of the vector, starting with 0. This enables code within the behavior to access the transactor state (*i.e.*, the contents of the reference cell) using the `contents`, rather than using the number 0. Internally there is no actual contents variable; the state is a vector with no named elements. So, this representation could also be considered syntactic sugar, but it is used to make the generated code easier to read rather than to make code easier to write; the original code written by the user

[1]A `'` mark is used in Maude to indicate quoted identifiers, which are similar to character strings.
[2]The state variable is actually syntactic sugar for a lambda expression.

$$\begin{array}{lcll} \mathcal{M} & = & \textbf{eq ready}[\textbf{eval}(\mathcal{A}, \textbf{msg}(\mathcal{L}))] = \mathcal{E} & \textit{Message Handler} \\ \mathcal{L} & = & \mathcal{A} \mid \mathcal{V}.\mathcal{L} & \textit{List of Values} \end{array}$$

Figure 2.7: Maude message handler grammar

makes use of variable names, which the generated code could have discarded.

The next section of the code represents the actual message handlers. Each type of message has an individual handler using the syntax described in Figure 2.7 to define the message handler function prototype and Figure 1.7 to define the message handler function itself.

In the case of the reference cell example in Figure 2.6, there are two message handlers, one being used for `set` messages and one being used for `get` messages. Unlike the semantics defined in the τ-calculus, this representation has the side effect that name collisions can occur between same-named behaviors; it is the responsibility of the user to give individual behaviors different names.

The final section of the code contains a special operator for generating transactor instances from transactor behaviors. Two versions of the operator are generated, one being for uninitialized transactors and the other being used for initialized transactors. The `Q1` argument is used as the name for the transactor instances. As the behavior code indicates that reference cells are initialized to zero, the initial state of the transactors is `declstate(T2 0)` in both cases. `T2` is prefixed before all elements of the state vector (*e.g.*, `declstate(T2 1 T2 3)`) and acts as a wrapper to enable users to define arbitrary types of state data.

The Maude implementation can be used to test transactor behaviors. Let us consider the differences between three types of reference cells: The standard reference cell, the persistent reference cell, and the persistent reliable reference cell. The reliable reference cell behaves similarly to the traditional reference cell, but maintains a promise not to rollback by stabilizing and checkpointing after processing each `set` message. Further, it stabilizes before responding to `get` messages and checkpoints afterwards, ensuring that transactors which send requests to the cell will not gain any new dependencies which could potentially rollback. However, it becomes unusable if it receives a message with any new dependencies as this compromises its ability to checkpoint. The persistent, reliable reference cell rejects such messages by rolling back upon receiving them.

Consider the differences in behavior of the three reference cells upon receiving `set` messages which include or do not include volatile dependencies. The standard reference cell will set its state in response to either message, but will not checkpoint (i.e. a message with volatile dependencies, as shown in the 3rd item in Figure 2.8). Because the standard reference cell does not checkpoint, any transactors which send it `get` messages in the future will lose the ability to checkpoint if the responses are used to alter state. Thus, a failure of such a reference cell will result in a failure of any transactors which depend on it. The persistent reference cell also accepts either type of message. However, messages which introduce new dependencies will freeze the state of the transactor until these dependencies become independent or rollback. Further, transactors requesting state information will gain these new dependencies. The persistent, reliable reference cell rejects messages with new dependencies.

Figure 2.8 illustrates the results of sending such messages, and is copied from the output of the Maude program (but written in the transactors semantics to promote readability). Note that the starred persistent, reliable reference cells start out with a persistent state and a checkpointed history; this is necessary to prevent it from being annihilated if it receives a message which causes it to roll back. In the first two examples, the volatile state of the transactor changes and new dependencies are introduced, but the third example, which is a reliable reference cell, rejects the message by rolling back. The fourth example behaves the same as the first: it accepts the message and sets it state, but does not checkpoint. Both the fifth and sixth examples checkpoint upon receiving the message.

2.1.1 Lessons Learned

We were not able to use the Maude implementation of the operational semantics for model checking. Because any transactor configuration within an execution trace can experience spontaneous failures (*i.e.*, node failures and message loss) at any time, including consecutively, the branching factor of the execution trace made model-checking impossible with spontaneous node failures as part of the model. Thus, it was not possible to perform model-checking on the complete model. One solution to this problem would be to use PMaude [2] in order to probabilistically execute the model, rather than exhaus-

Reference Cell receiving a message with volatile dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, \emptyset, 2 \rangle$
Transactor: $1 \xrightarrow{f} \langle \mathsf{Cell}, \emptyset\ ;\ \mathsf{ready}, 2\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{Cell}, \emptyset\ ;\ \mathsf{ready}, 11\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, 2 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], [1 \leftarrow 2], [1, 2] \rangle$

Persistent Reference Cell receiving a message with volatile dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, \emptyset, 2 \rangle$
Transactor: $1 \xrightarrow{f} \langle \mathsf{PCell}, \emptyset\ ;\ \mathsf{ready}, 2\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{PCell}, \emptyset\ ;\ \mathsf{ready}, 11\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, 2 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], [1 \leftarrow 2], \emptyset \rangle$

Persistent Reliable Reference Cell receiving a message with volatile dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, \emptyset, 2 \rangle$
*Transactor: $1 \xrightarrow{f} \langle \mathsf{PRCell}, 3\ ;\ \mathsf{ready}, 3\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), 0 \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{PRCell}, 3\ ;\ \mathsf{ready}, 3\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(1), 0 \rangle], \emptyset], \emptyset \rangle$

Reference Cell receiving a message with stable dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{S}(0), \emptyset \rangle, \emptyset, 2 \rangle$
Transactor: $1 \xrightarrow{f} \langle \mathsf{Cell}, \emptyset\ ;\ \mathsf{ready}, 2\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{Cell}, \emptyset\ ;\ \mathsf{ready}, 11\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle, 2 \xrightarrow{f} \langle \mathbf{S}(0), \emptyset \rangle], [1 \leftarrow 2], [1, 2] \rangle$

Persistent Reference Cell receiving a message with stable dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{S}(0), \emptyset \rangle, \emptyset, 2 \rangle$
Transactor: $1 \xrightarrow{f} \langle \mathsf{PCell}, \emptyset\ ;\ \mathsf{ready}, 2\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), \emptyset \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{PCell}, 11\ ;\ \mathsf{ready}, 11\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), 0 \rangle], \emptyset, \emptyset \rangle$

Persistent Reliable Reference Cell receiving a message with stable dependencies:
Message: $1 \Leftarrow \langle \mathsf{set}(11)\ ;\ 2 \xrightarrow{f} \langle \mathbf{S}(0), \emptyset \rangle, \emptyset, 2 \rangle$
*Transactor: $1 \xrightarrow{f} \langle \mathsf{PRCell}, 3\ ;\ \mathsf{ready}, 3\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), 0 \rangle], \emptyset, \emptyset \rangle$
$\Rightarrow$
Result: $1 \xrightarrow{f} \langle \mathsf{PRCell}, 11\ ;\ \mathsf{ready}, 11\ ;\ [1 \xrightarrow{f} \langle \mathbf{V}(0), 0 \rangle], \emptyset], \emptyset \rangle$

Figure 2.8: Example results of sending a message to differing types of reference cells

tively testing every branch using standard Maude. The Maude implementation did act as an effective simulation of the transactor model, allowing execution traces for programs to be examined by hand to see what the model is doing. In the course of examining these execution traces for correctness, the execution trace shown in Section 3.1.1 was discovered, thus showing that the model contained a safety error.

3. Towards Proving the Safety of the Transactor Model

3.1 The Safety Invariant and Proof

In this section we discuss the safety invariant. We first show that the POPL 2005 version of the transactor model [5] is not safe. The updated version is shown in Section 1.2. We first show an example execution trace for the POPL 2005 model which fails to maintain safety. In the next section, we discuss what constitutes a safe execution trace. We then define the safety invariant and show that it is necessary for the safety invariant to be an actual invariant for the safety property to hold.

3.1.1 An Example of Violating Safety

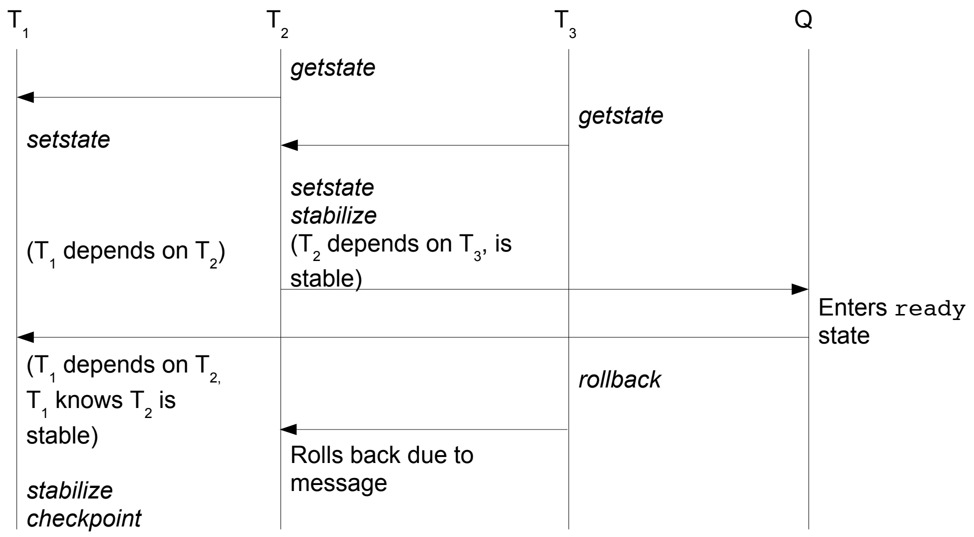

Figure 3.1: A demonstration of failed safety on the POPL transactors paper [5].

Figure 3.1 shows a simple example of communications between three transactors T_1, T_2, and T_3. Take note that the POPL version of the transactor model [5] did not track indirect dependencies. Instead, each transactor tracked only its direct dependencies. In the case that a transactor is permanent, messages it sends contain only its behavioral dependencies (analogous to the *root set* described in Section 1.2.2.2) and its own history. Initially, T_2 sends a message to T_1, causing T_1 to set its state and become dependent on

T_2. T_2 receives a message from T_3 and sets its state, making it dependent on T_3. T_2 also stabilizes. Next, T_2 sends a message to Q, which ignores the message. However, Q now knows that T_2 is stable. When Q sends a message to T_1, T_1 learns that T_2 is stable but does not learn about T_2's dependencies. It is important to note that this is a violation of the safety invariant. Thus, T_1 will be able to stabilize and checkpoint. However, T_2 may roll back in the future, as it is dependent on T_3 which may programmatically rollback or fail. In this case, the sequence of operations in Figure 3.1 could not be simulated without rollbacks, because T_1 now has a state which depends upon T_2 having sent it a message. This in turn depends upon the state of T_2 when that message was sent. However, T_3 rolled back (as did T_2), so T_2 has a state that would not have caused T_2 to send a message to T_1 whereas T_1 has a state which depends upon T_2 having sent T_1 a message. Note that this error was discovered by examining the results of simulated transactor programs under the Maude implementation of transactors.

3.1.2 The Safety Invariant: High Level Description

We define the transactor model as *safe* if it is possible to simulate an execution trace without including node failures. For example, suppose we had a transactor configuration k containing two transactors T_1 and T_2, both of which had just checkpointed. Thus, T_1 and T_2 are both volatile, have no dependencies, and have persistent states equal to their volatile states. Now, suppose T_1 applies a get state operation, then sends a message to T_2, which then sets its state as part of handling the message. Then, T_1 rolls back, and T_2 eventually rolled back as well (as it depends on T_1). Let us call this new configuration k'. This could be simulated by starting at k, but not sending the message. So, configuration k' is reachable without having failures. This property is important because it means that transactor failures are transparent to the programmer.

For illustrative purposes, we show an example of an execution trace that may exist which would violate safety, similar to the example in Figure 3.1. As with the previous example, we start with two just checkpointed transactors T_1 and T_2 in some configuration C. Both initially have state "0." T_1 receives a message from some volatile transactor T_3, then sets its state to "T3-INFO" in response. T_1 gets its state and sends a message to T_2 with contents, "set your state to T3-INFO." Upon receiving this message, T_2 stabilizes and checkpoints, but T_1 rolls back. At this point, T_1 has state “0” but T_2 has state “T3-INFO.” We cannot simulate this without failures, because T_2 setting its state to “T3-INFO” requires T_1 to get its state and then send a message with this data, which is only possible if its state is actually “T3-INFO.” Thus, a failure-free execution trace is not possible in this case because the state of T_1 and T_2 are inconsistent, due to T_2 checkpointing when dependent upon a transactor which rolled back. We can say that this specific example is not possible if we guarantee that no transactor will checkpoint if it depends (directly or transitively) upon a transactor which may potentially rollback.

The safety of the transactor model can be shown to be true by construction. A failure causes a transactor’s state to roll back to the last time it checkpointed, so we construct a failure free trace by removing all transitions between transactor configurations which occurred between a checkpoint and a rollback. A transition which involves sending a message must be treated as a special case, because after such a transition is removed, the state of the direct or indirect recipients of that message may have changed. So, construction of a failure free trace will fail unless the recipients of such a message also roll back, thus removing the transitions which were caused by the altered state of these transactors. We focus on proving that such recipients cannot checkpoint.

We introduce two new definitions. We recursively define a transactor as being *globally independent* if it is stable and all direct dependencies of the transactor are also globally independent. Equivalently, we can define a transactor as globally independent if it would become independent if it had access to every transactor’s worldview. We define the *safety invariant* as follows: For all worldviews w, for all stable transactors t with history h in the history map of w, there exists a configuration s where h is equal to the actual history of t and all direct dependencies in the worldview of t which have never been globally independent during any current or previous configuration with history in t’s history map are contained in the history graph of w. Less formally, the safety invariant states that if a worldview shows a transactor as being stable, that worldview knows all of that transactor’s direct dependencies, except those which are globally independent.

The goal is to show that if the safety invariant is actually an invariant, then a recipient r of a message (where the message causes the state of the recipient to change) from a transactor t which later rolled back cannot possibly checkpoint. We know that t is not globally independent as t later rolled back. Globally independent transactors cannot rollback, as such transactors are stable and all of their dependencies are also stable. We can further state the history map of t contains at least one volatile dependency by applying induction on the definition of global independence and on the safety invariant. That is, at least one of the direct dependencies of t is not globally independent, and if it is stable then all of its direct dependencies that are not globally independent are known to the worldview of t. When r receives the message, it will also receive t's worldview and thus will not be able to checkpoint due to this dependency.

3.1.3 The Safety Invariant: Formal Definition

Let $(\gamma_g, \delta_g, \_)$ be the global worldview of the configuration k, defined as the union over $\oplus$ of all worldviews of all transactors within the configuration, as described in Figure 1.4.

We define a global independence predicate, $independent_h(t, k)$. If a worldview knows about t having history h, then $independent_h(t, k)$ is defined as being true if t is now or was in the past independent with that same history. We formally define it as a call to $independent_h(t, k, \emptyset)$, where $independent_h(t, k, S)$ is defined as follows:

If $h \dashv \gamma_g(t)$, true.
If $h = \gamma_g(t)$, h is stable, and $\forall t'(t, t') \in \delta_g$: ($t' \in S$ or $independent_{\delta_g(t')}(t', K, S \cup \{t'\})$), true.
Otherwise false.

We say that a worldview $(\gamma, \delta, \_)$ in k follows the safety invariant if:
$\forall t \in dom(\gamma)$, if $\gamma(t)$ is stable, then
$\forall (t, t') \in \delta_g$: if $\neg independent_{\gamma_g(t')}(t', k)$, then
$((t, t') \in \delta$ and $(\gamma(t') = \gamma_g(t')$ or $(\gamma(t') \rightarrow_\Diamond \gamma_g(t')))$

A transactor configuration follows the safety invariant if all worldwiews in the configuration follow the safety invariant.

3.1.4 The Safety Invariant: Proof

Lemma 3.1.1 (Deleted Worldview). *If a transactor configuration k follows the safety invariant, a transactor configuration k' which is missing a worldview will also follow the safety invariant.*

Proof. Suppose some transactor configuration k follows the safety invariant. Then consider configuration k' for which a worldview from k has been deleted. Suppose some worldview $(\gamma, \delta, \_)$ in k' now does not follow the safety invariant. Let $(\gamma'_g, \delta'_g, \_)$ be the global worldview of k'. Then there exist transactors t and t' for which $\gamma(t)$ is stable, $(t, t') \in \delta'_g, \neg independent_{\gamma'_g(t')}(t', k')$ but $\neg((t, t') \in \delta$ and $(\gamma(t') = \gamma_g(t')$ or $(\gamma(t') \rightarrow_\Diamond \gamma_g(t')))$ However, the final conjunct is only possible in the case that $(\gamma, \delta, \_)$ is the worldview which was removed (and is a transactor's worldview, rather than a message's worldview), as k followed the invariant. But in this case, $\gamma(t)$ is not stable, as $t \notin dom(\gamma(t))$. By *reductio ad absurdum*, if a transactor configuration k follows the safety invariant, a transactor configuration k' which is missing a worldview will also follow the safety invariant. □

Theorem 3.1.2 (The Safety Invariant). *We prove that the safety invariant is actually an invariant by structural induction. Supposing all worldviews in some transactor configuration k follow the safety invariant. Then, we consider all possible transition rules.*

Proof. (1) Trivially, the invariant holds if a transition rule does not change the worldviews or only makes a copy of a worldview. Thus, the safety invariant holds after the transitions: `pure, get, self, set2, sta2, chk2, rol1, dep1, dep2, snd`.

(2) By Lemma 3.1.1, the safety invariant holds after message loss or transactor annihilation: `lose, fl2, rol3, rcv2, rcv4`.

(3) Consider the case where a transactor t will roll back or checkpoint. By Lemma 3.1.1, deleting the worldview of t is safe. The new worldview also follows the invariant, as it contains no dependencies and t is volatile: `rcv3, rol2, fl1, ck1`.

(4) In the case where a transactor adds new dependencies on a volatile transactor to a worldview, the safety invariant holds since there are no new transactors and all stable transactors did not lose their existing dependencies: `new, set1`.

(5) In the case where a transactor t stabilizes, the invariant holds. Worldviews other than the worldview of t do not know that t is stable. t_1 trivially continues to follow the

safety invariant because a transactor always knows all of its own direct dependencies: `sta`.

(6) Consider the case where a transactor t with worldview $(\gamma_{t_1}, \delta_{t_1}, \_)$ receives a message with worldview $(\gamma_m, \delta_m, \_)$. By Lemma 3.1.1, deleting both of these worldviews is safe. The new worldview $(\gamma_{tm}, \delta_{tm}, \_) = (\gamma_{t_1}, \delta_{t_1}, \_) \oplus (\gamma_m, \delta_m, \_)$ must also follow the safety invariant. Consider each stable transactor $t_2 \in \gamma_{tm}$. Without loss of generality, assume $\gamma(t_1) = \gamma_{tm}$ or $\gamma(t_1) \rightarrow_{\Diamond} \gamma_{tm}$. By the safety invariant, $\forall(t_1, t_2) \in \delta_g$: if $\neg independent_{\gamma_g(t_2)}(t_2, k)$, then $((t_1, t_2) \in \delta_{t_1}$. For $(t_1, t_2) \in \delta_t$ but $(t_1, t_2) \notin \delta_{tm}$, it must be the case that t_1 or t_2 were validated or invalidated by δ_m. In the case that either was invalidated, neither is globally independent (or even stable) with the history from δ_t so $(t_1, t_2) \notin \delta'_k$ (or the histories are different in γ'_g). In the case that t_1 or t_2 were validated, they must be globally independent with respect to their previous history, since a checkpointed form of that history exists in γ_m. So receiving messages follows the safety invariant: `rcv1`. $\square$

Of special note is step (6) of the proof, as this step did not follow the safety invariant in [5]. In particular, a transactor would not always update its dependency information to account for transitive dependencies. As a result, a transactor might be missing some dependency information on other transactors after receiving a message. This was due to the fact that the POPL transactor model only encoded direct dependencies rather than making use of a dependence graph.

3.1.5 Fixing Lack of Safety in the Transactor Model

Of paramount importance to the transactor model is that it be safe. One clear requirement for the safety of the transactor model is thus: If a transactor t_1 depends on a transactor t_2, it is not permissible for t_1 to checkpoint and t_2 to roll back, as this would enable t_1 to reach a state which was based on a non-existent state of t_2, thus resulting in an inconsistent global state. In fact, a significant portion of the transactor model is tracking dependency information to ensure that this does not happen. A previous iteration of the transactor model, as reported in [5], stored only direct dependencies, making the scenario in Figure 3.1 possible. The safety invariant is a requirement that no worldview will ever show a transactor as being independent if that transactor is not

```
(define DefFollowsInvariant
 (forall ?Configuration
  (iff
   (FollowsInvariant ?Configuration)
   (forall?Name ?NameOld ?Worldview
     (if
      (NameinConfigFollowsInvariantAntecedent
       ?Configuration ?Name ?NameOld ?Worldview)
      (NameinConfigFollowsInvariantConsequent ?Configuration ?NameOld ?Worldview)
)))))
```

Figure 3.2: The safety invariant in Athena

```
(define
 (NameinConfigFollowsInvariantAntecedentConfiguration Name NameOld Worldview)
 (and
  (or
   (InT(Transactor Name Worldview) Configuration)
   (InM(Message Name Worldview) Configuration)
  )
  (exists ?History(= (GetHistory  WorldviewNameOld)(Stabilize ?History)))
))
```

Figure 3.3: The safety invariant's antecedent in Athena

globally independent with respect to its history in that worldview. Thus, a transactor's worldview which showed some of the transactor's dependencies were stable would also show all of the direct dependencies of these stable dependencies, in direct contrast to Figure 3.1, in which the transactor does not know these dependencies (and thus can checkpoint).

3.2 Translating the Proof into a Machine-Checkable Athena Proof

3.2.1 The Safety Invariant (Athena)

The safety invariant is defined in Athena at several levels of abstraction. At the most general level, the safety invariant is defined as shown in Figure 3.2, with the antecedent `NameinConfigFollowsInvariantAntecedent` expanded in Figure 3.3 and the consequent `NameinConfigFollowsInvariantConsequent` expanded in Figure 3.4.

Informally, this states that any configuration follows the safety invariant if and only if for all worldviews fulfilling the requirements within

```
1   (define
2    (NameinConfigFollowsInvariantConsequent Configuration NameOld Worldview)
3    (exists ?WorldviewOld ?ConfigurationOld
4     (and
5      (ConfigurationSucceeds* ?ConfigurationOld Configuration)
6      (InT(Transactor NameOld ?WorldviewOld) ?ConfigurationOld)
7      (= (GetHistory ?WorldviewOld NameOld)(GetHistory Worldview NameOld))
8      (forall ?Dependency
9       (if
10       (DependsOn ?WorldviewOld NameOld ?Dependency)
11       (NameOldKnowsADependency Configuration Worldview NameOld
         ?WorldviewOld ?Dependency)
12  )))))
```

Figure 3.4: The safety invariant's consequent in Athena

`NameinConfigFollowsInvariantAntecedent` they also fulfill the requirements within `NameinConfigFollowsInvariantConsequent`.

`NameinConfigFollowsInvariantAntecedent` is defined as having `Worldview` within `Configuration`. This could be either a transactor with name `Name` (line 5) or a message with destination `Name` (line 6). Further, `NameOld` is stable according to `Worldview`. Thus, `NameinConfigFollowsInvariantAntecedent` represents that the safety invariant's requirements apply specifically to transactors which are stable within a worldview.

3.2.2 Machine-Checkable Proof

A proof for Lemma 3.1.1 is shown in Figure 3.5. We start by defining two consecutive configurations `x` and `y`, where every message and transactor in `y` is in `x`. Thus, there are no new worldviews, but some worldviews may have been lost. We also select a transactor `a2` which is known to be stable according to `b2` (using the `KnowsNIisStable function`), and thus must have its direct dependencies known. We can then select an arbitrary one of these dependencies `z`. Thus, `(DependsOn (D b2) a2 z)` may be interpretted to mean, "According to worldview b2, a2 depends on z". Lines `As0` through `As6` are used to separate and reorder the conjuncts in our initial definition (i.e. that there are two consecutive configurations etc.). Line `As7` states that because `a2` was known to be stable according to `b2` in transactor configuration `y`, it was also known in transactor configuration `x`, since that same worldview existed in `x`. Finally, line `As8` makes use of the immutable past lemma, described in Figure 3.9, in order to show that since the worldview followed the safety invariant before, it can still follow it now by making use of the worldviews that existed in the past, even if they do not exist now. This relates to the fact that the Athena proof defines the direct dependencies of a transactor as being the dependencies it had in the past, rather than making use of the global worldview, as is used by the safety invariant definition. Note that the latter definition is stronger, since such direct dependencies must originate from the past on the transactor that has them, but may have been eliminated in the future due to rollbacks or checkpoints.

With the exception of the rule for receiving messages, most steps within the transactor proof are very simple when making use of the lemmas described in Figure 3.5 and Figure 3.9, as the transactor's dependency maps do not significantly change during other transition rules. However, the receive rule makes use of the $\oplus$ operation, which is not easily represented or used in Athena, as it is somewhat akin to an imperative algorithm rather than being a purely logical definition. Unfortunately, we were unable to provide a machine-checkable proof of this rule's correctness for this reason.

Because safety is defined in terms of dependency information, we eschew the usage of behavior and state within our representations of transactors. Instead, we define transactors as having a name and a 3-tuple dependency component. Similarly, messages have a destination and a 3-tuple dependency component. Of particular note for the Athena proof is that last known histories as being associated with particular transactor configurations. That is to say, it is not possible to say a particular transactor or message has a specific dependency component without also indicating which transactor configuration it applies to. Intuitively, this means that a transactor's dependencies are mutable and only exist during specific transactor configurations.

In order to provide a proof of safety, it is necessary to provide a formal definition of safety. We define the safety invariant as follows: If the last known history component of a transactor or message indicates that some transactor t is stable, then the dependency graph component of that transactor or message includes any direct dependencies of t which are not independent. The safety invariant is used to guarantee that if some transactor t depends on some other transactor t', it is not possible for t to checkpoint while t' rolls

```
(define (LemmaLoss x y z a1 b1 a2 b2 )
  (assume
    (and
      (FollowsInvariant x )
      (ConfigurationSucceeds x y)
      (forall ?z(if (InT (Transactor a1 b1)y)(InT (Transactor a1 b1)x )))
      (forall ?z(if (InM (Message a1 b1)y)(InM (Message a1 b1)x)))
      (KnowsNIisStable a1 b1 a2 b2 y)
    )
    (assume (DependsOn (D b2)a2z)
      (dlet
        (
          (As0(!claim (and
            (FollowsInvariant x )
            (ConfigurationSucceeds x y)
            (forall ?z (if (InT (Transactor a1 b1)y)(InT (Transactor a1 b1)x )))
            (forall ?z (if (InM (Message a1 b1)y)(InM (Message a1 b1)x )))
            (KnowsNIisStable a1 b1 a2 b2 y)
           )))
          (As1 (!derive (FollowsInvariant x )[As0]))
          (As2 (!derive (ConfigurationSucceeds x y)[As0]))
          (As3 (!derive (forall ?z (if (InT (Transactor a1 b1)y)(InT (Transactor a1 b1)x )))
            [As0]))
          (As4 (!derive (forall ?z (if (InM (Message a1 b1)y)(InM (Message a1 b1)x )))[As0]))
          (As5 (!derive (KnowsNIisStable a1 b1 a2 b2 x)[As0]))
          (As6 (!derive (or(InT (Transactor a1 b1)x )(InM (Message a1 b1)x ))[As3 As4 As5 ]))
          (As7(!derive (KnowsNIisStable a1 b1 a2 b2 x)[As6 As5 DefESS*]))
        )
        (!derive (and(FollowsInvariant x )(ConfigurationSucceeds x y)
          (KnowsNIisStable a1 b1 a2 b2 x))[(KnowsNIisStable a1 b1 a2 b2 x)As1 As2 ]))
          (!mp(!LemmaInvariant1, x y z a1 b1 a2 b2 )A0)
        )
)))))
```

Figure 3.5: Athena proof that removing a worldview does not violate the safety invariant.

back. For t' to roll back, it must not be independent, so t would have to know about some volatile transactor which t' was dependent upon, so t could not checkpoint. We define safety as follows: If a transactor configuration follows the safety invariant, all subsequent transactor configurations follow the safety invariant.

The antecedent of the informal definition of the safety invariant only states that a transact t was dependent upon some not independent transactor t'. This is insufficient in a formal definition. A formal definition must further state that this dependency occured in some transactor configuration k and that t' has never been independent in any early

```
(define (KnowsNIisStable t1 d1 t2 d2 K)
  (and
    (or
      (InT (Transactor t1 d1)K)
      (InM (Message t1 d1)K)
    )
    (exists?y?K2
      (and
        (ConfigurationSucceeds ∗ ?K2 K)
        (InT (Transactor t2 d2)?K2)
        (= (HistoryMap(H d1)t2)(HistoryMap(Hd2)t2))
        (= (HistoryMap(H d1) t2)(Stabilize?y))
))))
```

Figure 3.6: Athena definition of knowledge of stability.

configuration k_{old}.

Figure 3.6 represents this antecedent. d_1 is the last known history component of some message or transactor. The `InM` and `InT` relations indicate a particular transactor or message with a dependency 3-tuple are within configuration k. Transactor t2 with dependency information d2 is in configuration k_2, which succeeds k. d_2 is the last known history component of another transactor in some other configuration k_2. So, this definition indicates that some dependency component d_1 in configuration k indicates that d_2 was stable, and in some configuration equal to or earlier than k t_2 had a history which matched the history in d_1. The fact that the history matches a real history in some older configuration K_2 means that the history is referring to a specific version of the transactor which existed earlier. It may have since checkpointed or stabilized and have a different history.

The safety invariant defined in Figure 3.7 requires that for any dependency component d_1 which knows that t_2 is stable, every transactor t_2 is dependent upon is either known to d_1 or is independent. As with the antecedent, the consequent must make use if existential quantifiers to match the the transactor z which d_2 is referring to with an actual transactor z. Thus, there must be some previous configuration in which these histories matched. Unlike within informal proofs, it is necessary to explicitly define an association between the stable transactor and the transactor configuration which the history corresponds to.

It is necessary to prove that each individual rule transition is safe. Because the

```
(define DefFollowsInvariant
   (forall K
      (iff
         (FollowsInvariant K )
         (forall t1 d1 t2 d2
            (if
               (KnowsNIisStable t1 d1 t2 d2 K )
               (forall ?z
                  (if
                  (DependsOn (D d2 )t2 ?z )
                  (exists?u
                     (and
                        (InT (Transactor ?z ?r )?u )
                        (ConfigurationSucceeds * ?u K )
                        (= (HistoryMap(H ?r )?z )(HistoryMap(H d1 )?z ))
                        (or
                           (Independent?u?z )
                           (DependsOn (D d1 )t2 ?z )
      ))))))))))
```

Figure 3.7: Athena definition of the safety invariant.

```
(define DefNoop
   (forall ?x?y
      (if
         (ConfigurationSucceedsRule ?x ?y Noop)
         (and
            (forall ?z
               (iff(InT ?z ?x)(InT ?z ?y))
            )
            (forall ?z
               (iff(InM ?z ?x)(InM ?z ?y))
)))))
```

Figure 3.8: Athena definition of the `Noop` transition.

Athena representation of transactors leaves both the behavior and the state of the transactors undefined, several basic rules are essentially `Noop` operations: Evaluating a pure redex, determining a transactor's own name, determining if a transactor is independent, stabilizing a transactor that is stable, and checkpointing a transactor that is not independent. Although such `Noop` operations appear to be trivially safe, proving their safety is important not only for completeness but also because all transition rules involve making changes to only one transactor, one message, or both. Thus, proofs of other rules will make use of similar assumptions for the remaining worldviews.

We can define a no-op operation as a transition between two transactor configurations such a message is in the initial transactor configuration if and only if it is in the successive transactor configuration. The Athena representation of this is shown in 3.8. Supposing some configuration y is the result of applying the `Noop` operations to x. In that case, every transactor z in x is in y. Further, every message z in x is in y.

Proving that the Noop operation is safe can be done by showing that any transactor known in the new transactor configuration to have been stable was previously known to be stable. Intuitively, this must be true because the dependency components of the different transactors and messages is unchanged. The safety invariant (i.e. a dependency component knows all of the direct dependencies of any stable transactor excepting the independent dependencies) is true of the previous transactor configuration. So, any transactor known to have been stable in that configuration will have known dependencies in that configuration. A dependency component which showed such dependencies in a previous configuration will still show them in that previous configuration; the past is immutable, shown in Figure 3.9.

We define `LemmaInvariant1` to show that if a configuration x follows the safety invariant and a particular dependency b_1 shows that transactor a_2 is stable, then all dependencies of that transactor will still be known in subsequent configuration y. It is important to note that this knowledge refers to knowledge from the past configuration. Thus, this invariant does not require that the transactor still exists within the new configuration, though this will be the case for no-op operations. The initial steps within the proof are to separate out the assumptions using conjunction elimination. The statements within the `dlet` clause are used to conclude the consequent of the safety invariant. That is, they are used to show that all not independent dependencies of stable transactor a_2 are known. The conclusion is similar, but take note that the conclusion allows for these dependencies to be known in some configuration up to and including configuration y whereas the `dlet` clause derived it as being known in a subset of this: up to and including y's predecessor, x. Thus, the conclusion contains `(ConfigurationSucceeds* ?u y)` whereas what was given was `(ConfigurationSucceeds* ?u x)`. The automated theorem prover is used to prove this based on the earlier assumptions and based on the definition of `ConfigurationSucceeds*`: (`DefESS*`).

```
(define (LemmaInvariant1 x y z a1 b1 a2 b2)
  (assume (and(FollowsInvariant x )(ConfigurationSucceeds x y)
    (KnowsNIisStable a1 b1 a2 b2 x))
    (dlet
      (
        (As1 (!left − and(and(FollowsInvariant x )
          (ConfigurationSucceeds x y)(KnowsNIisStable a1 b1 a2 b2 x))))
        (As2 (!left − and(!right − and(and(FollowsInvariant x )
          (ConfigurationSucceeds x y)(KnowsNIisStable a1 b1 a2 b2 x)))))
        (As3 (!right − and(!right − and(and(FollowsInvariant x )
          (ConfigurationSucceeds x y)(KnowsNIisStable a1 b1 a2 b2 x)))))
        (P1 (!left − iff(!uspecDefFollowsInvariantx)))
        (P2 (!mp P1 As1))
        (DD (!uspec(!uspec(!uspec(!uspecP2 a1)b1)a2)b2))
        (FOO1 (!mpDD As3))
        )
        (assume (DependsOn (D b2)a2 z)
          (!derive
            (exists ?r ?u
              (and
                (InT (Transactor z?r)?u)
                (ConfigurationSucceeds ∗ ?uy)
                (= (HistoryMap(H ?r) z)(HistoryMap (H b1)z))
                (or
                  (Independent ?u z)
                  (DependsOn (D b1)a2 z)
            )))
            [(DependsOn (D b2)a2 z)FOO1 As2 DefESS∗]
    )))))
```

Figure 3.9: The immutable past lemma

4. A High-Level Transactor Programming Language

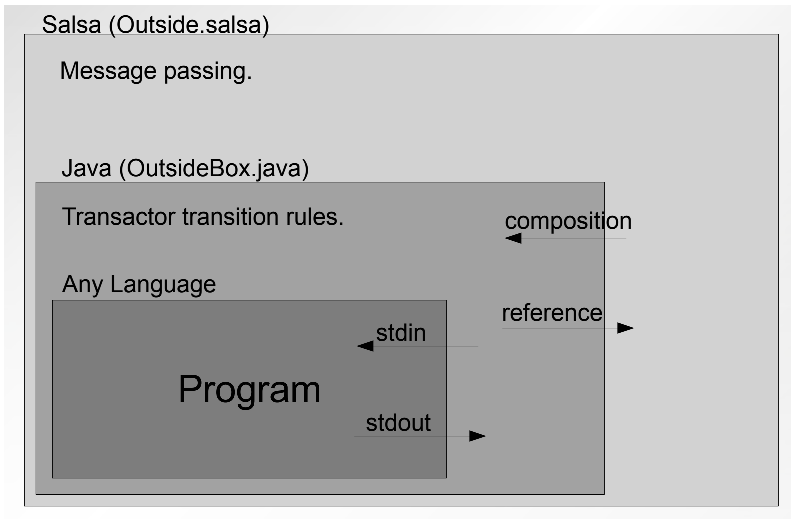

Figure 4.1: Coordination language diagram

Clearly, one goal in producing the *transactor* model should be the potential for programmers to write production languages which make use of *transactors*. Since the model is in fact an extension of actors, it makes sense to extend an existing actor language such as Salsa in order to support transactors. We propose a coordination language which makes use of Salsa and Java to enable the production of transactor programs written in any language.

4.1 Usage

Transactor behaviors can be written in any language, and are not run directly. Instead, the user runs the coordination language, which executes a process that runs the transactor behavior language each time a message is received. These languages may be used as normal, but it is worth noting that functions which produce side effects such as file input and output may result in state information which survives rollbacks, thus compromising the dependency system. Program outputs can be stored as state information rather than in files to avoid this, but this may be inefficient. Standard input and output can be used to allow the program to indirectly access transactor commands in order to

`Stabilize`	Causes actor to stabilize
`Getstate`	Gets state. Program state is received on standard input.
`Setstate <state>`	Sets state (Unlike the model, no acknowledgement is currently received)
`SendTo <destination> <message>`	Sends a message. <Destination> should be a UAN.
`Rollback`	Attempts a programatic rollback.
`Checkpoint`	Attempts to checkpoint.
`Dependent?`	Program receives independent/dependent status on standard input.
`Self`	Program's UAN is received on standard input.
`Ready`	Causes transactor to enter the ready state.
`New <behavior> <state>`	Creates a new actor. The new actor's UAN is received on standard input.
`Echo <output>`	The theater running the transactor displays <output> on its standard output.

Table 4.1: Commands used in transactor implementation

perform tasks such as message-passing and checkpointing.

4.1.1 Commands

Programs can send signals along standard output to access transactor functionality. Note that the commands are case sensitive. Arguments are in capitals and are terminated by an end line. The available commands are listed in Table 4.1.

4.1.2 Example Program

Figure 4.2 demonstrates the use of several common transactor commands. Take note that the program itself is purely written in Java. Messages are received as plain text strings. In this case, the program is written to interact with a theoretical program which can send three messages: `Crash.`, `set <argument>`, and `get`. It echoes messages it does not understand to the console of the theater running it by making use of the `Echo` command. `Crash.` merely quits out of the program without indicating that the program is ready to receive new messages. `Crash.` is functionally equivalent to a programmatic rollback, and is transparent to the user if the transactor is stable, since it will merely reload its state upon restarting. Because it is running in a separate process, such a crash will not result in crashing the coordination language itself. Of special note is the fact that the behavior makes use of a while loop rather than an if statement in order to receive messages. Because the program is rebooted any time it receives a new message, the while loop will

```
String fromServer;
while((fromServer = stdIn.readLine())! = null){
  if(fromServer.equals("Crash.")){
    return;
  }, else if(fromServer.length() > 4 && fromServer.substring(0, 4).equals("set")){
    System.out.println("Setstate" + fromServer.substring(4));
    System.out.println("Stabilize");
    System.out.println("Checkpoint");
  }, else if(fromServer.length() >= 3 && fromServer.substring(0, 3).equals("get")){
    System.out.println("Getstate");
    String state = stdIn.readLine();
    System.out.println("Echo My state is: " + state);
    System.out.println("Ready");
  }, else if(fromServer.equals("Self")){
    System.out.println("Self");
    System.out.println("Echo I am " + stdIn.readLine());
    System.out.println("Ready");
  }, else{
    System.out.println("Echo " + fromServer);
    System.out.println("Ready");
  }
}
```

Figure 4.2: A simple transactor behavior in Java

never actually repeat, though from a user perspective it does repeat since the program as a whole is executed over and over again as new messages are received. The `BoxProgram` class is intended for use as a destination for final results rather than as a mechanism for data storage. Unlike traditional reference cells, it cannot communicate its contents to other transactors and instead outputs its results to the console if it receives a *get* message. Using a special output transactor is superior to allowing arbitrary transactors to read and write to files in that it reduces the likelihood that state information will incorrectly survive in between messages. As written, the `BoxProgram` class is not safe for use in output as it could receive a `set` message from a dependent transactor, causing the stabilize and checkpoint steps to fail but still allowing it to temporarily set its own state to a value that may be rolled back. Because output will not be rolled back with the transactor, this is a potential concern.

4.2 Implementation

We make use of the existing actors implementation within Salsa in order to support transactor semantics which are inherited from actors such as message-passing. However, we use Java in order to implement dependency information, checkpointing, and rollbacks because the algorithms used to track this functionality are implemented sequentially. Actor behaviors can be implemented by users in any language. Users programs can communicate with the Java layer of the transactors implementation through the use of standard input/output. The system prevents programs from improperly saving state information between runs by rebooting.

4.2.1 Salsa Layer

The outermost layer of the implementation is a Salsa actor. The Salsa layer of the transactor program handles sending messages, receiving messages, and making new actors. It also provides the ability to discover an actor's name within the actor name service, which is used as an identifier for sending messages to actors. The Salsa layer contains an instance of the Java class used to hold and manipulate the transactor information. The facilities for making new transactors, sending messages, and getting the transactor's name act as wrapper functions which are called by the Java layer of the program at need. An initialization function is called at startup to start the program if needed. Finally, the function for receiving messages is called directly by Salsa in order to process messages sent via the send message function wrapper, `sndMessage`. Message sending is in the form of plain text `Strings` paired with dependency information. Upon receiving a new message, the Salsa layer forwards the message to the Java layer, which processes the message. The code for receiving messages, sending messages, making new transactors, and finding a transactor's name is displayed in Figure 4.3. Notice the use of the Java class, `OutsideBox`, to contain messages.

4.2.2 Java Layer

The Java layer of the actors implementation acts to facilitate communication between the transactor behavior (*i.e.*, the program) and the transactor system. The Java layer does not act as an API. Instead, each time a message is received, this layer executes a new

```
void rcvMessage(String mymessage , deptup d){
   single = new OutsideBox(this);
   int result = single.run(mymessage, d, OutsideBox.READY);
   if (result == OutsideBox.CRASH){
      System.out.println("The program crashed.");
   }, else if (result == OutsideBox.ROLLBACK){
      System.out.println("The program rolledback.");
   }
   if (single.dep.h.get(getUAN().toString()).isEphemeral()&&
   !single.dep.h.get(getUAN().toString()).isStable()){
      System.out.println("Ephemeral. Quitting");
      return ;
   }
}
void sndMessage(String mymessage, String recipient, deptup d){
   try{
      Outside remoteSVC = (Outside)Outside.getReferenceByName(recipient);
      remoteSVC < −rcvMessage(mymessage, d);
   }, catch(Exceptione){
      e.printStackTrace();
   }
}
String makeNew(String type, String state, deptup d){
   String appendo = "/" + String .valueOf(+ + count);
   String newUan = getUAN().toString() + appendo;
   String newUal = getUAL().toString() + appendo;
   Outside actor = new Outside(type, state, d) at (newUAN(newUan), newUAL(newUal));
   return newUan;
}
String getmyself(){
   return getUAN().toString();
}
```

Figure 4.3: Salsa code for transactors

instance of the relevant message handler. The message handler may be written using any language and communicates with the Java layer using standard input and output. The Java layer maintains the transactor worldview and contains code for each transition rule used by transactors. When a message is forwarded to this layer by the Salsa layer, the message handler program is executed and a loop is entered which continually reads from the handler's standard output for any further instructions. The code for the Java layer redexes is shown in Figure 4.4 and 4.5. The loop ends when the message handler finishes parsing the message, which ends either by sending `Ready`, `Checkpoint`, or `Rollback` to standard output. Thus, a message handler for a reference cell could respond to a request to

```
while ((inputLine = InnerBR.readLine())! = null){
  System.out.println("Client: " + inputLine );
  if (inputLine.equals("Stabilize")){
    System.out.println("Server: Received Stabilize, stabilizing");
    dep.h.get(getSelf()).stabilize();
    backup();
  }, else if (inputLine.equals("Getstate")){
    System.out.println("Server: Received Getstate, sending state:" + state);
    dep.b.add(getSelf());
    innerOut.println(state);
  }, else if (inputLine.length() > 9 &&
  inputLine.substring(0, 9).equals("Setstate ")){ if (!dep.h.get(getSelf()).isStable()){
      System.out.println("Server: Am volatile, State set to \"" + inputLine.substring(9) + "\"");
      state = inputLine.substring(9);
      Iterator < String > newstuff = dep.b.Iterator();
      if (dep.d.get(getSelf()) == null){
        dep.d.put(getSelf(), newHashSet < String > ());
      }
      while (newstuff.hasNext()){
        dep.d.get(getSelf()).add(newstuff.next());
      }
    }, else {
      System.out.println("Server: Am stable, unable to set State to "+
        inputLine.substring(9) + "\"");
    }
  }, else if (inputLine.length() > 7 && inputLine.substring(0, 7).equals("SendTo ")){
    String more = inputLine.substring(7);
    int i;
    for (i = 0; i! = more.length(); + + i){
      if (more.substring(i, i + 1).equals(" ")){
        break;
      }
    }
    System.out.println("Server: Told to send \"" + more.substring(i + 1)
      +"\"to" + more.substring(0, i));
    parent.sndMessage(more.substring(i + 1), more.substring(0, i), dep);
  }, else if (inputLine.equals("Rollback")){
    if (!dep.h.get(getSelf()).isStable()){
      System.out.println("Server: Received Rollback, rolling back.");
      return ROLLBACK;
    }, else {
      System.out.println("Server: Received Rollback but not volatile.");
      return READY;
    }
```

Figure 4.4: Java code for transactors

```
}, else if (inputLine.equals("Checkpoint")){
   if (independent()){
      System.out.println("Server: Received Checkpoint, am independent, checkpointing");
      history tmp = dep.h.get(getSelf());
      dep = newdeptup();
      tmp.checkpoint();
      dep.h.put(getSelf(), tmp);
      backup();
      return READY;
   }, else {
      System.out.println("Server: Received checkpoint but am not independent.");
   }
}, else if (inputLine.equals("Dependent?")){
   if (independent()){
      System.out.println("Server: Received Dependent?, false.");
      innerOut.println(false);
      }, else {
         System.out.println("Server: Received Dependent?, true.");
         innerOut.println(true);
         }
      }, else if (inputLine.equals("Self")){
         System.out.println("Server: Received Self");
         innerOut.println(parent.getmyself());
      }, else if (inputLine.equals("Ready")){
         System.out.println("Server: Received Ready. Creating new instance of program");
         return READY;
      }, else if (inputLine.length() > 4 && inputLine.substring(0, 4).equals("New ")){
         String type = inputLine.substring(4);
         String state = InnerBR.readLine();
         innerOut.println(parent.makeNew(type, state, dep));
      }, else if (inputLine.length() > 5 && inputLine.substring(0, 5).equals("Echo ")){
         System.out.println("Server: Echo " + inputLine.substring(5));
      }, else {
         System.out.println("Server: Unsupported message:" + inputLine );
      }
   }
```

Figure 4.5: Java code for transactors(continued)

send its state to a transactor "`uan://200.200.133.133:3030/agent`" as shown in Figure 4.6.

4.3 Issues and Trade-offs

In a practical implementation of the actor model, there are two conflicting goals: safety and efficiency. In the event that the implementation prioritizes safety, it is necessary

```
output("Getstate")
input(foo)
output("SendTo uan://200.200.133.133:3030/agent " + foo)
output("Ready")
```

Figure 4.6: A communication between a transactor behavior and the transactor system.

to ensure that the algorithm avoids allowing the program state to carry between message handlers. For example, this implementation actually starts a new instance of the program each time a message is received. On the other hand, the most efficient way to handle state is to keep state between program runs but roll back any new state information that has inadvertently been embedded into the handler. Of course, all such data can be kept between runs in the event that a `setstate` operation has been performed. Such an implementation avoids the high cost of initializing the message handler from scratch each time it is used, but it is impossible to prevent the user from maintaining the data from the message handler in between messages unless the transactor language is implemented as an independent language, rather than as a library or API. This is because there are numerous ways to hide such data in the form of file input/output or variables. Rebooting the handler between messages prevents some such techniques, but is very inefficient.

5. Related Work

5.1 Model Checking Transactional Memories

In the Guerraoui, Henzinger, Jobstmann, and Singh paper [7], the mechanisms needed to perform model checking on transactional memories are discussed. This model checking is difficult because the number and length of the transactions is unbounded, as is the size of the memory. Of importance is proving the safety of the transactional memories, which requires that there is consistency between transactions upon aborts and that the transactions themselves behave such that they can be treated as though they had been executed serially. Similarly, it is necessary to prove that liveness properties hold.

Firstly the paper proves that if a model violates these safety and liveness properties, then it is possible to violate these properties using distributed systems containing only two threads and two variables. This model can be tested using model checking to verify that the execution trace of the software transactional memory algorithm can be matched with reference algorithm which is known to be safe (i.e. because it only performs one transaction at a time and thus never aborts). This model is further altered to be a finite state machine.

5.2 Verifying Invariants of Distributed Systems

Graf and Sadi discuss the use of theorem proving in order to show that invariants of a distributed system hold [6]. They propose that rather than attempting to prove an invariant represented by predicate P holds directly, some other property represented by predicate P' should be computed. P' must be stronger than P but weaker than the *init* invariant. The *init* invariant holds for all reachable states. Thus, this algorithm seeks a predicate which is between P' and $init$. This is equivalent to saying that all states which are reachable must also follow the invariant.

This form of theorem proving is similar to model-checking, and can be automated just as is done with model-checking. There is a convergence between the *init* invariant and P wherein states are removed from *init* and added to P' (states are removed and added by changing the predicate definitions) until the two sets of states converge.

5.3 Atomos: A Transactional Programming Model

Atomos [4] is a programming language for use in programs with multiple threads which share resources. A traditional approach to making use of shared resources is to utilize some form of locking mechanism to ensure that particular elements of data are not accessed from multiple threads simultaneously. However, locking mechanisms are more prone to error and require extra work on the part of the programmer. An alternative approach is to make operations which access shared resources atomic. Such atomic operations are called transactions. Whereas transactors go from checkpoint to checkpoint, the Atomos uses 'checkpoints' only at the end of an atomic operation, as Atomos programs can only experience failures if mutual exclusion is broken.

The Atomos language replaces the use of code blocks which are locked based on particular resources with the ability for the programmer to produce a block of code which is atomic. This still ensures that there will not be an interleaving of code from two threads which are making use of the same resources, but avoids the requirement for the programmer to explicitly indicate which resources are being protected. It is worth noting that multiple atomic blocks can be interleaved by the language implementation so long as the atomic blocks do not share resources. However, the intermediate results of the transaction will be hidden until the transaction is committed. Thus, transactions are not necessarily executed independently, though the intermediate results are not visible outside of the atomic block itself.

In contrast to transactors which will (in an actual implementation) make a backup upon becoming immutable (`stable`), Atomos makes a backup upon starting an atomic operation which may include changes in state. Thus, Atomos is intended to handle failures caused by state changes happening at the same time. However, the transactions in Atomos which go from the start to the end of an atomic operation, which may or may not fail, are analogous to transactors, which go from checkpoint to checkpoint.

5.4 Stabilizers: A Checkpointing Abstraction for Concurrent Functional Programs

Stabilizers [10] is a linguistic abstraction which allows individual threads to be monitored and restored to globally consistent checkpoints after experiencing transient failures. The authors implement stabilizers with Concurrent ML and demonstrate that the overhead costs associated with stabilizers are about five percent, excepting in cases where threads are given a time quantum of less than 5ms. Stabilizers encapsulate three operations. First, code is monitored for communication and thread creation. Local checkpoints are performed when such code is evaluated. `Stable sections` are monitored sections of code which will be reverted as a single unit. Thus, checkpoints are directly associated with specific sets of performed actions. The second operation rolls back the state to a previous, safe global checkpoint. This checkpoint is determined dynamically. The third operation is used to delete checkpoints. The deleted checkpoints are thus unreachable, raising exceptions if a thread attempts to rollback to a state earlier than such a checkpoint.

6. Discussion and Conclusions

This thesis discussed three methods to progressively test and proof the safety of the transactor model. Firstly, we wrote an executable operational semantics using the Maude language. This allowed us to view step by step execution traces of programs written using the transactor model. The model discussed in this thesis incorporates the changes made to fix this error. Secondly, we formally proved that the new model follows the safety invariant using Athena. Finally, we developed a coordination language using Salsa and Java. This acts as a practical demonstration of a language following the transactor model. By examining the transactor model through the use of these tools, we were able to correct an error in the model and have greater confidence that the new model is safe.

6.1 Future Work

Future work possibilities include directly associating the Maude implementation of the model with the proof. This would mean using one proof document for the dual purposes of being an executable model of transactors and a set of assumptions for use in a proof. This has the clear advantage that the proof will be a proof that Maude program is safe rather than a proof that the Athena definition of the model is safe. Doing this would require the production of a separate modeling language to coordinate the two languages together. Such a language could make use of temporal logic in order to represent an execution trace between consecutive transactor configurations. Potentially, it could also use knowledge logic in order to represent worldviews, thus making it more practical to discuss how worldview information propagates using modal logic.

LITERATURE CITED

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, J. Meseguer, and K. Sen. Pmaude: Rewrite-based specification language for probabilistic object systems. *Electronic Notes in Theoretical Computer Science*, 153(2):213–239, May 2006.

[3] K. Arkoudas. *Denotational proof languages*. PhD thesis, Massachusetts Institute of Technology, 2000. Supervisor-Olin Shivers.

[4] B. D. Carlstrom et al. The ATOMOS transactional programming language. In *PLDI*, Ottawa, Canada, June 2006.

[5] J. Field and C. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In *ACM Conference on Principles of Programming Languages (POPL 2005)*, pages 195–208, Long Beach, CA, January 2005.

[6] S. Graf and H. Saïdi. Verifying invariants using theorem proving. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 196–207, London, UK, 1996. Springer-Verlag.

[7] R. Guerraoui, T. Henzinger, and V. Singh. Model Checking Transactional Memories. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[8] P. L. M. Clavel, S. Eker and J. Meseguer. Principles of maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[9] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology*

Track Proceedings, 36(12):20–34, Dec. 2001.
http://www.cs.rpi.edu/˜cvarela/oopsla2001.pdf.

[10] L. Ziarek, P. Schatz, and S. Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. *SIGPLAN Not.*, 41(9):136–147, 2006.

APPENDIX A
Symbol Table

$h_1 \rightarrow_{\Diamond} h_2$	h_1 stabilizes to h_2
$\Diamond(h)$	h is stable
$\surd(h)$	h has checkpointed
$t \leftarrow \rho$	t directly depends on ρ
$t \lhd_{\delta} t'$	t = t' or according to δ, t transitively depends on t'
$h_1 \looparrowright h_2$	h_1 rolls back to h_2
$h_1 \rightarrow_{\surd} h_2$	h_1 checkpoints to h_2
$h_1 \leadsto h_2$	h_1 is succeeded by h_2
$h_1 \leadsto_{*} h_2$	h_1 occurs at or before h_2
$\Diamond(t, \gamma, \delta)$	t is independent according to the worldview $(\gamma, \delta, \_)$
$h_1 \dashv h_2$	h_1 is validated by h_2
$h_1 \rtimes h_2$	h_1 is invalidated by h_2
$(\gamma_1, \delta_1, \rho_1) \oplus (\gamma_2, \delta_2, \_)$	The union of $(\gamma_1, \delta_1, \rho_1)$ and $(\gamma_2, \delta_2, \_)$