IMPLEMENTATION OF THE TRANSACTOR MODEL: FAULT TOLERANT DISTRIBUTED COMPUTING USING ASYNCHRONOUS LOCAL CHECKPOINTING

By

Phillip Kuang

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

Examining Committee:

_____

Carlos A. Varela, Thesis Adviser

_____

Charles V. Stewart, Member

_____

Ana Milanova, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2014
(For Graduation August 2014)

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to express my gratitude to Professor Carlos A. Varela for his guidance and support for my research. I would also like to thank John Field for his insight and advice that helped me gain a different perspective on my work.

ABSTRACT

The Transactor model, an extension to the Actor Model, is a well-formulated way to model distributed concurrent systems while maintaining a global distributed state. This model provides semantics to track dependencies among loosely-coupled distributed components that ensures fault tolerance through a two-phase commit protocol and issues rollbacks in the presence of failures or state inconsistency. We seek to introduce an implementation of this model through a transactor language as an extension of an existing actor language and highlight the capabilities of this programming model.

We developed our transactor language using SALSA, an actor language developed as a dialect of Java. This transactor language will in turn be a dialect of SALSA, which similarly will compile into Java code. We first develop our transactor language as a basic SALSA/Java library in conjunction with the SALSA Java library. This library implements the fundamental semantics of the transactor model following the defined transition rules. We then provide example programs written using this library such as the reference cell, banking transfer, and the house purchase transaction. Furthermore, we seek to expand our language by introducing a state storage property known as the *Universal Storage Location* as an extension of the *Universal Actor Name* and *Universal Actor Locator* in SALSA that levies a Storage Service to maintain checkpointed transactor states. We also introduce the *Consistent Transaction Protocol* and *Ping Director* that improves upon the *Universal Checkpointing Protocol* to aid transactor programs in reaching globally consistent states and perform reliable transactions.

1. Introduction

1.1 The Transactor Model

1.1.1 Overview

The transactor model introduced by Field and Varela is defined to be *"a fault tolerant programming model for composing loosely-coupled distributed components running in an unreliable environment such as the Internet into systems that reliably maintain globally consistent distributed state"*[1]. Therefore, transactors allow for guarantees about consistency in a distributed system by introducing semantics on top of the actor model that allows it to track dependency information and establish a two phase commit protocol. This allows transactors to recognize reliance on other transactors and how they directly influence its own current state. As an extension of the actor model, transactors inherit the core semantics of encapsulating state and a thread of control to manipulate its state as well as communication through asynchronous messaging. In addition to these, transactors introduce new semantics to explicitly model node failures, network failures, persistent storage, and state immutability.

Similar to an actor, when a transactor receives a message, it may create new transactors, send messages or modify its own state. Unlike an actor, it also has the option to stabilize, checkpoint, and rollback. Stabilization is considered the first step of a two-phase commit protocol and makes a transactor immutable until a checkpoint or rollback occurs. The second step of the two-phase commit is a checkpoint that, if successful, makes a transactor persistent and guarantees consistency among peer transactors. That is, the current transactors state does not have a dependence on any other volatile transactor states and will be able to survive local temporary node failures. Lastly rollback brings a transactor back to a previously checkpointed state.

1.1.2 Tau Calculus

The transactor model is formalized under the tau calculus, an extended, untyped, call by value lambda calculus. The terms of the tau calculus grammar is

shown in Figure 1.1 and syntactic sugar is provided in a set of defined forms shown in Figure 1.2.

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

Figure 1.1: Tau calculus grammar [1]

Figure 1.3 shows a collection of semantic domains that the tau calculus operational semantics will manipulate. A transactor history $h = \langle w, l_h \rangle, h \in \mathcal{Y}$ encodes its checkpointed history and is composed of a volatility value $w \in \mathcal{W}$ and an incarnation list. Here a volatility value encodes if a transactor is volatile ($w = \mathbf{V}(n),\ n \geq 0$) or stable ($w = \mathbf{S}(n),\ n \geq 0$). The value of n is known as the incarnation. The length of the incarnation list tracks how many times a transactor has checkpointed since its creation and its values reflect the incarnation at which each checkpoint occurred. Figure 1.4 defines four operations on histories and their resulting transitions. A history map $\gamma \in \mathcal{H}$ associates transactor names to their last known histories including its own. A dependency is a directed edge $t_1 \leftarrow t_2 \in \mathcal{D}$ that encodes that the state of t_1 depends on that of t_2 and a dependency graph $\delta \in \mathcal{G}$ is a set of these directed edges. This auxiliary structure collects all known transactor dependencies

including its own based off the names recorded in the history map. The root set $\rho \in \mathcal{R}$ of a transactor contains a set of names for which the current evaluation of a message depends on and is seen as potential dependencies if such a message results in a state mutation. These three components (γ, δ, ρ) comprise what is known as the worldview. Transactor messages $m \in \mathcal{M}$ involve a target recipient transactor, a value representing the payload of the message, and the message's worldview associated with the payload allowing dependence information to be transparently propagated among communicating transactors. Lastly, a transactor configuration $k \in \mathcal{K}$ consists of a network Ω or a multiset of messages and a name service Θ that maps transactor names to transactor references.

A transactor $\tau \in \mathcal{S} = \langle b, s_{\surd} \; ; \; e, s \; ; \; \gamma, \delta, \rho \rangle$ is modeled as a 7-tuple in the tau calculus. This 7-tuple is delimited by semicolons into three logical clusters as a syntactic convention. The first cluster represents the persistent component of a transactor which includes its behavior b, a fixed response to every incoming message, and its persistent state $s_{\surd}$, encoding the last checkpointed state modeling stable storage. The volatile component includes the current evaluation e, an intermediate evaluation of a transactor's behavior, and the volatile state s, that has yet to be checkpointed and is vulnerable to failures. The last component (γ, δ, ρ) represents the transactor's dependence worldview.

1.1.3 Transition Rules

Figures 1.5, 1.6, 1.7, and 1.8 depict the transition rules of the tau-calculus. Figure 1.5 encode the classical semantics of the actor model but differs in the treatment of state. Figure 1.6 augment the basic actor transitions with additional rules for managing distributed state. Figure 1.7 models spontaneous transient failures; failures beyond the control of transactors themselves such as a node failure. Finally, Figure 1.8 define the semantics of program induced failures through explicit rollback calls or implicit rollbacks as a result of being invalidated due to global inconsistency. Figure 1.9 illustrates the life of a transactor as it transitions to different possible states. Each transition rule is shown that progresses a transactor from being ephemeral or permanent, independent or dependent, and volatile or stable.

1.1.4 Worldview Union Algorithm

Figure 1.11 shows the worldview union algorithm used upon reception of messages to evaluate the dependency information included within the message. This algorithm unifies the current worldview of the recipient transactor with the worldview attached to the message by the sender in order to combine and update known information about transactors in the network. This is a key component of the transactor model, which allows dependency information to be shared appropriately through interactions. The first step of this process creates a versioning set to collect transactor name-history pairs from both history maps and a version dependency graph from these pairs based off both dependency graphs. Next, transactor history pairs are merged for duplicate names keeping the most recent history version and updating inferred history versions of related transactors. These inferences result from three possible progressions of history versions: stabilization, invalidation, and validation. Figure 1.10 defines different relations between history versions.

If a transactor history version is succeeded by stabilization then we can update the dependency graph where all edges containing the outdated history are replaced with the updated version. For a transactor history that is invalidated, we will rollback all transactor histories that were dependent on the outdated version. This is a logical inference we can make even if those transactors have not yet rolled back, we can reason that they will eventually rollback once they discover their state has been invalidated. Lastly, a validated history allows us to infer that the transitive closure of transactors it is dependent on has to have performed stabilization in accordance to the requirements of the two-phase commitment protocol. This also allows us to remove any transitive dependency edges the outdated history had since semantically all transitively backward edges are no longer true dependencies because all of these transactors are independent as defined in the [**dep1**] transition. Similarly we can remove any edges from any transactor that has a dependency on any of these independent transactors.

Finally we extract the updated worldview from the resulting version set and

version dependency graph and we take the message root set less any invalidated names as our potential dependencies for the message to be processed. Note that the worldview union algorithm presented here is a slightly optimized version of the original algorithm presented by [2] that reduces the amount of unnecessary information tracked by worldviews, providing a more efficient implementation.

1.2 SALSA Actor Language

SALSA [3, 4], which stands for Simple Actor Language System and Architecture, is a concurrent actor oriented programming language developed by Varela and Agha. SALSA allows users to easily facilitate dynamically reconfigurable open distributed systems and implements the core semantics of the actor model on top of Java. SALSA code is pre-processed into Java code, which compiles in conjunction with a SALSA actor library into Java byte code and can be run on a Java Virtual Machine. Figure 1.12 shows the class hierarchy of the SALSA actor library and Figure 1.13 shows the compilation process for SALSA. In SALSA, actor programs are written as "behaviors" which translates to Java classes, each of which extends the `salsa.language.UniversalActor` class to inherit actor semantics. We used SALSA as a base language for implementing the transactor model and follow a similar approach to developing our language by augmenting the SALSA library with a transactor library to add functionality to support the transactor model. We also leverage the integration of Java in SALSA to maintain the sequential nature of dependency semantics.

1.3 Transactor Language

We introduce our transactor language as an extension of the SALSA language similar to how the transactor model naturally extends the actor model. Our language allows users to follow an actor oriented programming paradigm for concurrent computation in order to build loosely coupled distributed systems. Such systems built with our transactor language will be able to maintain a globally consistent state as guaranteed by the transactor model. A need for central coordination is now obsolete since our language takes into consideration the high latencies of a wide

area network where node and link failures are common occurrences. Therefore, dependencies are evaluated lazily and users only need to reason about the possibility of lost messages since node and application-level failures are handled internally.[1] Through our language we also introduce the *Consistent Transaction Protocol* allowing users to resolve dependencies to reach global consistent states eagerly and the `Proxy`, which does not introduce any dependencies. These abstractions encode common programming patterns to make it easier to compose transactor programs.

1.4 Roadmap

Chapter 2 presents a few possible useful applications of a transactor language that highlight the importance of maintaining a consistent global state. Chapter 3 describes our implementation of a transactor language including how we handle persistent state storage, introduction of two new abstractions, and syntactic language support. Chapter 4 describes in detail three example programs written in our language highlighting the key semantics of the transactor model. Chapter 5 presents related work in other similar implementations of transactional languages and systems. Chapter 6 concludes with a discussion of contributions of our implementation and future work.

[1] The current version of our language only has support for application-level failures.

[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** 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** $x_1 = \mathbf{fst}(e_1)$ **in**
...
let $x_n = \mathbf{fst}(\mathbf{snd}(\ldots \mathbf{snd}(e_1) \ldots))$ **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* i*th 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* i*th name declared in the closest*
statically-enclosing **declstate** *scope, of length* n, $n \geq i > 0$

Figure 1.2: Tau calculus defined forms [1]

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

Figure 1.3: Semantic domains [1]

$$
\begin{array}{lcll}
\langle \mathbf{V}(n), l_h \rangle & \not\rightarrow & \langle \mathbf{V}(n+1), l_h \rangle & \text{(“rolls back to”)} \\
\langle \mathbf{S}(n), l_h \rangle & \not\rightarrow & \langle \mathbf{V}(n+1), l_h \rangle & \text{(“rolls back to”)} \\
\langle \mathbf{V}(n), l_h \rangle & \rightarrow_{\Diamond} & \langle \mathbf{S}(n), l_h \rangle & \text{(“stabilizes to”)} \\
\langle \mathbf{S}(n), l_h \rangle & \rightarrow_{\surd} & \langle \mathbf{V}(0), n :: l_h \rangle & \text{(“checkpoints to”)}
\end{array}
$$

Figure 1.4: History transitions [1]

[pure] *Evaluate pure redex.*

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

[new] *Create new transactor.*

$$\mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,\mathbf{trans}\ b'\ \mathbf{init}\ s'\ \mathbf{snart}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[new]}]{t} \mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,t'\,], s\,;\, \gamma', \delta \cup \delta', \rho \cup \{t'\}\rangle]\ [t' \mapsto \langle b', \mathsf{nil}\,;\, \mathbf{ready}, s'\,;\, \gamma', \delta' \cup \delta, \emptyset\rangle]$$

t'fresh
$\gamma' = \gamma[t' \mapsto \mathbf{H}_0]$
$\delta' = t' \leftarrow (\rho \cup \{t\})$

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

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

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

$$\frac{\neg((\gamma_m \downarrow \rho_m) \rtimes \gamma') \qquad \neg(\gamma(t) \not\rightarrow \gamma'(t))}{(\mu \uplus \{t \Leftarrow \langle v_m\,;\, \gamma_m, \delta_m, \rho_m\rangle\}) \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathbf{ready}, s\,;\, \gamma, \delta, \_\rangle] \xrightarrow[\text{[rcv1]}]{t} \mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, (b\ v_m), s\,;\, \gamma', \delta', \rho'\rangle]}$$

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

[get] *Retrieve state.*

$$\mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,\mathbf{getstate}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[get]}]{t} \mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,s\,], s\,;\, \gamma, \delta, \rho\rangle]$$

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

$$\frac{\neg\Diamond(\gamma(t))}{\mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,\mathbf{setstate}(s)\,], \_\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[set1]}]{t} \mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,\mathsf{true}\,], s\,;\, \gamma, \delta \cup (t \leftarrow \rho), \rho\rangle]}$$

[self] *Yields reference to own name.*

$$\mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,\mathbf{self}\,], s\,;\, \gamma, \delta, \rho\rangle] \xrightarrow[\text{[self]}]{t} \mu \parallel \theta[t \mapsto \langle b, \not\surd\,;\, \mathcal{R}[\,t\,], s\,;\, \gamma, \delta, \rho\rangle]$$

Figure 1.5: Transition rules encoding basic actor semantics [1]

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

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

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

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

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

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

[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**.

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

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

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

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

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

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

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

[lose] *Message loss.*

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

Figure 1.6: Transition rules encoding basic transactor semantics [1]

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

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

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

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

Figure 1.7: Transition rules modeling spontaneous failures [1]

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

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

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

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

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

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

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

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

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

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

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

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

Figure 1.8: Transition rules for programmatic rollback and consistency management [1]

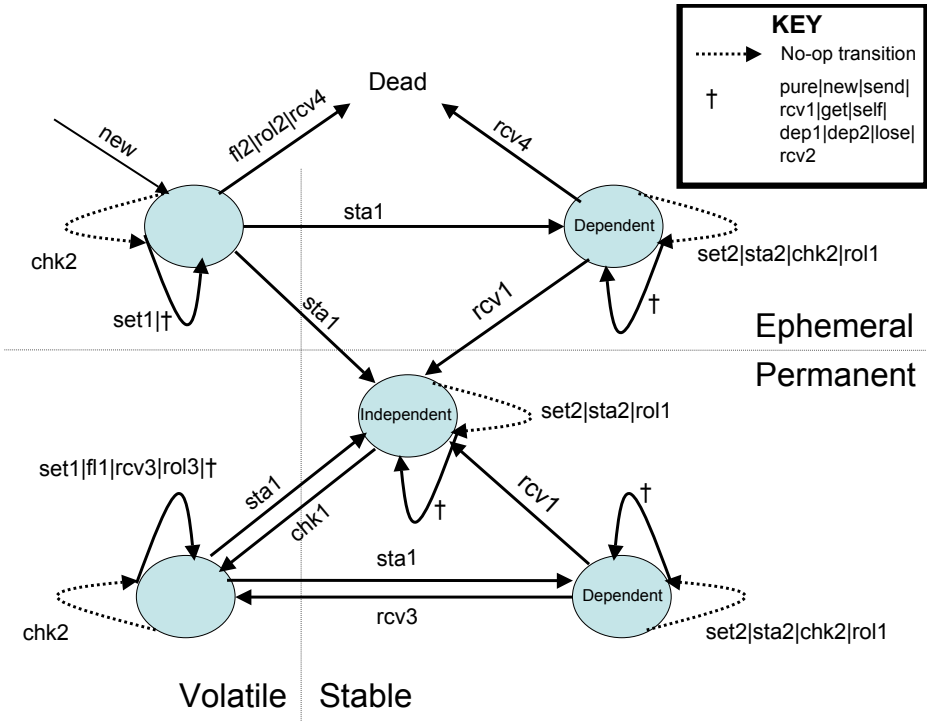

Figure 1.9: Transactor transition diagram [1]

$$
\begin{array}{lcll}
\rightsquigarrow & \triangleq & (\not\mapsto \cup \rightarrow_{\Diamond} \cup \rightarrow_{\surd}) & \text{(“is succeeded by”)} \\
\bowtie & \triangleq & [\rightarrow_{\Diamond}]^{\cdot} \not\mapsto \cdot \rightsquigarrow_{*} & \text{(“is invalidated by”)} \\
\dashv & \triangleq & [\rightarrow_{\Diamond}]^{\cdot} \rightarrow_{\surd} \cdot \rightsquigarrow_{*} & \text{(“is validated by”)}
\end{array}
$$

Figure 1.10: Relations on history versions [1]

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** $h' \rightarrow_\Diamond h''$.
$E = E - \{\langle t', h'\rangle \leftarrow \_\} - \{\_ \leftarrow \langle t', h'\rangle\}$
$V = V - \{\langle t, h_1\rangle\}$
$E = E - \{\langle t, h_1\rangle \leftarrow \_\} - \{\_ \leftarrow \langle t, h_1\rangle\}$

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 1.11: Dependence worldview algorithm $(\gamma',\ \delta',\ \rho') = (\gamma_t,\ \delta_t,\ \_) \oplus (\gamma_m,\ \delta_m,\ \rho_m)$ [1]

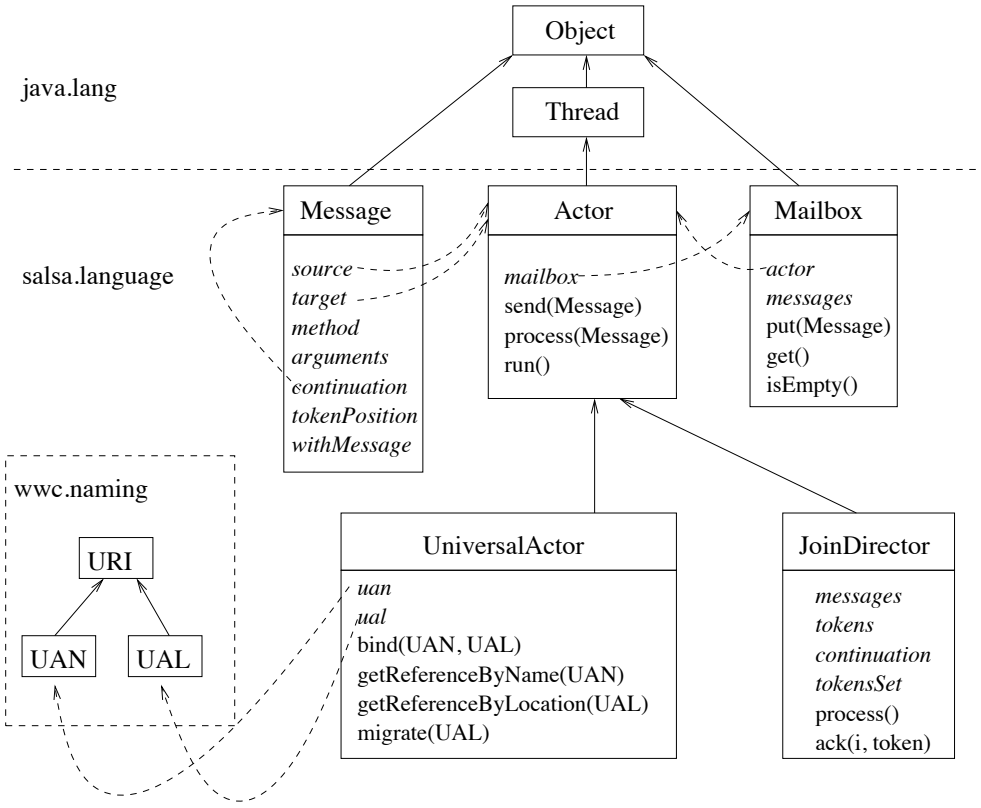

Figure 1.12: Class hierarchy diagram for the SALSA actor library [3]

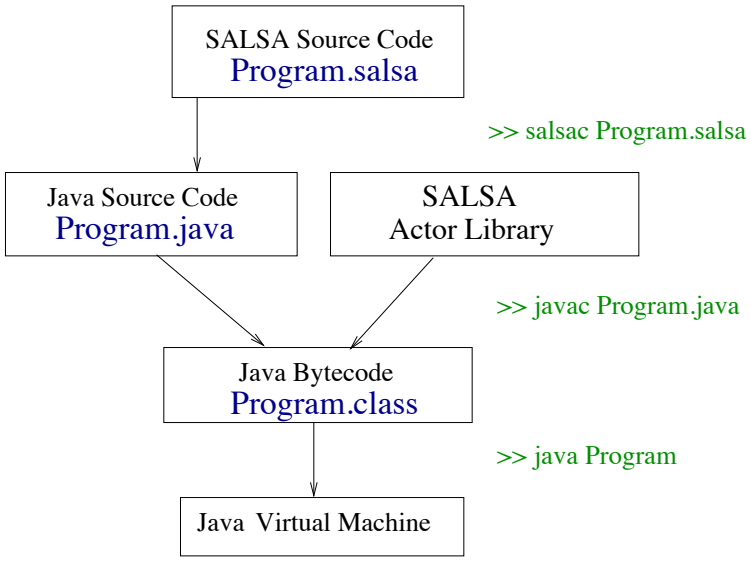

Figure 1.13: SALSA code compilation process [3]

2. Applications

In this chapter we briefly describe a few motivating application scenarios that would benefit from a transactor model language implementation. A transactor language benefits composing transactions where we have the ability to analyze meaningful dependencies between participating agents. The transactor model allows us to recognize meaningful dependencies through state influences and minimizes the overhead of rolling back transactions in the presence of failure. This is opposed to traditional transaction management schemes that delimit a subset of operations on a set of participating agents that define a transaction and processes them atomically to reach global consistent states. This results in a total commit or revert of these operations without taking into consideration which operations are directly invalidated from a discovered failure.

Take for example a house purchase service that involves collaborating with multiple other services. Possible transaction events would include a buying service communicating with an appraisal service to value the requested property, communicating with a bank or lending service to obtain a mortgage, requesting a title service to obtain documents about the property, and negotiating with a seller. Among these sub-transactions, there are multiple potential failure scenarios. One of which may be a denial of a mortgage due to bad credit. Semantically a house purchase that depends on the approval of the requested mortgage to fulfill the users requirement to commit on the purchase would invalidate the entire transaction causing a rollback. However, a transactor language lets us model this failure to only rollback sub-transactions directly dependent on the mortgage. Even though the "parent transaction" has failed, sub-transactions such as the title search and appraisal information should be able to commit and preserve their results if the purchase were to be reattempted. By leveraging the fine-grained dependencies of transactors, we are able to reuse resources without having to inefficiently redo these actions. This application is implemented and explained in greater detail in an example in chapter 4.

Another application would be a travel agency service. The task of this travel service would be to make travel reservations based off the preferences of the customer. A simple travel arrangement may specify an origin and destination, departure and return date ranges, and a maximum travel budget. To handle this request the travel service may create sub-transactions to perform searches on airline tickets, hotel accommodations, and car rental options. Under a transactor language, we can present varying levels of dependencies such as hotel accommodations dependent on the success of obtaining airline tickets. A hotel search needs to be rolled back if no successful flights are found and a different mode of transportation results in a later arrival date. A more dominant dependency causing total recall of the transaction could be insufficient funds to satisfy the requirements of the travel request.

The key benefit of a transactor language in these examples is establishing these dependencies implicitly by observing state mutations as a direct result of interaction with another agent. Influential communications are thereby distinguishable from casual ones. Any application with a collection of distributed components engaging to negotiate a mutually desirable outcome can be implemented nicely with an existing transactor language. Under this language, open distributed systems are easier to build with a dynamic topology so that non-transactional and transactional components can easily interact.

3. Language Implementation

3.1 Transactor Library

3.1.1 Transactor, Worldview, History

Our language is first developed as a transactor library on top of the SALSA library. Figure 3.1 shows the class hierarchy diagram of our transactor library. A transactor is encoded in the `transactor.language.Transactor` class that extends and inherits from the `salsa.language.UniversalActor` class. In additional to the semantics inherited from a SALSA actor we create Java classes that encapsulate the semantics of a transactor worldview and history. Figure 3.2 shows the java class

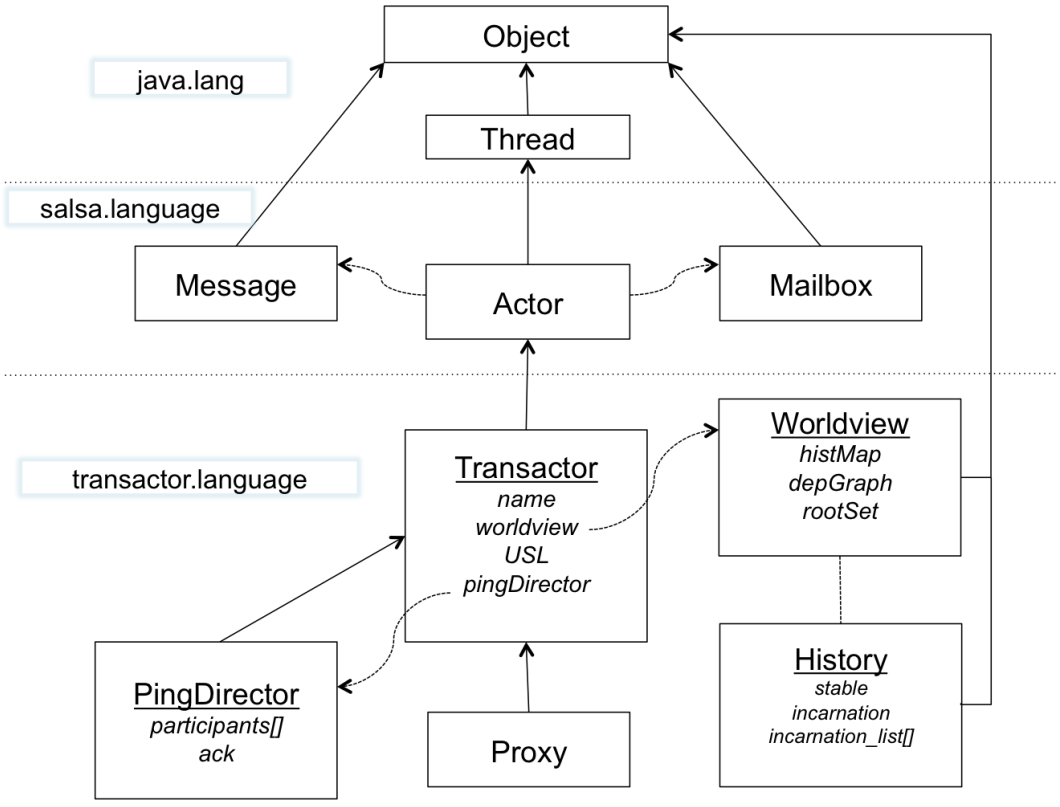

Figure 3.1: Class hierarchy diagram for the transactor library

implementation of a transactor history. This class includes all the fields that define a history: `stable` encodes the volatility value, `incarnation` encodes the current value of its incarnation, and `incarnation_list` records the list of checkpointed in-

```
public class History implements Serializable {
     private boolean stable;
     private AbstractList<Integer>incarnation_list;
     private int incarnation;

     public History();
     public History(History other);
     public History(boolean stable, int incarnation,
                    AbstractList<Integer>incarnation_list);
     public boolean isStable();
     public boolean isPersistent();

     public void stabilize();
     public void checkpoint();
     public void rollback();

     public boolean succeeds(History other);
     public boolean stablizesFrom(History other);
     public boolean invalidates(History other);
     public boolean validates(History other);
}
```

Figure 3.2: History class

carnation values. `isStable()` and `isPersistent()` are two primitives that indicate if the current history is stable and if it has previously checkpointed to stable storage making it persistent to failures. `stabilize()`, `checkpoint()`, and `rollback()` implement the history transitions described in Figure 1.4. `stabilizesFrom(History other)` indicates if a given history is an updated version of the current history that has been succeeded by stabilization. `succeeds(History other)`, `validates(History other)`, and `invalidates(History other)` implement the history relations defined in Figure 1.10.

Figure 3.3 shows the Java class implementation of the transactor worldview. This class naturally contains three fields: `histMap` represents the history map which maps name strings to `History` instances, `depGraph` represents the dependency graph implemented here as a mapping of name strings to a set of names the name key is dependent on, and `rootSet` represents the root set as a set of name strings.

The `union(Worldview other)` method implements the worldview union algo-

```
public class Worldview implements Serializable {
     private HashMap<String, History>histMap;
     private HashMap<String, HashSet<String> > depGraph;
     private HashSet<String> rootSet;

     public Worldview();
     public Worldview(HashMap<String, History> hmap,
                      HashMap<String, HashSet<String>> deps,
                      HashSet<String> roots);

     public Worldview union(Worldview other);
     public boolean invalidates(HashMap<String, History> other,
                                HashSet<String> domain);

     public boolean independent(String t);
}
```

Figure 3.3: Worldview class

rithm defined in Figure 1.11 which returns a worldview that contains updated dependency information reconciled by the algorithm. The `union(Worldview other)` method is invoked on reception of a new message and the `other` argument represents the worldview associated with the message in order to apply to the worldview of the recipient. This is important since the worldview union algorithm is not commutative. `invalidates(HashMap<String, History> other, HashSet<String> domain)` determines if the current worldview invalidates the set of names given in `domain` and its respective history mappings given in `other`. This method is also invoked on reception of a new message by the unionized worldview in order to evaluate if the recipient transactor invalidates the message dependencies. The first precondition of the transition rule [**rcv2**] in Figure 1.8 highlights use of this method. Lastly, `independent(String t)` determines if the given transactor name `t` is independent with respect to its dependency information known in the current worldview. This method is called by the transactor primitive `dependent` which implements the [**dep1**] and [**dep2**] transition rules defined in Figure 1.6. It is important to note that this primitive determines a weak independence which means all transactors it is dependent on are known to have stabilized but it itself may not be stable.

Each transactor instantiates a `Worldview` but dependency semantics are meant

to be transparent to the user. Similar to how SALSA implements a mailbox to handle message reception transparently, worldview operations are handled internally and the user cannot directly access such information except with supplied transactor operators. The class skeleton of the Transactor class is shown in Figure A.1 of Appendix A.

3.1.2 Message Sending/Reception

Message sending is inherited from SALSA as remote method invocations. We leverage the existing actor message handling implementation and mount dependency information along with messages to accommodate the transactor model. Just as in SALSA, message sending is asynchronous and message processing is sequential though the ordering of messages is not guaranteed. Message parameters are pass by value to ensure there is no shared memory with transactor states. We provide two methods to the `Transactor` class that implements transactor message handling:

```
public void sendMsg(String method, Object[] params,
                    Transactor recipient);
public void recvMsg(Message msg, Worldview msg_wv);
```

`sendMsg(String method, Object[] params, Transactor recipient)` implements a message send by taking as arguments a string, `method`, that represents the type of message being sent which should match an appropriate message handler on the recipient transactor and an array of message parameters, `params`, which should match the arguments of the message handler signature. With these arguments we instantiate a `salsa.language.Message` object the same way messages are created in SALSA but instead we will actually create a new message that wraps this requested message with the senders worldview and call the corresponding `recvMsg(Message msg, Worldview msg_wv)` message handler on the recipient. This method implements the [**snd**] transition rule of Figure 1.5.

`recvMsg(Message msg, Worldview msg_wv)` implements the [**rcv1**], [**rcv2**], [**rcv3**], and [**rcv4**] transition rules shown in Figures 1.5 and 1.8. On reception

of a message this method first evaluates the message dependencies by invoking the union algorithm on the recipients worldview along with the message worldview. Afterward we branch off to one of four receive transition rules by analyzing the reconciled dependency information. Invalidation of the recipient causes a rollback, annihilating it if it is ephemeral; invalidation of the message discards the message before it is processed; and if no invalidations are detected the message is placed in the transactor's mailbox to be processed normally by the SALSA library. In the cases where the recipient does not become invalidated, its worldview is updated to reflect the unionized worldview.

3.1.3 Stabilization, Checkpointing, Rollback

Stabilization, checkpointing and rollbacks are provided in the form of the following three transactor operator methods:

```
public void stabilize();
public void checkpoint();
public void rollback(boolean force, Worldview updatedWV);
```

`stabilize()` is a simple method that updates the transactor history volatility value to be stable if it is not already stable and implements the [**sta1**] and [**sta2**] transition rules presented in Figure 1.6. `checkpoint()` implements [**chk1**] and [**chk2**] and stores the current transactor state in stable storage if the transactor is independent and stable and clears its worldview. Implementation of stable storage is described in a later section. `rollback(boolean force, Worldview updatedWV)` implements the [**rol1**], [**rol2**], and [**rol3**] transition rules described in Figures 1.6 and 1.8. The arguments `force` and `updatedWV` are used when an implicit rollback is caused by being invalidated by a received message. By passing a `true` to the first argument we can force the transactor to rollback under this scenario even if it is stable. The second argument represents the updated worldview obtained by the union algorithm so the rolled back state reflects this information.

Implementation of a state rollback is inspired by SALSA actor migration.

Each transactor is inherently a SALSA actor, which encapsulates state in a thread of control so we handle rollbacks by halting the current thread and starting a new thread from a preserved checkpointed state and attaching the transactor name to it. However, before doing so, we create a special placeholder transactor, defined by the `transactor.language.Rollbackholder` class, to buffer incoming messages while the rollback operation is taking place. We register this placeholder state with the current transactor name under the SALSA naming service so messages can be routed correctly. We then read the checkpointed state from stable storage and tell the local system to start the transactor state as a new thread and reassign its name with the naming service. All buffered messages from the placeholder are then forwarded to the newly reloaded transactor's mailbox and normal processing resumes.

3.1.4 Transactor Creation

Transactor creation is done with the following transactor method:

```
public Transactor newTActor(Transactor new_T);
```

We use this method to extend the usual call to the `new` keyword in order to instantiate the newly created transactors worldview to reflect dependence on its parent. The `new_T` argument is an instantiated object of the transactor class to be created. The new transactor inherits the history map and dependency graph of the parent augmented with the new transactor's name and dependencies laid on the new transactor by the names in the parent's root set. Both parent and new child transactor will reflect the same history map and dependency graph but the parent will append the new transactor's name in its root set while the child starts with a fresh root set. This method returns the same reference to the new instantiated transactor with an updated worldview, as depicted in the [**new**] transition in Figure 1.5. The returned reference must then be type casted back to the constructed transactor class. The following code sample shows use of this method to create a new HelloWorld transactor:

```
HelloWorld hwObject = (HelloWorld) newTActor(new HelloWorld());
```

3.1.5 State Mutation/Retrieval

State mutation and retrieval is done with the following two transactor methods:

```
public boolean setTState(String field, Object newValue);
public Object getTState(String field);
```

[**get**], [**set1**], and [**set2**] described in Figures 1.5 and 1.6 are implemented by the above two methods. `setTState(String field, Object newValue)` takes as arguments a string that represents the `field` being modified and the `newValue` to mutate the state with. We use Java reflection to reference the appropriate field in its state and mutation is done by replacing the value with the new value. State fields are therefore inherently immutable so set states are actually creating new states similar to mutating references to a Java `String` and how the *Clojure* [5] dynamic programming language handles state changes. Similarly `getTState(String field)` uses Java reflection to reference and return the value of the requested `field`. Dependencies are created on the current root set if a set state occurs and the transactor name is appended to its root set if a get state occurs.

3.2 Persistent State Storage

3.2.1 Universal Storage Locator

In order to handle persistent state storage, we introduce the *Universal Storage Locator* (USL) to represent the location where checkpoints will be made. This location can be the local system, a remote server, or even the cloud, allowing the user to specify the optimal location to create persistent storage. The USL is inspired from the *Universal Actor Name* (UAN) and *Universal Actor Locator* (UAL) in SALSA and is a simple uniform resource identifier. Some examples of USLs are shown below. The first USL indicates local storage, the second indicates remote storage on a specified FTP server, and the last USL specifies storage on Amazon's Simple Storage Service (S3) cloud storage [6].

```
file://path/to/storage/directory/
ftp://username:password@domain.com:1234/path/to/storage/directory/
http://s3.amazonaws.com/bucket/
```

Transactors are instantiated with a USL and if none is specified, checkpoints are made locally in the current directory. Specifying a USL is similar to specifying UAN and UAL in SALSA:

```
HelloWorld helloWorld = new HelloWorld ()
              at (new UAN("uan://nameserver/id") ,
                  new UAL("rmsp://host1:4040/id"),
                  new USL("file://path/to/storage/directory/");[1]
```

When a transactor checkpoints it will reference its USL to serialize its state and store a `<transactor-name>.ser` file at the location given by its USL. A rollback will reference the same file at the USL location to retrieve and de-serialize its state. The implementation of a USL also allows the possibility of mobile transactors similar to how a SALSA actor's UAN and UAL allow it to perform migration. Separating a transactor's storage location makes it location independent, allowing it migrate as opposed to a locally checkpointing transactor. However further research still needs to be done on modeling mobile transactors whose state may be location dependent.

3.2.2 Storage Service

Here we introduce the *Transactor Storage Service*, a service class that handles performing serialization/de-serialization of a transactor's state and storing/retrieving it at the transactor's USL. We implement this service as an interface shown in Figure 3.4. This simple interface has two methods for storing and retrieving state. We chose to create a interface to give the user the ability to implement his or her own desired serialization technique and USL protocol. Doing so gives the user the flex-

[1]This example is written with language support syntax which would compile to use the `newTActor(Transactor new_T)` method.

ibility to define the optimal implementation that best caters to the given program specifications and performance requirements.

```
public interface TSTorageService {

    public void store(Object state, URI USL);

    public void Object get(URI USL);

}
```

Figure 3.4: Transactor storage service interface

For example a user might wish to use a FTP server to handle checkpoints and will create USLs with the `ftp://` scheme and implement the `store(Object state, URI USL)` and `get(URI USL)` methods to handle the FTP protocol with authentication. A high performance program can implement the use of cloud storage that has many benefits to program performance such as data redundancy and locality. Another high performance example is an implementation that utilizes memory storage instead of persistent storage to achieve fast checkpoint and rollback calls in a program that disregards the possibility of node failures.

We implement this service by extending the `salsa.language.ServiceFactory`, which manufactures references to standard location dependent services in SALSA. If no storage service implementation is specified then the system will default to a local file storage implementation. Internally, checkpoints and rollbacks invoke this service through the service factory with the following calls:

```
ServiceFactory.getTStorage().store(this, USL);
Transactor.State savedState =
    (Transactor.State) ServiceFactory.getTStorage().get(USL);
```

3.3 Proxy Transactor

In the transactor model, the semantics of message passing with dependencies were developed while taking into consideration the possibility of transactors whose task is to pass along messages they receive without affecting the dependencies of

those messages. Such a transactor is known as a *proxy transactor*. Similar to a network proxy, a proxy transactor routes messages to other transactors and in doing so must not introduce any new dependencies on that proxy. Proxy transactors can prove useful in order to provide privacy for a certain resource or perform message filtering.

We implement this abstraction by creating a `transactor.lanuage.Proxy` transactor class that extends the `transactor.language.Transactor` class. By doing so we inherit all the semantics of a traditional transactor, however we will override the message send and receive implementations to prevent inserting volatile dependencies. We do so by simply issuing an explicit call to `stabilize` prior to sending or processing a new message. By stabilizing before sending a message, we guarantee that the recipient transactor remains independent with respect the proxy. This affects situations where the proxy may perform a get state introducing its name to the message root set and the recipient subsequently performing a set state creating new dependencies on the names in the message root set. If the recipient wishes to perform a checkpoint in the future then its worldview would have knowledge of the proxy being stable and therefore not impede it from doing so.

We perform stabilization before processing a message upon reception to guarantee new dependencies are not introduced to the proxy. By being stable, any set state calls while processing a message become no-ops and therefore the proxy will not inherit any new dependencies from the message. Similarly, programmatic rollbacks are also no-ops due to being stable. However, a proxy is allowed to checkpoint and rollbacks due to node failures are possible. These two possibilities are also handled by stabilization: the proxy returns to a stabilized state after checkpointing to a volatile state; intermediate evaluation states of a stable transactor's behavior are logged to persistent storage according to the transactor model.[2]

The last consideration to make to ensure proxy transactors do not influence any global dependencies is the creation of child proxy transactors and creation of child transactors by the proxy transactor. We remove the ability of a proxy transactor to create child transactors since doing so will alter its worldview to reflect a volatile

[2]This feature has not yet been implemented in our language.

child and could carry on as a dependency by future message recipients. A proxy that attempts to create a child transactor becomes a no-op. Lastly, to eliminate any transitive dependencies that stem from a proxy, we restrict proxy creation only to transactors who meet two conditions: the transactor must be independent and stable and the names in the transactor's root set must also be stable. We reason that this is logical because any invalidation of the parent transactor or transactors whose state resulted in the creation of the proxy will also invalidate the proxy and possibly any recipients of the proxy messages. This would be inconsistent with the semantics of a proxy transactor.

3.4 Global Consistency

3.4.1 Consistent Transaction Protocol

To aid in composing transactor programs, we introduce the *Consistent Transaction Protocol* (CTP). This protocol draws inspiration from the *Universal Checkpointing Protocol* (UCP) presented in [1]. The UCP was developed to ensure the liveness property of the tau calculus under a set of preconditions. If these preconditions are met then global checkpoints are established through this protocol. However, a strict precondition of the UCP states that no failures can occur while the UCP is taking place and no transactors will rollback during the UCP. This assumes previous application dependent communication and a fault resistant system to guarantee these conditions are met. While it is proven that global checkpoints are possible in this type of situation, any failure would render the UCP useless. Such failures may halt program progress if the rest of the program is unaware of the failure without extra communication. Therefore we have introduced this new protocol to ensure global consistent states can be reached even in the presence of failures.

We abstract over a transaction defined as a set of participating transactors communicating with each other that results in a global checkpoint or rollback. Transactions are started by a *trigger message* sent by an outside agent whose recipient is defined to be the *coordinator*. On reception of the *trigger message* the message handler behavior defined in the *coordinator* starts the transaction by starting the

"conversation" among the participating transactors. During this "conversation", states may be altered and new dependencies might be created. Eventually the transaction ends when all "conversations" have been completed and ideally we wish to reach a globally consistent state by issuing a global checkpoint or a rollback in the case of failure, to promote atomicity and consistency. Durability of the transaction is ensured by checkpointing a successful transaction and is dependent on the storage implementation.

In order to reach a consistent state each participant must be made aware of the state of the transactors it has become dependent on and more importantly it must be made aware of any failures that may have invalidated itself. The UCP handles relaying this information though the use of ping messages whose main purpose is to carry dependency information and tell the recipient to attempt a checkpoint if it has enough information to be aware of its independence, otherwise it is a no-op and the cycle of ping messages continue until a checkpoint can be made. These ping messages can also be used to inform others of failure and cause rollbacks to invalidated transactors. The UCP only permits the use of ping messages to reach checkpoints while our protocol will also allow invalidation information to be carried along in ping messages.

In our proposed protocol we define 5 preconditions:

1) The transitive closure of all participants and their dependencies accrued during the transaction must be known ahead of time and each participant must be able to receive and issue ping messages.

2) There must be isolation of the participating transactors during the transaction; i.e., communication only within the set of participants and messages may not be received from an outside transactor that would introduce new dependencies.

3) Each participant starts from a state that is independent of any transactors other than those among the participants. We also assume the outside agent who sends the *trigger message* will not introduce any dependencies on itself or any other outside transactors.

4) Each participant must be stable at the end of the transaction unless it has rolled back at some point during the transaction.

5) The *coordinator* must be able to recognize a transaction has come to an end and indicates start of the consistency protocol.
Once a transaction completes, each participant will send ping messages to all other participants and attempt a checkpoint if it is independent. On reception of a ping message, the transactor will also attempt a checkpoint.

Since each transactor arrives at a stable state at the end of a transaction if it has not rolled back, a checkpoint succeeds if it is independent or has received enough ping messages to know it is independent. In the case of failure, a rolled back transactor is volatile at the end of a transaction so all checkpoint calls will be no-ops. On the other hand, ping messages sent out from the failed transactor will alert all those who were dependent on it and invalidate them, causing them to also rollback. Therefore a globally consistent state is reached at the end of the transaction through this protocol. This protocol also exemplifies eager evaluation of dependencies as opposed to the natural lazy evaluation of the transactor model.

3.4.2 Ping Director

In order to accommodate the CTP we introduce a new abstraction know as the *Ping Director* shown in Figure 3.5. The *Ping Director* is responsible for triggering the CTP by requesting all participants to ping each other. We also extend the transactor with a new operator and three additional message handlers native to all transactors shown below:

```
public void startTransaction(Transactor[] participants,
                             Transactor coordinator,
                             String msg,
                             Object[] msg_args);
public void transactionStart(String msg, Object[] msg_args,
                             PingDirector director);
public void pingreq(Transactor[] pingreqs);
public void ping()
```

```
behavior PingDirector extends Transactor {
     private Transactor[] participants;
     private boolean ack;

     public PingDirector();

     public void pingStart(Transactor[] participants,
                           Transactor coordinator, String msg,
                           Object[] msg_args);

     public void ping();
     public void endTransaction();
}
```

Figure 3.5: `PingDirector`

The last two methods, `pingreq(Transactor[] pingreqs)` and `ping()`, give transactors the ability to send and receive ping messages. `pingreq(Transactor[] pingreqs)` takes an array of transactors and issues ping messages to each one, and `ping()` handles the reception of ping messages to attempt a `checkpoint`. The `pingStart(...)` method is a new transactor operator that is invoked by the outside agent who triggers the transaction. This method takes as arguments the array of participants, the coordinator transactor to receive the trigger message, the trigger message, and the trigger message arguments. Internally this method will create a instance of the `PingDirector` that will handle the current transaction and send a `pingStart` message to the `PingDirector` instance with the array of participants, coordinator transactor reference, trigger message and its arguments. The `PingDirector` will then record in its state the array of participants and then send a `transactionStart` message with the trigger message and its arguments and a reference to itself to the coordinator. The `PingDirector` also sends itself a `ping()` message to be described later. The `transactionStart(String msg, Object[] msg_args, PingDirector director)` method records the `PingDirector` instance reference in its state and sends the trigger message to itself to be processed and start the transaction.

Since messages from the `PingDirector` affect the state of the coordinator,

we need the `PingDirector` to be independent so it will not affect the dependencies of the transaction. We do so by having the system create an instance of the `PingDirector` through a new service known as the *Transaction Director*. We access the *Transaction Director* through the `salsa.language.ServiceFactory` and request a new instance of the `PingDirector` instead of explicitly creating one in the `startTransaction(...)` method.

When the coordinator recognizes the completion of the transaction it will send a `endTransaction()` message to the `PingDirector` causing the `PingDirector` to stabilize. This stabilization alerts the `PingDirector` that the transaction is complete. The `PingDirector` recognizes this alert through the `ping()` it sent itself at the start of the transaction. On reception of a `ping()` message the `PingDirector` attempts a set state on the `ack` field. The value set is not important but the success of this set state is an important indicator of if the transaction has completed. Before the transaction has completed, the `PingDirector` will be volatile and therefore set states are legal and will return `true`; however, when the `PingDirector` receives an `endTransaction()` message causing it to stabilize, the set state will no longer be legal and will return `false`. We use this as the indicator and have the `PingDirector` resend the `ping()` message to itself until it recognizes it has stabilized in a polling manner. At that point the `PingDirector` will send `pingreq` messages to every participant and pass to each one the array of participants for them to ping. We use stabilization as an indicator instead of parsing the actual value of `ack` because updating the value of `ack` by the coordinator will apply new dependencies that will be carry along the `pingreq` messages. This makes it possible for some `pingreq` messages to be dropped due to being invalidated by the newly discovered dependency information and therefore some participants omit themselves from communicating `ping` messages.

The complex overhead of preparing a transaction and completing a transaction is shown in Figure 3.6. Fortunately, this protocol is simplified by our abstraction and the user only needs to worry about indicating the start and end of a transaction. An example of this abstraction being utilized is shown in the examples in chapter 4. We also note that a proxy transactor, described in the previous section, cannot

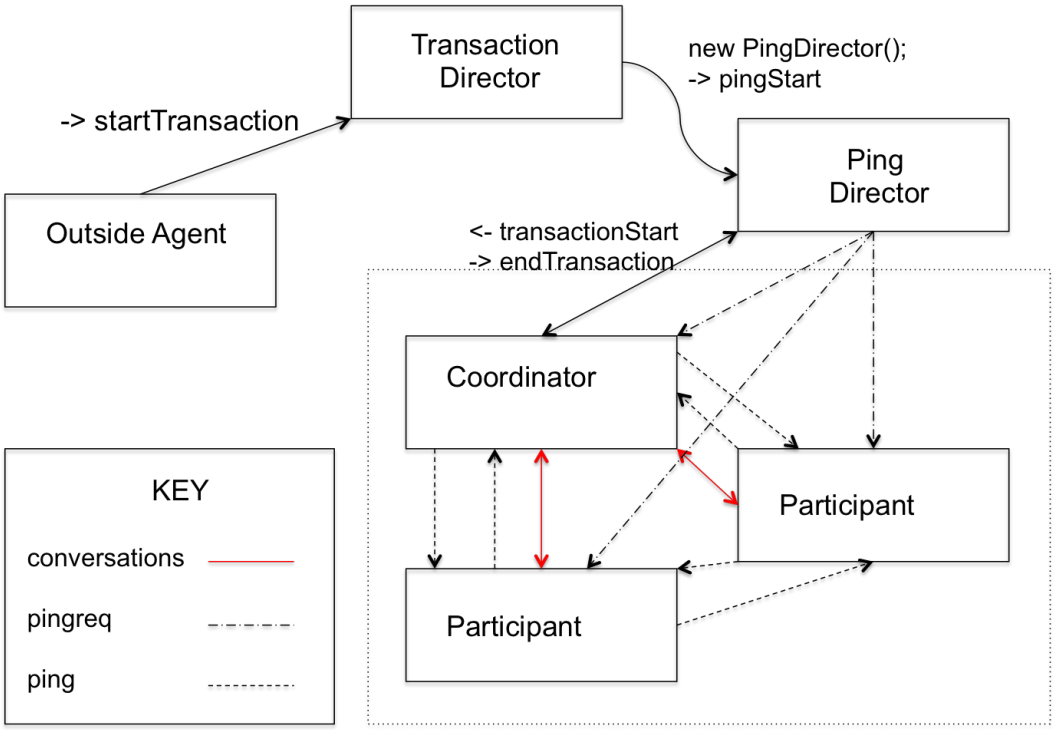

Figure 3.6: Consistent transaction protocol using the `PingDirector`

be designated as a coordinator since it cannot alter its state to record a reference to the `PingDirector`. Semantically, proxies have no effect on the global dependency so therefore they do not participate in the CTP, being that they will always be consistent with the global state.

We note that currently the CTP assumes that the coordinator and ping director are resistant to failure. However if one of these agents failed then the CTP would not be able to be triggered. To accommodate for this possibility we propose extending the protocol to provide fault tolerance in the form of redundancy. This can be done by assigning multiple coordinators where each would be able to recognize a transaction completion and trigger the protocol if one fails. The same can be done with creating multiple ping directors for a transaction and supplying a reference to each one to the coordinator. The exact details of implementing a fault tolerant CTP are left as future work.

3.5 Language Syntax

Similar to SALSA, transactor programs are written as actor behaviors that are compiled into Java classes that extend the `transactor.language.Transactor` class. Through this inheritance chain, behaviors have access to an augmented set of operators that include both the actor and transactor library. These operators can only be called by the transactor itself and are not explicit message handlers; therefore other transactors cannot directly issue a `stabilize`, `checkpoint`, or `rollback` on another transactor. These operators must be placed in message handlers inside the transactor's behavior. These operations are also sequential in nature, unlike message sends, which are concurrent. We define here our proposed syntax changes for our new transactor language that extends the SALSA/Java syntax.

The following statements are added/modified in the SALSA grammar along with the compiled transactor library code. A more illustrative example of our proposed syntax is shown in the house purchase example in chapter four.

```
stabilize;       ::=     this.stabilize();
checkpoint;      ::=     this.checkpoint(); return;
rollback;        ::=     this.rollback(false, null); return;
behavior <Identifier> ::= behavior <Identifier> extends Transactor
behavior proxy <Identifier> ::= behavior <Identifier> extends Proxy
dependent;       ::=     this.dependent();
tself;             ::=      this.self();

startTransaction(<ArgumentList>);
::= this.startTransaction(<ArgumentList>);

endTransaction; ::= this.sendMsg("endTransaction", new Object[0],
((PingDirector)this.getTState("pingDirector")));

new <Transactor-Behavior>
::= (<Transactor-Behavior>)this.newTActor(new <Transactor-Behavior>);
```

```
<State-Identifier>:=<Expression>;
::= this.setTState("<State-Identifier>", <Expression>);

~!<State-Identifier>;
::= ((<State-Identifier-Class>)this.getTState("<State-Identifier>"));
```

4. Examples

4.1 Reference Cell

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

Figure 4.1: Simple reference cell [1]

The example shown in Figures 4.1, 4.2, and 4.3 shows three different implementations of a reference cell written in the tau calculus. These three different versions highlight the semantics of the transactor model by providing progressively more refined notions of consistent state under different failure and interaction assumptions. The first version (*cell*) defines an unreliable reference cell that is volatile and never checkpoints so it does not tolerate failures. Any transactor that depends on the cell's value will not be able to checkpoint and a failure will cause it to be annihilated. This version of the reference cell is simply the actor implementation. The second version ($pcell_1$) presents a persistent reference cell that performs a checkpoint on initialization so it will be able to survive failures. This cell will also stabilize and checkpoint after each set to ensure the new value is persistent but this makes the assumption that the sender is stable and independent, which may not be the case. It also stabilizes before responding to gets to ensure no dependencies are incurred in the cell's customer and checkpoints to preserve the invariant of being checkpointed and volatile. The last version ($pcell_2$) ensures that the cell is reliable in that it will only recognize set messages if the sender is independent and stable so no outstanding

```
let pcell_1 = 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 4.2: Persistent reference cell [1]

dependencies are created. A volatile sender will cause the cell to rollback and discard any changes to its state. Figures 4.4, 4.5, and 4.6 shows implementations of the there different reference cells written in compiled code in SALSA of our transactor language.

4.2 Bank Transfer

The example shown in Figures 4.7, 4.8, and 4.9 is the classic money transfer transaction implemented with three transactors in the tau calculus. There is a *teller* transactor that acts as the ATM machine, which performs the desired money transfer between two given accounts. The *bankaccount* transactor represents a simple bank account with a given balance and message handler to perform adjustments to its balance. Lastly the *pinger* transactor is a auxiliary transactor that sends ping messages to each participant to broadcast their current state to each other in order for dependencies to be resolved and checkpoints to be reached or inconsistencies to be discovered. Each transactor is assumed to be initially persistent. The example

```
let pcell3 = trans
  declstate ⟨contents⟩ in
    msgcase
       initialize⟨⟩ ⇒
         stabilize;
         checkpoint
    |  set⟨val⟩ ⇒
         contents := val
         if dependent? then
           rollback
         else
           stabilize;
           checkpoint
         fi
    |  get⟨customer⟩ ⇒
         stabilize;
         send data⟨!contents⟩
              to customer
         checkpoint
    esac
  etats
init
  ⟨0⟩
snart
```

Figure 4.3: Persistent reliable reference cell [1]

presented here is an updated version of the one presented in [1] with the addition of the pinger transactor to correct a issue with the old example that prevented transactors from resolving an inconsistent state. This updated example is also more robust allowing the *teller* to work with any two bank accounts.

The transaction starts with an outside transactor sending the *teller* a transfer message with the amount to transfer, *delta*, and accounts, *inaccount* and *outaccount*, to credit and debit respectively. The last argument required is the *pinger* transactor to be used in this transaction. We assume that the outside transactor creates an appropriate *pinger* transactor initialized with a reference to the *teller*. The outside transactor is assumed to be stable and both it and the created *pinger* transactor are assumed to be independent so extra dependencies will not be introduced to the *teller* and bank accounts. On reception of the transfer message, the *teller* will send a

```
behavior cell extends Transactor {
     private int contents = 0;

     public cell(int contents) {
          super(self);
          this.setTState("contents", contents);
     }

     public void set(int val) {
          this.setTState("contents", val);
     }

     public void get(Transactor customer) {
          Object[] args = {((int)this.getTState("contents"))};
          this.sendMsg("data", args, customer);
     }
}
```

Figure 4.4: Simple reference cell implementation

`pingStart` to the *pinger* with references to both accounts and adjustment messages to both accounts with the amount to adjust. The *pinger* sets its state with both accounts, stabilizes, and sends itself a `ping` telling itself to `ping` the *teller*. It is important to note here that the `pingStart` message is the first message sent by the *teller* and therefore the *teller* passes this message like a proxy without introducing new dependencies to the *pinger*. The subsequent `adj` messages sent by the *teller* involve a reference to `self` which adds the *teller* name to the message root set. When each account receives the `adj` message, both perform a set state updating their balances and add a dependency on the *teller*. Each account determines if the updated balance is a valid amount. If so, it stabilizes and sends a `done` message to the *teller*, otherwise it sends a `done` message and performs a rollback. Each time the *teller* receives a `done` message it updates the number of acknowledgements it has received and stabilizes once it receives both. At this point the *teller* does not know if each adjustment was successful or not, assuming the `done` message is not interpreted, and it is also now dependent on both accounts since it has also performed a set state.

The *pinger* transactor starts the commit protocol after the *teller* gets an ac-

```
behavior pcell extends Transactor {
     private int contents = 0;

     public pcell(int contents) {
          super(self);
          this.setTState("contents", contents);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void set(int val) {
          this.setTState("contents", val);
          this.stabilize();
          this.checkpoint(); return;
     }

     public void get(Transactor customer) {
          this.stabilize();
          Object[]args = {((int)this.getTState("contents"))};
          this.sendMsg("data", args, customer);
          this.checkpoint(); return;
     }
}
```

Figure 4.5: Persistent reference cell implementation

knowledgement from both accounts. We can observe that the `ping` message handler of the *teller* performs a set state on an auxiliary member, *acked*, and replies with a `ping` if it is successful and `pingreq` if it fails. This is used to indicate if the *teller* is currently stable, which only occurs after obtaining both acknowledgements. Until the *teller* is stable, it will exchange `ping` messages with the *pinger* and once it is, it sends a `pingreq`. Another important point to be made here is even though the *teller* is dependent on the *pinger*, it has no real effect since the *pinger* is independent. On reception of a `pingreq` the *pinger* sends `pingreq` messages to both accounts and checkpoints to return it to a volatile state so it can process new transactions. These `pingreq` messages cause both accounts to reply with a `pingreq` to the *teller*, which informs the *teller* the status of both accounts. Therefore if either

```
behavior prcell extends Transactor {
     private int contents = 0;

     public prcell(int contents) {
          super(self);
          this.setTState("contents", contents);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void set(int val) {
          this.setTState("contents", val);
          if (this.dependent()) {
               this.rollback(false, null); return;
          }
          else {
               this.stabilize();
               this.checkpoint(); return;
          }
     }

     public void get(Transactor customer) {
          this.stabilize();
          Object[] args = {((int)this.getTState("contents"))};
          this.sendMsg("data", args, customer);
          this.checkpoint(); return;
     }
}
```

Figure 4.6: Persistent reliable reference cell implementation

account has rolled back it will cause the *teller* to also rollback, and according to [**rcv3**] the message remains so the *teller* will `ping` the other account causing that to also rollback. In this scenario the checkpoint calls are no-ops and `ping` messages received by a rolled back account are dropped since they are invalidated. In the absence of failure each transactor will be able to successfully checkpoint and a globally consistent state is reached once the transaction completes. Figures 4.10, 4.11, and 4.12 show the SALSA transactor implementation of the *teller*, *bankaccount*,

```
let teller = trans
  declstate ⟨acks, acked⟩ in
    msgcase
       transfer⟨delta, inacct, outacct, pinger⟩ ⇒
         send pingStart⟨inacct, outacct⟩ to pinger;
         send adj⟨delta, self⟩ to inacct;
         send adj⟨−delta, self⟩ to outacct;
    |  done⟨msg⟩ ⇒
         send println⟨msg⟩ to stdout;
         acks := !acks + 1;
         if !acks = 2 then
           stabilize;
         fi
    |  pingreq⟨requester⟩ ⇒ // may cause rollback
         send ping⟨⟩ to requester;
         checkpoint
    |  ping⟨pinger⟩ ⇒
         if (acked := nil) = true then
           send ping⟨⟩ to pinger;
         else
           send pingreq⟨⟩ to pinger;
         fi
    esac
  etats
init
  ⟨0, nil⟩
snart
```

Figure 4.7: Electronic money transfer example: Teller [1]

and *pinger* respectively. The execution trace of our implementation is shown in Appendix B Figures B.1, B.2, and B.3.

The implementation of the bank transfer along with the *pinger* transactor led to the development of our *Consistent Transaction Protocol* and *Ping Director*. Figures 4.13 and 4.14 present the same example implemented with the CTP and `PingDirector`, allowing us to simplify the original implementation and remove the *pinger* transactor. We assume a `startTransaction` message is sent by an outside agent to start this transaction.

```
let bankaccount = trans
  declstate ⟨bal⟩ in
    msgcase
       adj⟨delta,atm⟩ ⇒
        bal := !bal + delta;
        if !bal < 0 then
          send done⟨"Not enough funds!"⟩ to atm;
          rollback
        else
          stabilize;
          send done⟨"Balance update successful"⟩ to atm;
        fi
    |  pingreq⟨requester1,requester2⟩ ⇒
          send pingreq⟨requester2⟩ to requester1;
    |  ping⟨⟩ ⇒ // may cause rollback
          checkpoint
    esac
  etats
init
  ⟨0⟩
snart
```

Figure 4.8: Electronic money transfer example: Bankaccount [1]

4.3 House Purchase

This example simulates the subset of operations that might be performed by a collection of web services involved in the negotiation of a house purchase. Traditionally, a house purchase is a complex task that involves multiple parties and back and forth communication. Some steps required include appraising the desired house, searching for the title, applying for a mortgage, and making negotiations. We represent these operations using five services: the `buySrv` representing the buyer, the `sellSrv` representing the seller, the `apprSrv` representing the appraisal service, the `lendSrv` representing the mortgage lender, and the `srchSrv` representing the title search service. Our example defines the following steps taken to complete a house purchase:

1) The buyer chooses a candidate house and initiates the `buySrv` to manage the house purchase process

2) The `buySrv` contacts the appraisal service, `apprSrv`, in order to obtain the market

```
let pinger = trans
  declstate ⟨acct1, acct2, atm⟩ in
    msgcase
       init⟨⟩ ⇒
         stabilize;
         checkpoint;
    |  pingStart⟨inacct, outacct⟩ ⇒
         acct1 := inacct;
         acct2 := outacct;
         stabilize;
         send ping⟨⟩ to self;
    |  ping⟨⟩ ⇒
         send ping⟨self⟩ to !atm;
    |  pingreq⟨⟩ ⇒
         send pingreq⟨!atm, !acct2⟩ to !acct1;
         send pingreq⟨!atm, !acct1⟩ to !acct2;
         checkpoint
    esac
  etats
init
  ⟨nil, nil, atm⟩
snart
```

Figure 4.9: Electronic money transfer example: Pinger

value of the house

3) The `apprSrv` contacts the `sellSrv` and requests basic information about the house

4) The `apprSrv` combines the house specifications with other reference information to compute a tentative market price. This tentative market price is only an estimate which is not a definite appraisal until an on-site visit is made to the house to verify the accuracy of the original specifications

5) The `buySrv` makes an offer to the `sellSrv` based on the appraisal. The `buySrv` also contacts the `srchSrv` to perform a title search and the `lendSrv` to obtain a mortgage

6) The `lendSrv` contacts the `apprSrv` to confirm the appraisal information which is given after an on-site verification is completed

7) The `lendSrv` approves the mortgage and the `buySrv` will close the house purchase once it receives a response from the `srchSrv` and the `sellSrv` accepts the offer

```
behavior teller extends Transactor {
     private int acks = 0;
     private Object acked;

     public teller() {
          super(self);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void transfer(int delta, bankaccount inacct,
                          bankaccount outacct, pinger acct_pinger) {
          Object[] accts = {inacct, outacct};
          this.sendMsg("startPing", accts, acct_pinger);
          Object[] in_params = {delta, this.self()};
          Object[] out_params = {-1*delta, this.self()};
          this.sendMsg("adj", in_params, inacct);
          this.sendMsg("adj", out_params, outacct);
     }

     public void done(String msg) {
          standardOutput<-println(msg);
          this.setTState("acks", ((int)this.getTState("acks")) + 1);
          if (((int)this.getTState("acks")) == 2){
               this.stabilize();
          }
     }

     public void pingreq(Transactor requester) {
          this.sendMsg("ping", new Object[0], requester);
          this.checkpoint(); return;
     }

     public void ping(Transactor acct_pinger) {
          if (this.setTState("acked", null))
               this.sendMsg("ping", new Object[0], acct_pinger);
          else
               this.sendMsg("pingreq", new Object[0], acct_pinger);
     }
}
```

Figure 4.10: Teller account implementation

```
behavior bankaccount extends Transactor {
     private int bal = 0;

     public bankaccount(int balance) {
          super(self);
          this.setTState("bal", balance);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void adj(int delta, teller atm) {
          this.setTState("bal",
                         ((int)this.getTState("bal")) + delta);
          Object[] response = new Object[1];
          if (((int)this.getTState("bal")) < 0){
               response[0] = "Not enough funds!";
               this.sendMsg("done", response, atm);
               this.rollback(false, null); return;
          }
          else {
               this.stabilize();
               response[0] = "Balance update successful!";
               this.sendMsg("done", response, atm);
          }
     }

     public void pingreq(Transactor requester1,
                         Transactor requester2) {
          Object[] req = {requester2};
          this.sendMsg("pingreq", req, requester1)
     }

     public void ping() {
          this.checkpoint(); return;
     }
}
```

Figure 4.11: Bankaccount implementation

```
behavior pinger extends Transactor {
     private bankaccount acct1;
     private bankaccount acct2;
     private teller atm;

     public pinger() {
          super(self);
     }

     public void init(teller atm) {
          this.setTState("atm", atm);
          this.stabilize();
          this.checkpoint();
     }

     public void startPing(bankaccount acct1,
                           bankaccount acct2) {
          this.setTState("acct1", acct1);
          this.setTState("acct2", acct2);
          this.stabilize();
          this.sendMsg("ping", new Object[0], this.self());
     }

     public void ping() {
          Object[] me = {this.self()};
          this.sendMsg("ping", me,
                       ((Transactor)this.getTState("atm")));
     }

     public void pingreq() {
          Object[] ping1 = {((Transactor)this.getTState("atm")),
                            ((Transactor)this.getTState("acct2"))};
          Object[] ping2 = {((Transactor)this.getTState("atm")),
                            ((Transactor)this.getTState("acct1"))};
          this.sendMsg("pingreq", ping1,
                       ((Transactor)this.getTState("acct1")));
          this.sendMsg("pingreq", ping2,
                       ((Transactor)this.getTState("acct2")));
          this.checkpoint(); return;
     }
}
```

Figure 4.12: Pinger implementation

```
behavior teller extends Transactor {
     private int acks = 0;
     private Object acked;

     public teller() {
          super(self);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void transfer(int delta, bankaccount inacct,
                          bankaccount outacct) {
          Object[] in_params = {delta, this.self()};
          Object[] out_params = {-1*delta, this.self()};
          this.sendMsg("adj", in_params, inacct);
          this.sendMsg("adj", out_params, outacct);
     }

     public void done(String msg) {
          standardOutput<-println(msg);
          this.setTState("acks", ((int)this.getTState("acks")) + 1);
          if (((int)this.getTState("acks")) == 2){
               this.stabilize();
               this.sendMsg("endTransaction", new Object[0],
                    ((PingDirector)this.getTState("pingDirector")));
          }
     }
}
```

Figure 4.13: Teller implementation with CTP and `PingDirector`

The steps above describe a scenario where every step runs accordingly without any semantic failures. However, one possible way this house purchase may fail can be observed in step 6 in the case of the verification discovering inaccurate information. Upon this discovery the `apprSrv` voluntarily rolls back its state in order to reprocess the verified specifications. This in turn causes the mortgage information to be inconsistent with the information the `buySrv` has. As a result, the `buySrv` must also be caused to rollback due to this invalidated dependency where it may choose

```
behavior bankaccount extends Transactor {
     private int bal = 0;

     public bankaccount(int balance) {
          super(self);
          this.setTState("bal", balance);
     }

     public void initialize() {
          this.stabilize();
          this.checkpoint(); return;
     }

     public void adj(int delta, teller atm) {
          this.setTState("bal",
                         ((int)this.getTState("bal")) + delta);
          Object[] response = new Object[1];
          if (((int)this.getTState("bal")) < 0){
               response[0] = "Not enough funds!";
               this.sendMsg("done", response, atm);
               this.rollback(false, null); return;
          }
          else {
               this.stabilize();
               response[0] = "Balance update successful!";
               this.sendMsg("done", response, atm);
          }
     }
}
```

Figure 4.14: Bankaccount implementation with CTP and `PingDirector`

to renegotiate the sale price. Figure 4.15 depicts this failure scenario.

Figures 4.16, 4.17 4.18, 4.20, 4.21, 4.22, 4.23, and 4.19 show our implementation of this example written in our proposed language syntax. Figure 4.23 is an implementation of the on-site verification process and Figure 4.19 represents a credit database contacted by the lender in order to obtain the buyers credit history used to calculate the requested mortgage. `searchSrv`, `verifySrv`, `creditDB` are implemented as proxies because they only provide access to a resource in order to obtain information and thus will not have an effect on the global dependency. This

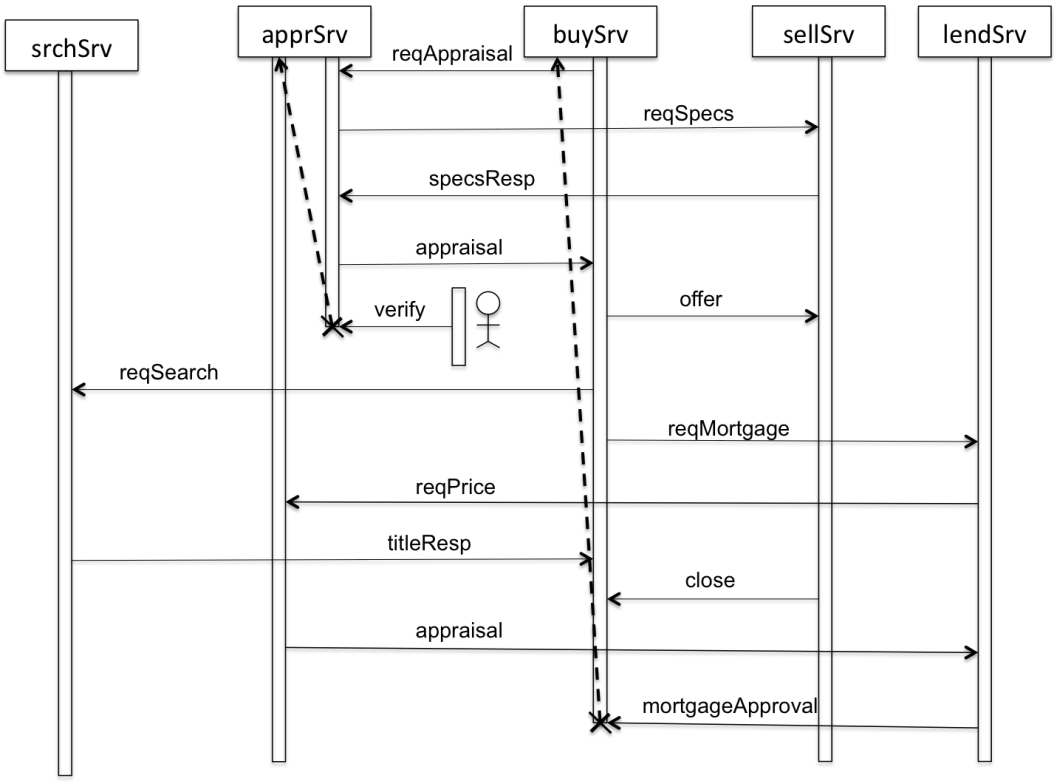

Figure 4.15: House purchase scenario involving semantic failure [1]

implementation also allows other types of failures to occur such as an offer rejection and mortgage denial. We make use of our *Consistent Transaction Protocol* and *Ping Director* in this example to manage the house purchase transaction to notify all participants of a failure or issue a global checkpoint so we arrive at a globally consistent state. This transaction is started by the following call by an outside agent:

```
Transactor[] participants = {<buySrv>, <sellSrv>, <apprSrv>,
                             <lendSrv>, <searchSrv>,
                             <verifySrv>, <creditDB>};
startTransaction(participants, <buySrv>,
                 "newHousePurchase", <houseid>);
```

An important observation can be made from this example highlighting how the transactor model tracks fine grained dependencies. Though the use of the CTP promotes atomicity of a transaction, its primary purpose is to guarantee consistency,

as the name suggests. The atomicity aspect of the CTP and the transactor model only applies to participants who are strictly invalidated by a dependency on a failed component. In that regard, other participants, such as the `srchSrv` who remains independent throughout the transaction, will not rollback even if another participant encounters failure. This key feature separates the transactor model from other traditional transaction methodologies that have an "all or nothing" approach. Like the `srchSrv`, any participant who is semantically not affected by the overall result of the transaction will not have its operations reverted. This offers benefits in terms of preventing unnecessary rollbacks and not having to redo the same task if the transaction is attempted again allowing it to reuse results without having to recompute them. The `srchSrv` is a highly simplified implementation of an actual title search service that would involve a much more complex process. This process locates the required information for the title to the house, and this result would have to be re-computed if the search service were to rollback. If the overall transaction does fail and is reattempted, that title information will still be persistent, allowing us to reuse resources. The fact that the `srchSrv` is implemented as a proxy also ensures us that it has no effect on the global dependency of the transaction and will not incur any upon itself. Similarly, the `verifySrv` and `creditDB` both being proxies have no effect on the rest of the transaction and will not be caused to rollback. An execution trace of our implementation is shown in Appendix B, Figures B.4, B.5, B.6, B.7, B.8, and B.9.

```
behavior sellSrv {
     HashMap minPrices, specs;
     int offeredPrice = 0;

     sellSrv(HashMap specsInfo, HashMap mins) {
          specs := specsInfo;
          minPrices := mins;
     }

     void initialize() {
          stabilize;
          checkpoint;
     }

     void reqSpecs(String houseId, Transactor customer) {
          customer<-specsResp(~!specs.get(houseId),
                              ~!minPrices.get(houseId));
     }

     void offer(String houseId, int price, buySrv buyer) {
          offeredPrice := price;
          if (price >= ~!minPrices.get(houseId)) {
               stabilize;
               buyer<-close();
          } else {
               buyer<-rejectOffer();
               rollback;
          }
     }
}
```

Figure 4.16: sellSrv implementation

```
behavior buySrv {
     searchSrv searcher;
     apprSrv appraiser;
     sellSrv seller;
     lendSrv lender;
     verifySrv verifier;
     creditDB creditHistory;
     int price = 0;
     String title, mortgage, houseid;

     buySrv(searchSrv srchr, apprSrv appr,
            sellSrv sellr, lendSrv lendr,
            verifySrv verifr, creditDB cHistory) {
          searcher := srchr;
          appraiser := appr;
          seller := sellr;
          lender := lendr;
          verifer := verifr;
          creditHistory := cHistory;
     }

     void initialize() {
          stabilize;
          checkpoint;
     }

     void newHousePurchase(String newHouseId) {
          houseid := newHouseId;
          ~!appraiser<-reqAppraisal(~!houseid, self,
                                    ~!seller, ~!verifier);
     }

     void appraisal(int newPrice) {
          price := newPrice;
          ~!seller<-offer(~!houseid, ~!price, self);
          ~!searcher<-reqSearch(~!houseid, self);
          ~!lender<-reqMortgage(~!houseid, self, ~!price,
                                ~!appraiser, ~!creditHistory);
     }
```

Figure 4.17: buySrv implementation

```
    void titleResp(String newTitle) {
        title := newTitle;
    }

    void mortgageApproval(String approvalid) {
        mortgage := approvalid;
    }

    void close() {
        if (~!title != null && ~!mortgage != null) {
            stabilize;
            endTransaction;
        } else {
            self<-close();
        }
    }

    void rejectOffer() {
        endTransaction;
        rollback;
    }

    void mortgageDeny() {
        endTransaction;
        rollback;
    }
}
```

Figure 4.18: buySrv implementation cont.

```
behavior proxy creditDB {
    creditDB() {}

    void initialize() {
        checkpoint;
    }

    void getCreditApproval(String houseid, buySrv buyer,
                           int price, lendSrv requester) {
        requester<-approvalResp("approval-" + houseid);
    }
}
```

Figure 4.19: creditDB implementation

```
behavior apprSrv {
     String house, specs;
     int price = 0;
     buySrv buyer;
     Transactor requester;
     verifySrv verifier;

     apprSrv() {}

     void initialize() {
          stabilize;
          checkpoint;
     }

     void reqAppraisal(String houseid, buySrv buyr,
                       sellSrv seller, verifySrv verifr) {
          buyer := buyr;
          house := houseid;
          verifier := verifr;
          seller<-reqSpecs(~!house, self);
     }

     void specsResp(String newSpecs, int newPrice) {
          specs := newSpecs;
          price := newPrice;
          ~!buyer<-appraisal(newPrice);
     }

     void reqPrice(Transactor customer) {
          requester := customer;
          ~!verifier<-verifySpecs(~!house, ~!specs, self);
     }

     void verify(boolean ok, int verifiedPrice) {
          if (ok) {
               stabilize;
               ~!requester<-appraisal(verifiedPrice);
          } else {
               ~!requester<-appraisal(verifiedPrice);
               rollback;
          }
     }
}
```

Figure 4.20: apprSrv implementation

```
behavior lendSrv {
     buySrv buyer;
     String house;
     int price = 0;
     creditDB creditAgency;

     lendSrv() {}

     void initialize() {
          stabilize;
          checkpoint;
     }

     void reqMortgage(String houseid, buySrv buyr, int reqPrice,
                      apprSrv appraiser, creditDB creditHistory) {
          house := houseid;
          price := reqPrice;
          buyer := buyr;
          creditAgency := creditHistory;
          appraiser<-reqPrice(self);
     }

     void appraisal(int newPrice) {
          price := newPrice;
          ~!creditAgency<-getCreditApproval(~!house, ~!buyer,
                                            ~!price, self);
     }

     void approvalResp(String approvalid) {
          if (approvalid != null) {
               stabilize;
               ~!buyer<-mortgageApproval(approvalid);
          } else {
               ~!buyer<-mortgageDeny();
               rollback;
          }
     }
}
```

Figure 4.21: lendSrv implementation

```
behavior proxy searchSrv {
     HashMap titlesDB;

     searchSrv(HashMap titlesInfo) {
          titlesDB := titlesInfo;
     }

     void initialize() {
          checkpoint();
     }

     void reqSearch(String houseId, Transactor customer) {
          customer<-titleResp(~!titlesDB.get(houseId));
     }
}
```

Figure 4.22: srchSrv implementation

```
behavior proxy verifySrv {
     HashMap specs, prices;

     verifySrv(HashMap newSpecs, HashMap newPrices) {
          specs := newSpecs;
          prices := newPrices;
     }

     void initialize() {
          checkpoint;
     }

     void verifySpecs(String houseid, String reqSpecs,
                      Transactor customer) {
          if (reqSpecs.equals(~!specs.get(houseid))) {
               customer<-verify(true, ~!prices.get(houseid));
          } else {
               customer<-verify(false, ~!prices.get(houseid));
          }
     }
}
```

Figure 4.23: verifySrv implementation

5. Related Work

5.1 Argus Programming Language and System

Argus [7] is a programming language and system developed to support distributed programs. Argus was designed to address the various problems that arise in a distributed system such as non-deterministic behavior of concurrent activities and node failures. To resolve these issues, Argus defines a set of abstractions to model distributed behavior. One abstraction is called the *guardian*. In Argus, guardians are very much akin to a SALSA actor. Like an actor, guardians are meant to encapsulate a resource and permit access to its resources through *handlers*. Direct access to these resources is strictly forbidden except through the use of handlers. Handlers are called by other guardians through a message based communication mechanism and arguments are passed by value to prevent shared access. Guardians reside in a single node and have the ability to migrate. Guardians can also create other guardians and pass names of guardians and handlers to each other. Unlike the single message processing implementation of a SALSA actor, guardians invoke a new process for each handler call and each guardian may contain one or more processes running concurrently.

Fault tolerance in Argus is provided with stable objects: resources within a guardian that are periodically written to stable storage to survive failures. This is opposed to volatile objects that are discarded in the event of a crash. These objects are implemented as *atomic objects*, which allocate access through the use of locks to resolve concurrency. Similar to a transactor persistent and volatile state, atomic objects use versioning to handle recovery from failures. A base version is written to stable storage and modifications are applied to a copy that will be committed to the base version if a transaction is successful.

Argus also defines another abstraction known as an *action* to model a transaction. Actions are atomic and serializable in nature. There is also a notion of sub actions enabling actions to be nested and run concurrently within a top action. Further concurrency control is enforced by lock inheritance and propagating version

changes up to the parent action. Successful sub actions push version changes to be inherited by the parent in a stack maintenance manner and the success of the top action would commit all changes to stable storage, otherwise all changes are discarded.

Though there are many similarities with Argus and actors, Argus takes a very different approach toward managing fault tolerant distributed state compared to transactors. Unlike transactors, Argus does not directly track dependencies and takes an "all or nothing" approach to determining if a set of operations should be committed. Another noticeable difference is the level of concurrency with a guardian where multiple messages are processed at the same time whereas messages are queued with transactors.

5.2 Atomos Transactional Programming Language

Atmos [8] is a transactional programming language with implicit transactions, strong atomicity, and scalable multiprocessor implementation. That is, it makes transactional memory operation behaviors implicit and uncommitted states of transactional code unseen, and abstracts transactional memory for multiprocessor implementation. Atomos relies on the transactional memory model which executes read and write instructions in an atomic way implemented through read and write sets as a collection of memory addresses. These read sets are used in Atomos to emulate the locking mechanism found in Argus but also as a failure detection technique.

Atomos declares transactions inside a `atomic` block which follows closed loop semantics in that reads to addresses in the transaction's write set outside of the transaction do not observe uncommitted changes and writes to addresses within the read set of a transaction cause a violation possibly causing a rollback. These situations are defined as an intersection of the read and write sets of two different transactions. Atomos also introduces new conditional waiting constructs through its `watch` and `retry` statements. A `watch` observes a particular object in a *watch set* that leverages read sets to observe violations to flag reevaluation of the value of the watch condition. A `retry` statement, usually used in conjunction with a `watch`, rolls back all operations within the transaction, yields the current thread,

and communicates the watch set to the scheduler to listen for violations.

Nested atomic transactions are also supported and inherit child read and write sets to the parent similar to how locks are inherited in the Argus implementation of nested transactions. Unlike Argus, but comparable to transactors, Atomos provides open nested transactions, which immediately commit child transactions at completion. Like transactors where independent agents of a failed transaction can still checkpoint, the rollback of a parent transaction is independent from completed open nested transactions.

5.3 Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs

Stabilizers [9] is a linguistic abstraction that models transient failures in concurrent threads with shared memory. These abstractions enforce global consistency by monitoring thread interactions to compute the transitive closure of dependencies. Like transactors, any non-local action such as thread communication or thread creation constitutes state dependency; however, these dependencies are recorded even if there is no state mutation. In the presence of transient failure, rollbacks are performed that revert state to immediately preceding some non-local action which become implicit checkpoints.

Stabilizers introduce three primitives: `stable`, `stabilize`, and `cut`. `stable` allows the user to define a particular region of code to be monitored for non-local actions and whose effects would be reverted as a single unit. `stabilize` is analogous to a transactor rollback, reverting execution to a dynamically calculated global state corresponding immediately prior to a stable section or non-local action. This state reversion rolls out thread communication and creation in pairs where the communicating partner and the created thread will also rollback. The third primitive delimits how far stabilization can occur. Therefore, a rollback cannot revert state past a `cut`. Cuts are useful to prevent rolling back irrevocable operations such as file I/O.

Unlike transactors, there is no predefined concrete checkpoint to rollback to since stabilizers perform thread monitoring instead of state captures. Therefore, a

`stabilize` call may not necessarily rollback to the dynamically closest stable section. Also, there is no notion of proxies since all communication is logged as a dependency. Another key difference between transactors and stabilizers is that stabilizers have eager evaluation of dependencies to immediately evaluate a consistent global state.

6. Discussion and Conclusions

6.1 Contributions

The main contribution of this paper is introducing a language that provides a comprehensive functional implementation of the transactor model. By creating a language that realizes the semantics of the transactor model we are able to shift away from the necessity of developing middleware to handle global state. Instead of building customized infrastructure to support distributed state, our transactor language provides the flexibility to compose various types of distributed programs while handing dependencies in message passing implicitly. This flexibility can exploited through the *Universal Storage Locator* giving users an option to define the best fault tolerant storage technique for their needs. Also, by having a language implementation, we are able to more thoroughly test the transactor model and promote further research into transactor program semantics. From being able to robustly test the transactor model, we are also able to optimize some of the operational semantics to yield a more efficient implementation such as the optimized version of the *Worldview Union Algorithm* presented in this paper. Lastly, a contribution made to the study of transactors is our introduction of the *Consistent Transaction Protocol* that highlights the restrictions and enhances the *Universal Checkpointing Protocol*. This new protocol abstracts a technique to promote progress in transactor programs and reach globally consistent states. Our proposed protocol also allows for programming transactions that follow traditional A.C.I.D. properties.[1]

6.2 Future Work

Though our language is currently a functional implementation of the transactor model, it is still in a developmental stage and leaves much work to be done. As a consequence of its development it also opens up new directions in the study of

[1] Atomicity is guaranteed for all dependent participants, isolation is assumed as a precondition of the protocol, and durability depends on the level of fault tolerance of the underlying storage system.

transactors. We summarize some of the key future work in the following.

6.2.1 Compiler

As with any programming language, a robust compiler is required to parse the syntactic support of a language and create an executable program. Currently, our transactor language can be written in SALSA with our transactor library. However, this involves a lot of tedious effort to call the constructs of the library. Our next objective would be to develop a compiler similar to the SALSA preprocessor to produce Java code that can be compiled and run on a JVM. This compiler would greatly simplify writing transactor programs with the proposed syntax, which inherits much of the familiar SALSA and Java grammar. A compiler would also allow our language to implicitly support object model semantics such as inheritance and polymorphism [2] as well as produce more efficient preprocessed code.

6.2.2 Node Failure

Modeling node failures is still left to be implemented in our language. Following the transition rules of the transactor model, a transactor system needs to be able to recognize node failures and reload transactors from persistent storage. A record of previously running transactors on the node would be required, perhaps as an extension of the naming service. The program would then proceed normally as if a rollback has occurred. This also opens up concerns on how to bootstrap programs and restart the network of messages. Along with bootstrapping programs there is an open question of whether to initially checkpoint the startup transactor to prevent total program annihilation if the startup node fails before it becomes persistent.

6.2.3 Isolation

One transaction property that the Consistent Transaction Protocol does not guarantee is isolation. In fact, this is a strict precondition in order for the protocol to be applied correctly. As an improvement of our proposed protocol we wish to explore methods that can establish isolation among participants of a given transaction. One possible technique is to apply a two-phase transaction initialization protocol similar

[2] This is currently only supported by modifying Java code preprocessed from SALSA code.

to the two-phase commit protocol. The necessity of a two-phase process is due to the message passing nature of transactors where there is no guarantee of when messages will arrive or even be received. Achieving isolation would be valuable so the user would only have to reason about the specifics of a transaction rather than consider its reliability.

6.2.4 Migration

Migration is a powerful ability to change location to another node to make use of environmental services residing in that node or locate more optimal computational resources [11]. SALSA has built in support for actor migration and our transactor language allows transactors to be initialized in different SALSA theaters. However, transactor migration is still an area of research. Under the transactor model, migration is not explicitly modeled by the tau calculus and there are concerns over whether location is represented by a transactor's state where an implementation would have to perform reverse migrations should a transactor ever rollback. Migration also becomes a factor in implementing node failure where each node would have to track which transactors would have to be recovered. Transactor USL was developed to permit the possibility of mobile transactors so persistent state storage would not become a limiting factor.

6.2.5 Garbage Collection

While research has been done to implement distributed garbage collection in SALSA [10], we have disabled the SALSA garbage collector in our current transactor implementation. In order to correctly apply garbage collection to transactors, the SALSA garbage collector needs to be extended to accommodate rollbacks in order to reclaim failed transactors and correctly recognize new references to reloaded transactors.

6.2.6 Continuations

Currently, in order to retrieve information from another transactor, the sender's name needs to be passed along with the message so the recipient knows where to send a reply. This interaction between transactors can be simplified by implementing

continuations. Continuations would make it easier to compose transactor programs by emulating serialized execution among asynchronous transactors. SALSA provides this in the form of *tokens*. However, research needs to be done to consider how to model tokens in tau calculus under the transactor model so that dependencies can be applied correctly.

REFERENCES

[1] J. Field and C. Varela, "Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments," unpublished.

[2] B. Boodman, "Implementing and verifying the safety of the transactor model," M.S. thesis, Dept. Comp. Sci., R.P.I., Troy, NY, 2008.

[3] C. Varela, "Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination," Ph.D. dissertation, Dept. Comp. Sci., U. of Illinois at Urbana-Champaign, 2001.

[4] C. Varela et al., "The salsa programming language: 1.1.2 release tutorial," Dept. Comp. Sci., R.P.I., Troy, NY, Tech. Rep. 07-12, 2007.

[5] R. Hickey. (2014). *Values and Change - Clojure's approach to Identity and State,* [Online]. Available: http://www.clojure.org/state, [Accessed: 1 July 2014].

[6] Amazon Web Services. (2014). *Amazon Simple Storage Service Documentation,* [Online]. Available: http://aws.amazon.com/documentation/s3/ [Accessed: 1 July 2014].

[7] B. Liskov, "Distributed programming in argus," *Commun. ACM*, vol. 31, no. 3, pp. 300-312, Mar. 1988.

[8] B. D. Carlstrom et al., "The atomos transactional programming language," in *2006 ACM SIGPLAN Conf. Programming Language Design and Implementation*, New York, NY, 2006, pp. 1-13.

[9] L. Ziarek et al., "Stabilizers: a modular checkpointing abstraction for concurrent functional programs," in *Proc. 11th ACM SIGPLAN Int. Conf. Functional Programming*, New York, NY, 2006, pp. 136-147.

[10] W. Wang, "Distributed Garbage Collection for Large-Scale Mobile Actor Systems," Ph.D. dissertation, Dept. Comp. Sci., R.P.I., Troy, NY, 2006.

[11] C. Varela, *Programming Distributed Computing Systems: A Foundational Approach*. Cambridge, MA: MIT Press, 2013.

APPENDIX A
Transactor Class

```
public class Transactor extends UniversalActor {
     public class State extends UniversalActor .State {
          private Worldview wv;
          private String name;
          private URI USL;
          public PingDirector pingDirector;

          public void recvMsg(Message msg, Worldview msg_wv);
          public void sendMsg(String method, Object[] params,
                              Transactor recipient);

          public void stabilize();
          public void checkpoint();
          public void rollback(boolean force, Worldview updatedWV);

          public Transactor newTActor(Transactor new_T);
          public boolean dependent();
          public Transactor self();

          public boolean setTState(String field, Object newValue);
          public Object getTState(String field);

          public void startTransaction(Transactor[] participants,
                                       Transactor coordinator,
                                       String msg, Object[] msg_args);

          public void transactionStart(String msg, Object[] msg_args,
                                        PingDirector director);
          public void pingreq(Transactor[] pingreqs);
          public void ping();
     }
}
```

Figure A.1: Transactor class skeleton

APPENDIX B
Example Execution Outputs

The following figures shows the execution traces of our bank transfer and house purchase examples. We present the initial worldviews of all participating transactors before starting the transaction and the resulting worldviews after the transaction completes. Each printed worldview output is labeled by the transactor implementation, the transactor name mapped to its history where history displays the current volatility and incarnation value followed by the list of checkpointed incarnations inside the square brackets, its current history map, dependency graph, and root set. Figures B.1 shows the initial state of transactors before issuing a balance transfer. Figures B.2 and B.3 shows the state of transactors after a successful and failed transaction of transferring a balance of 50 and 500 respectively. The current balance of bank accounts and number of acks for the teller is also shown in our output. Figures B.4 and B.5 shows the initial state of transactors before issuing a house purchase transaction. Figures B.6 and B.7 shows the state of transactors after a successful transaction and Figures B.8 and B.9 shows a failed transaction. Transactor names `1` and `67` in this example refer to the outside transactor which triggers the transaction and the ping director respectively. Note that `srchSrv`, `verifySrv`, and `creditDB` are implemented as proxy so they remain in a stable state no matter the outcome of the transaction.

```
SAVINGS ACCOUNT:
Balance: 100
savings -> V(0) [ 0  ]
History Map:
savings->V(0) [ 0  ],
Dep Graph:

Root Set:
savings,

CHECKING ACCOUNT:
Balance: 100
checking -> V(0) [ 0  ]
History Map:
checking->V(0) [ 0  ],
Dep Graph:

Root Set:
checking,

TELLER:
Acks: 0
teller -> V(0) [ 0  ]
History Map:
teller->V(0) [ 0  ],
Dep Graph:

Root Set:


PINGER:
pinger -> V(0) [ 0  ]
History Map:
pinger->V(0) [ 0  ],
Dep Graph:

Root Set:
```

Figure B.1: Bank transfer example initial state

```
Balance update successful!
Balance update successful!
==================================
SAVINGS ACCOUNT:
Balance: 150
savings -> V(0) [ 0 0  ]
History Map:
savings->V(0) [ 0 0  ],
Dep Graph:

Root Set:
savings,

CHECKING ACCOUNT:
Balance: 50
checking -> V(0) [ 0 0  ]
History Map:
checking->V(0) [ 0 0  ],
Dep Graph:

Root Set:
checking,

TELLER:
Acks: 2
teller -> V(0) [ 0 0  ]
History Map:
teller->V(0) [ 0 0  ], savings->S(0) [ 0  ],
pinger->S(0) [ 0  ], checking->S(0) [ 0  ],
Dep Graph:

Root Set:

PINGER:
pinger -> V(0) [ 0 0  ]
History Map:
pinger->V(0) [ 0 0  ],
Dep Graph:

Root Set:
```

Figure B.2: Bank transfer example checkpointed state

```
Balance update successful!
Not enough funds!
===================================
SAVINGS ACCOUNT:
Balance: 100
savings -> V(1) [ 0  ]
History Map:
teller->V(1) [ 0  ], savings->V(1) [ 0  ],
pinger->S(0) [ 0  ], checking->V(1) [ 0  ],
Dep Graph:

Root Set:
savings,

CHECKING ACCOUNT:
Balance: 100
checking -> V(1) [ 0  ]
History Map:
teller->V(1) [ 0  ], savings->V(1) [ 0  ],
pinger->S(0) [ 0  ], checking->V(1) [ 0  ],
Dep Graph:

Root Set:
checking,

TELLER:
Acks: 0
teller -> V(1) [ 0  ]
History Map:
teller->V(1) [ 0  ], savings->V(1) [ 0  ],
pinger->S(0) [ 0  ], checking->V(1) [ 0  ],
Dep Graph:

Root Set:

PINGER:
pinger -> V(0) [ 0 0  ]
History Map:
pinger->V(0) [ 0 0  ],
Dep Graph:

Root Set:
```

Figure B.3: Bank transfer example failed state

```
SEARCHER:
srchSrv -> V(0) [  ]
History Map:
srchSrv->V(0) [  ],
Dep Graph:

Root Set:


SELLER:
sellSrv -> V(0) [ 0  ]
History Map:
sellSrv->V(0) [ 0  ],
Dep Graph:

Root Set:


VERIFIER:
verifySrv -> V(0) [  ]
History Map:
verifySrv->V(0) [  ],
Dep Graph:

Root Set:


APPRAISER:
apprSrv -> V(0) [ 0  ]
History Map:
apprSrv->V(0) [ 0  ],
Dep Graph:

Root Set:
```

Figure B.4: House purchase example initial state

```
CREDITOR:
creditDB -> V(0) [  ]
History Map:
creditDB->V(0) [  ],
Dep Graph:

Root Set:


LENDER:
lendSrv -> V(0) [ 0  ]
History Map:
lendSrv->V(0) [ 0  ],
Dep Graph:

Root Set:


BUYER:
buySrv -> V(0) [ 0  ]
History Map:
buySrv->V(0) [ 0  ],
Dep Graph:

Root Set:
```

Figure B.5: House purchase example initial state cont.

```
SEARCHER:
srchSrv -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], srchSrv->S(0) [  ],
TransDirector->S(0) [  ], sellSrv->V(0) [ 0  ], apprSrv->V(0) [ 0  ],
buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, apprSrv<-67, apprSrv<-1,
apprSrv<-sellSrv, apprSrv<-apprSrv, apprSrv<-buySrv, buySrv<-67,
buySrv<-1, buySrv<-sellSrv, buySrv<-apprSrv, buySrv<-buySrv,
Root Set:
67, 1, srchSrv, sellSrv, apprSrv, buySrv,

SELLER:
sellSrv -> V(0) [ 0 0  ]
History Map:
67->S(0) [  ], creditDB->S(0) [  ], 1->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], lendSrv->S(0) [ 0  ],
TransDirector->S(0) [  ], sellSrv->V(0) [ 0 0  ],
apprSrv->S(0) [ 0  ], buySrv->S(0) [ 0  ],
Dep Graph:

Root Set:

VERIFIER:
verifySrv -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], verifySrv->S(0) [  ],
lendSrv->V(0) [ 0  ], TransDirector->S(0) [  ],
apprSrv->V(0) [ 0  ], sellSrv->V(0) [ 0  ],
buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, lendSrv<-67, lendSrv<-1,
lendSrv<-sellSrv, lendSrv<-apprSrv, lendSrv<-buySrv, apprSrv<-67,
apprSrv<-1, apprSrv<-sellSrv, apprSrv<-apprSrv, apprSrv<-buySrv,
buySrv<-67, buySrv<-1, buySrv<-sellSrv, buySrv<-apprSrv,
buySrv<-buySrv,
Root Set:
67, 1, verifySrv, sellSrv, apprSrv, buySrv,
```

Figure B.6: House purchase example checkpointed state

```
APPRAISER:
apprSrv -> V(0) [ 0 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ],
srchSrv->S(0) [  ], lendSrv->S(0) [ 0  ], TransDirector->S(0) [  ],
apprSrv->V(0) [ 0 0  ], sellSrv->S(0) [ 0  ], buySrv->S(0) [ 0  ],
Dep Graph:

Root Set:

CREDITOR:
creditDB -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], lendSrv->V(0) [ 0  ],
TransDirector->S(0) [  ], apprSrv->S(0) [ 0  ], sellSrv->V(0) [ 0  ],
buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, lendSrv<-67, lendSrv<-1,
lendSrv<-verifySrv, apprSrv<-1, buySrv<-1,
lendSrv<-apprSrv, lendSrv<-sellSrv, lendSrv<-buySrv, apprSrv<-67,
apprSrv<-sellSrv, apprSrv<-apprSrv, apprSrv<-buySrv, buySrv<-67,
buySrv<-sellSrv, buySrv<-apprSrv, buySrv<-buySrv,
Root Set:
67, 1, verifySrv, lendSrv, sellSrv, apprSrv, buySrv,

LENDER:
lendSrv -> V(0) [ 0 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], apprSrv->S(0) [ 0  ],
buySrv->S(0) [ 0  ], lendSrv->V(0) [ 0 0  ],
TransDirector->S(0) [  ], sellSrv->S(0) [ 0  ],
Dep Graph:

Root Set:

BUYER:
buySrv -> V(0) [ 0 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ],
lendSrv->S(0) [ 0  ], TransDirector->S(0) [  ], sellSrv->S(0) [ 0  ],
apprSrv->S(0) [ 0  ], buySrv->V(0) [ 0 0  ],
Dep Graph:

Root Set:
```

Figure B.7: House purchase example checkpointed state cont.

```
SEARCHER:
srchSrv -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], srchSrv->S(0) [  ],
TransDirector->S(0) [  ], sellSrv->V(0) [ 0  ],
apprSrv->V(0) [ 0  ], buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, apprSrv<-67, apprSrv<-1,
apprSrv<-sellSrv, apprSrv<-apprSrv, apprSrv<-buySrv, buySrv<-67,
buySrv<-1, buySrv<-sellSrv, buySrv<-apprSrv, buySrv<-buySrv,
Root Set:
67, 1, srchSrv, sellSrv, apprSrv, buySrv,

SELLER:
sellSrv -> V(1) [ 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], lendSrv->V(1) [ 0  ],
TransDirector->S(0) [  ], sellSrv->V(1) [ 0  ], apprSrv->V(1) [ 0  ],
buySrv->V(1) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector,
Root Set:

VERIFIER:
verifySrv -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], verifySrv->S(0) [  ],
lendSrv->V(0) [ 0  ], TransDirector->S(0) [  ],
apprSrv->V(0) [ 0  ], sellSrv->V(0) [ 0  ], buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, lendSrv<-67, lendSrv<-1,
lendSrv<-sellSrv, lendSrv<-apprSrv, lendSrv<-buySrv, apprSrv<-67,
apprSrv<-1, apprSrv<-sellSrv, apprSrv<-apprSrv, apprSrv<-buySrv,
buySrv<-67, buySrv<-1, buySrv<-sellSrv, buySrv<-apprSrv,
buySrv<-buySrv,
Root Set:
67, 1, verifySrv, sellSrv, apprSrv, buySrv,
```

Figure B.8: House purchase example failed state

```
APPRAISER:
apprSrv -> V(1) [ 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], lendSrv->V(1) [ 0  ],
TransDirector->S(0) [  ], sellSrv->V(1) [ 0  ], apprSrv->V(1) [ 0  ],
buySrv->V(1) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector,
Root Set:

CREDITOR:
creditDB -> S(0) [  ]
History Map:
67->V(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], lendSrv->V(0) [ 0  ], TransDirector->S(0) [  ],
apprSrv->V(0) [ 0  ], sellSrv->V(0) [ 0  ], buySrv->V(0) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector, lendSrv<-67, lendSrv<-1,
lendSrv<-verifySrv, lendSrv<-apprSrv, lendSrv<-sellSrv,
lendSrv<-buySrv, apprSrv<-67, apprSrv<-1, apprSrv<-sellSrv,
apprSrv<-apprSrv, apprSrv<-buySrv, buySrv<-67, buySrv<-1,
buySrv<-sellSrv, buySrv<-apprSrv, buySrv<-buySrv,
Root Set:
67, 1, verifySrv, lendSrv, sellSrv, apprSrv, buySrv,

LENDER:
lendSrv -> V(1) [ 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], lendSrv->V(1) [ 0  ],
TransDirector->S(0) [  ], sellSrv->V(1) [ 0  ], apprSrv->V(1) [ 0  ],
buySrv->V(1) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector,
Root Set:

BUYER:
buySrv -> V(1) [ 0  ]
History Map:
67->S(0) [  ], 1->S(0) [  ], creditDB->S(0) [  ],
verifySrv->S(0) [  ], srchSrv->S(0) [  ], lendSrv->V(1) [ 0  ],
TransDirector->S(0) [  ], sellSrv->V(1) [ 0  ], apprSrv->V(1) [ 0  ],
buySrv->V(1) [ 0  ],
Dep Graph:
67<-67, 67<-1, 67<-TransDirector,
Root Set:
```

Figure B.9: House purchase example failed state cont.

APPENDIX C
Symbol List

$h_1 \rightarrow_{\Diamond} h_2$	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'
$h_1 \nrightarrow h_2$	h_1 rolls back to h_2
$h_1 \rightarrow_{\surd} h_2$	h_1 checkpoints to h_2
$h_1 \rightsquigarrow h_2$	h_1 is succeeded by h_2
$h_1 \rightsquigarrow_* h_2$	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 $(\gamma_2, \delta_2, \_)$

Figure C.1: Symbol list [2]