Simulating Abstract State Machines (ASMs) with Concurrent Transaction Logic (CTR)

WSMO Working Draft – August 10, 2007

Authors:
Dumitru Roman
Michael Kifer

This version:
http://www.wsmo.org/TR/d14/d14.2/v0.1/20070810/

Latest version:
http://www.wsmo.org/TR/d14/d14.2/v0.1/

Previous version:
http://www.wsmo.org/TR/d14/d14.2/v0.1/

Copyright © 2007 DERI®, All Rights Reserved. DERI liability, trademark, document use, and software licensing rules apply.
Abstract

ASMs provide a computational model but they do not come with a reasoning mechanism. One of the important challenges for the semantic Web services is to enable reasoning about service choreographies. A transformation from ASMs to CTR could be a first step to enable such reasoning. In this deliverable we propose one such transformation.
Contents

1 Introduction 4
  1.1 Overview of CTR 4
  1.2 Overview of ASMs 6

2 Transformation of Sequential ASMs in CTR 9
  2.1 Representing content locations in CTR 9
  2.2 Labeling Rules with unique IDs 9
  2.3 Transformation 10
    2.3.1 Step 1 11
    2.3.2 Step 2 12

3 Summary and Outlook 14

Bibliography 14
1 Introduction

ASMs provide a computational model but they do not come with a reasoning mechanism. One of the important challenges for the semantic Web services is to enable reasoning about service choreographies. A transformation from ASMs to CTR could be a first step to enable such reasoning. In this deliverable we propose one such transformation. Concurrent Translation Logic (CTR) [1] is a logical framework for declarative specification, analysis, and execution of database transactions; we give an informal overview of CTR in Sub-section 1.1. Abstract State Machines (ASMs) [4] provide a general model of computations; they are introduced in Sub-section 1.2. Section 2 gives a possible simulation of ASMs in CTR by providing a translation of ASMs transitions rules to CTR programs. Finally, Section 3 provides a summary and outlook of this document.

1.1 Overview of CTR

This section provides a short summary of the relevant parts of CTR syntax and informally explains the semantics. This should be enough for the purpose of simulating ASMs. For detailed information on the model theory of the logic or its proof theory, the reader is referred to [1].

CTR is a conservative extension of the classical predicate logic in the sense that both its proof theory and the model theory reduce to classical logic for formulas that do not cause state transitions (but only query the current state).

**Basic syntax.** The atomic formulas of CTR are identical to those of the classical logic, i.e., they are expressions of the form $p(t_1, \ldots, t_n)$, where $p$ is a predicate symbol and the $t_i$'s are function terms. More complex formulas are built with the help of connectives and quantifiers.

Apart from the classical $\lor$, $\land$, $\lnot$, $\forall$, and $\exists$, CTR has two additional connectives, $\otimes$ (serial conjunction) and $\parallel$ (concurrent conjunction), and a modal operator $\diamond$ (isolated execution). For instance, $\diamond(p(X) \otimes q(X)) \parallel (\forall Y(r(Y) \lor s(X,Y)))$ is a well-formed formula.

**Informal semantics.** Underlying the logic and its semantics is a set of database *states* and a collection of *paths*. For the purpose of this paper, the reader can think of the states as just a set of relational databases, but the logic does not rely on the exact nature of the states—it can deal with a wide variety of them.

A *path* is a finite sequence of states. For instance, if $s_1, s_2, \ldots, s_n$ are database states, then $(s_1), (s_1, s_2)$, and $(s_1, s_2, \ldots, s_n)$ are paths of length 1, 2, and $n$, respectively.

Just as in classical logic, CTR formulas assume truth values. However, unlike classical logic, the truth of CTR formulas is determined over paths, not at states. If a formula, $\phi$, is true over a path $(s_1, \ldots, s_n)$, it means that $\phi$ can *execute* starting at state $s_1$. During the execution, the current state will change to $s_2, s_3, \ldots$, etc., and the execution terminates at state $s_n$.
With this in mind, the intended meaning of the CTR connectives can be summarized as follows:

• $\phi \otimes \psi$ means: execute $\phi$ then execute $\psi$. Or, model-theoretically, $\phi \otimes \psi$ is true over a path $\langle s_1, ..., s_n \rangle$ if $\phi$ is true over a prefix of that path, say $\langle s_1, ..., s_i \rangle$, and $\psi$ is true over the suffix $\langle s_i, ..., s_n \rangle$.

• $\phi | \psi$ means: $\phi$ and $\psi$ must both execute concurrently, in an interleaved fashion.

• $\phi \land \psi$ means: $\phi$ and $\psi$ must both execute along the same path. In practical terms, this is best understood in terms of constraints on the execution. For instance, $\phi$ can be thought of as a transaction and $\psi$ as a constraint on the execution of $\phi$.

• $\phi \lor \psi$ means: execute $\phi$ or execute $\psi$ non-deterministically.

• $\neg \phi$ means: execute in any way, provided that this will not be a valid execution of $\phi$. Negation is an important ingredient in temporal constraint specifications.

• $\odot \phi$ means: execute $\phi$ in isolation, i.e., without interleaving with other concurrently running activities.

**Concurrent-Horn subset of CTR.** Implication $p \leftarrow q$ is defined as $p \lor \neg q$. The form and the purpose of the implication in CTR is similar to that of Datalog: $p$ can be thought of as the name of a procedure and $q$ as the definition of that procedure. However, unlike Datalog, both $p$ and $q$ assume truth values on execution paths, not at states.

More precisely, $p \leftarrow q$ means: if $q$ can execute along a path $\langle s_1, ..., s_n \rangle$, then so can $p$. If $p$ is viewed as a subroutine name, then the meaning can be re-phrased as: one way to execute $p$ is to execute its definition, $q$.

The control flow parts of service choreographies are formally represented using concurrent-Horn goals and concurrent Horn rules. A concurrent Horn goal is:

• any atomic formula is a concurrent-Horn goal;

• $\phi \otimes \psi$, $\phi | \psi$, and $\phi \lor \psi$ are concurrent-Horn goals, if so are $\phi$ and $\psi$;

• $\odot \phi$ is a concurrent-Horn goals, if so is $\phi$.

A concurrent-Horn rule is a CTR formula of the form $head \leftarrow body$, where $head$ is an atomic formula and $body$ is a concurrent-Horn goal.

Observe that the definition of concurrent-Horn rules and goals does not include the connective $\land$. In general, $\land$ represents constrained execution, which is usually hard to implement, since constraints must be checked at every step of the execution. If a constraint violation is detected, a new execution path must be tried out. In contrast, the concurrent-Horn fragment of CTR is efficiently implementable, and there is an SLD-style proof procedure that proves concurrent-Horn formulas and executes them at the same time [1].

The efficiency gap between concurrent-Horn execution and constrained execution is the main motivation for our results. In a previous work by one of the authors [5], it was shown that for a certain class of constraints, formulas of the form $ConcurrentHornGoal \land Constraints$, have an equivalent concurrent-Horn
representation (which, therefore, does not use the connective $\land$). This enabled the use of the proof theory for Horn CTR as a means of obtaining a linear run-time scheduling algorithm (as opposed to, for example, exponential run-time scheduling in [7, 8]).

**Elementary updates.** We complete our informal introduction to CTR by explaining how execution of (some) formulas may actually change the underlying database state. Most of the machinery has already been introduced (albeit very informally). What is missing is the notion of elementary updates.

In CTR, elementary updates are represented by ordinary atomic, variable-free formulas. Syntactically, CTR does not distinguish elementary updates in any way, but the user may want to do so by adopting a syntactic convention (e.g., a convention could be that $\text{insert.p}(t)$ represents the act of insertion of tuple $t$ into the relation $p$).

What distinguishes elementary updates is their semantics. Through some black magic, called transition oracle, CTR arranges that each elementary update is always true along certain arcs, i.e., paths of the form $\langle s_1, s_2 \rangle$. Informally, one can think of an elementary update as a binary relation over states. For instance, if $\langle s_1, s_2 \rangle$ belongs to the relation corresponding to an elementary update $u$, it means that $u$ can cause a transition from state $s_1$ to state $s_2$. Note that an update can be non-deterministic (any one of a number of alternative state transitions might be possible) and it is possible for an update to be inapplicable in certain states (for instance, $\text{delete.p}(t)$ may be applicable only if $p(t)$ is true in the current state).

This mechanism is very general. It accounts for a wide variety of elementary state changes: from simple tuple insertions and deletions, to relational assignments, to updates performed by legacy programs, to whatever workflow activities might do. The connectives of CTR are then used to build more complex updates from the elementary ones and then to combine these complex updates into even more complex update programs. This process of building CTR programs from the ground up is very natural and powerful. The reader is referred to [2, 3, 1] for concrete examples.

### 1.2 Overview of ASMs

Abstract State Machines [4], formally called Evolving Algebras [6], are a general computation model which has been applied in the area of software engineering for systems design and analysis. This section provides a summary of the relevant concepts of ASMs which should be enough for the purpose of this paper. For detailed information on ASMs, the reader is referred to [4].

**Abstract States and Updates.** The states of ASMs are algebraic structures, as introduced in standard mathematical logic. A *signature* $\Sigma$ (or *vocabulary*) is a finite collection of function names. Each function name has an *arity*, a non-negative integer. A *state* $\mathfrak{A}$ for the signature $\Sigma$ is a non-empty set $X$, the *superuniverse* of $\mathfrak{A}$, together with *interpretations* of the function names of $\Sigma$. If $f$ is an $n$-ary function name of $\Sigma$ then its interpretation is $f^\mathfrak{A}$ is a function
from $X^n$ into $X$; if $c$ is a constant of $\Sigma$, then its interpretation $c^A$ is an element of $X$. The superuniverse $X$ of the state $A$ is denoted by $|A|$. The elements of a state are the elements of the superuniverse of the state.

A location of $A$ is a pair $(f,(a_1, ..., a_n))$, where $f$ is an $n$-ary function name and $a_1, ..., a_n$ are elements of $|A|$. The value $f^A(a_1, ..., a_n)$ is called the content of the location in $A$. The elements of the location are the elements of the set $\{a_1, ..., a_n\}$.

An update for $A$ is a pair $(l, v)$ where $l$ is a location of $A$ and $v$ is an element of $|A|$. An update set is a set of updates. An update set $U$ is said to be consistent if it has no clashing updates, i.e. for any location $l$, and all elements $v, w$, it is true that if $(l, w) \in U$ and $(l, w) \in U$, then $v = w$.

**Transition Rules and Runs of ASMs.** States are dynamic and they evolve by being updated during computations. Updating abstract states means to change the interpretations of (some of) the functions of the underlying signature. The way ASMs update states is described by transition rules of the following form which define the syntax of ASM programs. The transition rules $P, Q$ are syntactic expressions generated as follows:

1. **Skip Rule:** $skip$
   Meaning: Do nothing.

2. **Update Rule:** $f(s_1, ..., s_n) := t$
   Meaning: Update the value of $f$ at $(s_1, ..., s_n)$ to $t$.

3. **Block Rule:** $P \ par Q$
   Meaning: $P$ and $Q$ are executed in parallel.

4. **Conditional Rule:** $if \ \varphi \ then \ P \ else \ Q$
   Meaning: If $\varphi$ is true, then execute $P$, otherwise execute $Q$.

5. **Let Rule:** $let \ x = t \ in \ P$
   Meaning: Assign the value of $t$ to $x$ and then execute $P$.

6. **Forall Rule:** $forall \ x \ with \ \varphi \ do \ P$
   Meaning: Execute $P$ in parallel for each $x$ satisfying $\varphi$.

7. **Choose Rule:** $choose \ x \ with \ \varphi \ do \ P$
   Meaning: Choose an $x$ satisfying $\varphi$ and then execute $P$.

8. **Sequence Rule:** $P \ seq Q$
   Meaning: $P$ and $Q$ are executed sequentially, first $P$, and then $Q$.

9. **Call Rule:** $r(t_1, ..., t_n)$
   Meaning: Call transition rule $r$ with parameters $t_1, ..., t_n$.

The variables $x$ in $let$, $forall$ and $choose$ are logical variables. Their values can not be updated by a transition rule are not stored in the state but in a finite environment. The scope of $x$ in $let$ is the rule $P$, whereas the scope of $x$ in $forall$ and $choose$ is the formula $\varphi$ and the transition rule $P$.

A rule declaration for a rule name $r$ of arity $n$ is an expression $r(x_1, ..., x_n) = P$, where $P$ is a transition rule and the free variables of $P$ are contained in the list $x_1, ..., x_n$. 
An **abstract state machine** $M$ consists of a signature $\Sigma$, a set of initial states for $\Sigma$, a set of rule declarations, and a distinguished rule name of arity zero called the **main rule name** of the machine. A **run** of $M$ is a finite or infinite sequence $A_0, A_1, \ldots$ of states for $\Sigma$ such that $A_0$ is an initial state of $M$, and for each $n$, either $M$ can make a move from $A_n$ into the next internal state $A'_n$ and the environment produces a consistent set of external or shared updates $U$ such that $A_{n+1} = A'_n + U$, or $M$ cannot make a move in state $A_n$ and $A_n$ is the last state in the run.
2 Transformation of Sequential ASMs in CTR

We start this section by giving a representation of ASMs content locations in CTR (Subsection 2.1), followed by a mechanism for associating rules with unique identifiers (Subsection 2.2), and then the actual transformation in Subsection 2.3.

2.1 Representing content locations in CTR

One of the core concepts of ASMs is the content, \( v \), of a location \( (f, (a_1, \ldots, a_n)) \), where \( v \) is an element of \(|\mathcal{A}|\), \( f \) is an \( n \)-ary function name and \( a_1, \ldots, a_n \) are elements of \(|\mathcal{A}|\). In order to capture the concept of content in CTR, we introduce the predicate

\[
\text{content}(L, V)
\]

where \( L \) is a variable for locations (like \((f, (a_1, \ldots, a_n))\) above) and \( V \) is a variable for values (like \( v \) above).

2.2 Labeling Rules with unique IDs

In order to cope with sequential ASMs, our transformation is based on the unique identification of rules in an ASM program. In the following we define an inductive mechanism, \( \text{Id} : \text{main.tex} \), for associating rules with unique identifiers. Strings are used as identifiers and "+" denotes string concatenation:

\[
\text{Id}(\text{Rule}) = \begin{cases} 
'' & \text{if Rule is the main rule of the ASM} \\
\text{id} + "a" & \text{if Id(Rule seq Q) = id} \\
& \text{or Id(Rule par Q) = id} \\
& \text{or Id(if } \varphi \text{ then Rule else Q) = id} \\
& \text{or Id(let } x = t \text{ in Rule) = id} \\
& \text{or Id(choose X with } \varphi \text{ do Rule) = id} \\
& \text{or Id(forall X with } \varphi \text{ do Rule) = id} \\
\text{id} + "b" & \text{if Id(P seq Rule) = id} \\
& \text{or Id(P par Rule) = id} \\
& \text{or Id(if } \varphi \text{ then P else Rule) = id.}
\end{cases}
\]

For simplicity we assume the id associated with the main rule of the ASM to be the empty string "". The idea is that for each rule a new character
("a" or "b") is concatenated with the id of the "super-rule" the rule belongs to. To ensure that the $P$ and $Q$ rules get unique ids for the sequence, block, and conditional rules, we add an "a" for $P$ rules and an "b" for $Q$ rules.

## 2.3 Transformation

We are now ready to define the transformation $Trans$ which transforms a sequential ASM program into an equivalent CTR program. Since CTR is based on interleaved execution of updates, it can not directly capture parallel updates in the sense of ASMs. However, we can simulate such parallelism through a two-step mechanism. In the first step we capture in CTR the structure of the ASM program, and in the second step we check the consistency of updates, and do the actual updates:

- **Step 1**: based on the structure of ASMs we recursively generate a CTR program which, if executed, instead of doing the actual updates, it only keeps track of the updates. For keeping track of the updates a log-based mechanism is used. That is, for example instead of deleting $content(f, (a_1, ..., a_n), v)$ and inserting $content(f, (a_1, ..., a_n), t)$, we only insert $log(id, f(a_1, ..., a_n), t)$, where $id$ is the id of the update rule.

- **Step 2**: whereas the execution of ASMs can result in inconsistent updates, the execution of the CTR program generated in step one does not detect such inconsistencies, since it only keeps track of the updates. The detection of inconsistencies and the actual updates take place in step two. In this way we can correctly capture the behavior of ASM programs in CTR.

$Trans$ is defined as a sequential composition of the two steps as follows:

$$Trans(ASM) = T(id_{MainRule}, MainRule) \otimes post_{trans} \quad (2.2)$$

$MainRule$ is the ASM rule which is considered as the main rule of the ASM program, and $id_{MainRule}$ is the id associated with the main rule. $T$ is a transformation which reflects the first step and is defined in Subsection 2.3.1. $post_{trans}$ reflects the second step and is defined in Subsection 2.3.2.
### 2.3.1 Step 1

The transformation \( T \) captures the structure of ASM programs in CTR; it is defined recursively as follows:

\[
T(id, Rule) = \begin{cases} 
\text{state}, & \text{if Rule is skip} \\
insert.log(id, f(s_1, ..., s_n), t), & \text{if Rule is } f(s_1, ..., s_n) := t \\
T(id, P) \mid T(id, Q), & \text{if Rule is } P \text{ par } Q \\
\odot(T(id, P) \otimes T(id, Q)), & \text{if Rule is } P \text{ seq } Q \\
(\varphi \otimes T(id, P)) \lor (\neg \varphi \otimes T(id, Q)), & \text{if Rule is } if \varphi \text{ then } P \text{ else } Q \\
(X = t) \otimes T(id, P), & \text{if Rule is } let \ x = t \ in \ P \\
(\varphi(X) \otimes T(id, P)) \lor \text{empty}_\varphi, & \text{if Rule is } choose \ x \ with \ \varphi \ do \ P \\
r(t_1, ..., t_n), & \text{if Rule is } r(t_1, ..., t_n) \\
\end{cases}
\]  

(2.3)

The transformation of an update rule results in the insertion of the \textit{log} relation into the database; \textit{log} is a relation between the \textit{id} of the update rule, the location of the update, \( f(s_1, ..., s_n) \), and the new value, \( t \). In this way we can keep track of the updates that different rules make.

The transformation of the block, sequence, conditional, let, and call rules is straightforward: we use the concurrent operator "\mid" for block rules, the sequential operator "\otimes" for sequence rules, and disjunction "\lor" for conditional rules. The transformation of choose rule is also straightforward, however a new predicate, \textit{empty}_\varphi, is introduced in order to check the emptiness of \( \varphi \); \textit{empty}_\varphi is defined as follows:

\[
empty_\varphi \leftarrow \neg \text{nonempty}_\varphi \\
\nonempty_\varphi \leftarrow \varphi(X)
\]  

(2.4)

The CTR equivalent for \textit{forall} \( x \ with \ \varphi \ do \ P \) in (2.3) is defined as follows:

\[
\forall_x \varphi \leftarrow [\text{new}_\varphi := \varphi] \otimes \forall_x \text{new}_\varphi \\
\forall_x \text{new}_\varphi \leftarrow (\text{new}_\varphi(X) \otimes \text{delete.new}_\varphi(X) \otimes (T(id, P) \mid \forall_x \text{new}_\varphi)) \lor \text{empty}_\text{new}_\varphi
\]  

(2.5)

In order to capture the \textit{forall} rule in CTR we use recursion. In order to find out all \( X \) satisfying \( \varphi \) we need to change the \( \varphi \) relation in the recursion. However, since \( \varphi \) can appear as well in other rules, we do not want to chance it.
To cope with this situation, in the first rule above we copy $\phi$ into a new relation $new_\phi$ which then can be change, without changing $\phi$. In the second rule above, we pick up an $X$ satisfying $new_\phi$, delete it from $new_\phi$, and then concurrently execute $P$ with recursively calling $forall_{new_\phi}$. Recursively calling $forall_{new_\phi}$ ensures that $P$ is executed for all $X$ satisfying $new_\phi$. The recursion continues until $new_\phi$ if empty; this is captured by $empty_{new_\phi}$, which is defined in a similar way as $empty_\phi$:

\[
empty_{new_\phi} \leftarrow \neg nonempty_{new_\phi}
\]

\[
nonempty_{new_\phi} \leftarrow new_\phi(X)
\]

### 2.3.2 Step 2

The second step in our transformation checks the consistency of the logs resulted from step one, and if the logs are consistent then the actual updates take places. This step (to which we refer as post-transformation, and define it via the $posttrans$ predicate below) is independent on the structure of the ASM programs. However, it relies on the definitions of two relations $belongsTo$ and $after$ which are determined based on the structure of the ASM programs. These relations are between the ids of rules and are needed when checking consistency. For example, in a $P$ seq $Q$ rule a location can be changed with different values in $P$ and $Q$, which would result in consistent updated, however if the rule is $P$ par $Q$, than the execution would result in inconsistent updates. The $after$ and relation is are used to detect such kind of inconsistencies as we shall see below. The $belongsTo$ relation is used in the definition of $after$ relation, and can be obtained from the structure of the ASM program as follows:

\[
belongsTo_{Rule} = \begin{cases} 
belongsTo_P \cup belongsTo_Q \\
\cup \{belongsTo(Id(P), Id(Rule)), belongsTo(Id(Q), Id(Rule))\}, \\
\quad \text{if Rule} = P \text{ seq } Q \\
\quad \text{or Rule} = P \text{ par } Q \\
\quad \text{or Rule} = \text{ if } \phi \text{ then } P \text{ else } Q \\
\end{cases}
\]

\[
belongsTo_{Rule} = \begin{cases} 
belongsTo_P \cup \{belongsTo(Id(P), Id(Rule))\}, \\
\quad \text{if Rule} = \text{ let } x = t \text{ in } P \\
\quad \text{or Rule} = \text{ choose } X \text{ with } \phi \text{ do } P \\
\quad \text{or Rule} = \text{ forall } X \text{ with } \phi \text{ do } P 
\end{cases}
\]

(2.7)

That is, for every sub-rule $X$ of a rule $Y$, we have a relation $belongsTo(Id(X), Id(Y))$. Furthermore, every sub-rule of $X$ also belongs to $Y$, i.e. $belongsTo$ is transitive, and we define it as follows:

\[
belongsTo(Id_1, Id_2) \leftarrow belongsTo(Id_1, Id_3) \otimes belongsTo(Id_3, Id_2)
\]

(2.8)

The $after$ relation is obtained from the structure of ASMs as follows:

\[
after_{P \text{ seq } Q} = \{after(Id(P), Id(Q))\}
\]

(2.9)
That is, for every rule $P \text{ seq } Q$, we have a relation $after(Id(P), Id(Q))$. Furthermore, if $X$ and $Y$ are in an after relation, then all the rules that belong to $X$ are in an after relation with all the rules that belong to $Y$. Thus we can generalize the definition of after as follows:

$$after(Id_1, Id_2) \leftarrow \text{belongsTo}(Id_1, Id_3) \otimes \text{belongsTo}(Id_2, Id_4) \otimes after(Id_3, Id_4)$$

(2.10)

We can now return to the definition of the post-transformation step, which is independent on the structure of the ASM program:

$$\text{post}_\text{trans} \leftarrow \neg \text{inconsistent} \otimes \text{do}_\text{updates}$$

(2.11)

In the post-transformation step we check for inconsistency and then do the actual updates. Inconsistency is defined as follows:

$$\text{inconsistent} \leftarrow \log(Id_1, X, Y_1) \otimes \log(Id_2, X, Y_2) \otimes Y_1 \neq Y_2 \otimes \neg after(Id_1, Id_2)$$

(2.12)

The rule above expresses the fact that if in the logs there exists a location with two different values set by two different rules not in an after relation, then we have inconsistent updates.

The actual updates take place recursively as follows:

$$\text{do}_\text{updates} \leftarrow (\log(\_, X, Y) \otimes \text{delete.content}(X, \_) \otimes \text{insert.content}(X, Y) \otimes \text{delete.log}(\_, X, Y) \otimes \text{do}_\text{updates})$$

$$\lor \text{empty}_\text{log}$$

(2.13)

In the rule above we recursively replace (delete and then insert) the content of the locations appearing in logs with the values we have the logs. The recursion ends when the log is empty, by checking emptiness of the empty relation which is defined in a similar way as empty:\n
$$\text{empty}_\text{log} \leftarrow \neg \text{nonempty}_\text{log}$$

$$\text{nonempty}_\text{log} \leftarrow \log(X, Y)$$

(2.14)
3 Summary and Outlook

In this deliverable we gave a possible simulation of ASMs in CTR; this shows that that CTR is a more general framework than the one provided by ASMs. This simulation can potentially be used to enable reasoning about ASMs by reusing (and extending) reasoning techniques developed for CTR.
Bibliography


