RAPID PROTOTYPING USING OBJECT-ORIENTED TERM-REWITING

Adam Steele
DePaul University CTI
Chicago, IL, 60604
asteele@cs.depaul.edu

Peter Grogono
Concordia University
Montreal, QC, H3G 1M8
grogono@cs.concordia.ca

Abstract
In this paper, we explore the theory and practice of defining the state changes that an object undergoes during the execution of a program, using a synthesis of Object-Oriented programming and traditional Term-Rewriting theory (OOTR). We discuss the practical aspects of implementing OOTR using the rule engine in an expert system, programming issues, and consider the application of this technology to real-world problems. We also discuss the lessons learned from using this technology to rapidly prototype and implement the network model for a large electrical power network.

Introduction
We are interested in reactive systems that model large-scale industrial applications such as power networks, or Air Traffic Management at a busy airport like Chicago’s O’Hare. In these kinds of systems there may be a very large number of inputs that are arriving in real-time without any predictable pattern or structure. In traditional Object-Oriented programming practice, state change is effected through the (possibly concurrent) execution of assignment statements following a rigid flow of control. The probability of unintended interactions between objects is very high, and exceedingly difficult to control. These problems require complex mechanisms for synchronization and blocking that are difficult to understand, and very prone to errors. All of this makes testing these systems complicated, and expensive.

In this paper, we discuss a programming technique that addresses this problem by adapting concepts from Term-Rewriting and applying them to Object-Oriented programming. Term-Rewriting is a field of study that grew out of equational theorem proving; and in its practical instantiation concerns the successive application of rules in order to transform a term from one form to another. Our adaptation of Term-Rewriting offers a way to model the changes in state of an object in a program that, like the reductions in a confluent Term-Rewriting system, are fundamentally asynchronous, and non-interacting. From a programming point of view, the rules we provide are simple to understand and completely characterize an object’s dynamic state changes. Furthermore, without the requirement to specify the concurrent interactions in a program, the programs we write are short and relatively easy to modify.

The theoretical developments in this paper have been motivated by practical experience with implementing these systems of Object-Oriented rewrite rules using the Rete algorithm that is at the heart of expert-system shells such as CLIPS[1], or Jess[2]. Actual experience in designing the network model for a large public utility has shown that this programming paradigm is very effective for developing prototypes that are efficient enough to serve in customer-deployed systems. The paradigm is also flexible enough to model diverse tasks such as navigating in Virtual Reality, and simulating a Freeflight Air Traffic Management system[3].

Object-Oriented Term-Rewriting
Object-Oriented Term-Rewriting (OOTR) is solidly based in traditional Term-Rewriting theory, but we will focus on critical-pairs, rather than termination issues[4, 5]. We provide first, definitions for some of the traditional Term-Rewriting concepts in the context of Object-Oriented programming, in order to lay the groundwork for our theoretical developments.

Unconditional Rules
We define objects in the standard record fashion[6], and we adapt, *grosso modo*, the standard definitions from term-rewriting theory[7]. An *unconditional object rewrite rule* is a triple \((l,r,m)\), written \(l \Rightarrow r\), where \(l\) and \(r\) are objects, and \(m\) is a possibly empty set of methods calls (which we define below); again, as in the rewrite rules for terms, we have \(\text{FV}(r) \cup \text{FV}(m) \subseteq \text{FV}(l)\), where \(\text{FV}(t)\) is the function that returns the free variables of a term \(t\). As a simple example, we have the non-terminating rule that is not part of a method definition, and has no associated methods:

\[[x=xpos,y= ypos] \Rightarrow [x= xpos+1, y= ypos+1]\]

which matches any object with fields \(x\) and \(y\), and adds one to values in those fields. It is non-terminating...
because any object which is matched by the rule, and has its fields updated; now becomes a candidate for matching again. We will see how we may handle this problem in the next section. We also show a more complicated rule that is a simplified version of one from one of the OOTR systems we discuss as an example development in a later chapter. This rule matches on any object that has a token field, and a field with collection of outgoing links. The intention of this rule is to remove the token from the object and place it on each of the outgoing links, using the collections iterator in the method distribute_token:

\begin{verbatim}
[token=tok, links=lnks] ⇒
[token=nil, power=tok]<lnks.distribute_token(tok )>
\end{verbatim}

So for this rule would match and rewrite the following object:

\begin{verbatim}
[spindle=7,token=120KV,links=[...=<GH17,LBS42>...]]
\end{verbatim}

\(→\)

\begin{verbatim}
[spindle=7,token=nil, power=120KV,
links=[...=<GH17, LBS42>...]
<...=<GH17,LBS42>...].distribute_token(120KV )
\end{verbatim}

In this rule, < and > are the generic collection constructors. One of the problems with the above rule is that it matches the updated object itself; and the rule that matched and fired on this updated object would update all objects outgoing links with nil tokens. The standard, if inelegant, way to solve this problem[7] is to order the unconditional equations, and consider potential matches in textual order, i.e. our rule set would look like:

\begin{verbatim}
[token=nil] ⇒ // a no-op
[token=tok, links=lnks] ⇒ [token=nil,
power=tok]<lnks.distribute_token(tok )>
\end{verbatim}

We can solve this problem by placing conditions on rules, which increases their expressiveness with respect to programming, but complicates the theoretical analysis.

**Conditional Rules**

We are obviously limited by the use of unconditional rules, so we introduce the notion of a condition that has to be satisfied before the rule can fire. For example, considering the set of rules above. We can replace them with the following rule:

\begin{verbatim}
[token=tok, links=lnks]: (tok ≠ nil) ⇒
[token=nil, power=tok]<lnks.distribute_token(tok )>
\end{verbatim}

More formally, we define a conditional object rewrite rule as a quadruple \((l,c,r,m)\), written \(l_1(c)⇒r_1(m)\), we have \(FV(c)∪FV(r)∪FV(m) ⊆ FV(l)\) and we impose the restriction that the condition \(c_0\), where \(σ\) is a substitution, is in ground normal form. We often write the condition \(c\) as \(c_1,c_2,...,c_n\) where , indicates conjunction; for example the following rule will match and fire on any ColorPoint (or any member of one of its subclasses) that is white, and has an even value for its \(x\) component:

\begin{verbatim}
[x=xpos,c=color]: (even(xpos)=true, color=white) ⇒
[x=(xpos div 2),c=black]
\end{verbatim}

To effectively calculate the relation \(=\) we will use the notion of joinability \(\downarrow_R\) to capture this notion of equality. It is also possible to express \(=\) by conversion, i.e. \(leftrightarrow\). the disadvantage of this definition is that it is not effective, e.g. with rules \(a⇒b\) and \(b⇒c\) we have \(a\downarrow_R c\) but if we consider conversion as a way to define equality we may have the non-terminating computation \(a⇒b⇒c⇒a⇒b...\). We have show that there are some theoretical problems with joinability as a definition for equality; however, we are able to recover an effective notion of equality, by restricting the conditions we may allow in our rules[3].

**Implementing OOTR**

**Objects and classes**

The expert system shell CLIPS has a very simple Object-Oriented sub-language called COOL (CLIPS Object-Oriented Language). We define classes with the defclass keyword and create, modify and destruct and object instance with the make-instance, modify-instance, and unmake-instance keywords, respectively. For example, we can define a class that represents tokens in an electrical power network simulation.

\begin{verbatim}
(defclass TOKEN (is-a USER)

(role concrete)
(pattern-match reactive)
(slot Type (create-accessor

(read-write)

(default nil))
\end{verbatim}

This identifies the class TOKEN which can form instances which can be matched upon, with one field Type which has the default value nil. The problem with this object is that it exists solely in the knowledgebase and any changes in the state of a real C++ (or other OOPL) object must be done by proxy with an external procedure call.

In Jess we define classes with the keyword deftemplate, and we can define the communication between Jess and Java[8] a number of ways. From Java to Jess we can create and update an object in the following manner:

\begin{verbatim}
Rete r = new Rete();
r.executeCommand("(batch jess/scriptlib.clp)"); r.executeCommand("(deftemplate point \"A 2D point\")
(slot x) (slot y))"); Fact f = new Fact("point", r);
f.setSlotValue("x", new Value(37,RU.INTEGER));
f.setSlotValue("y", new Value(49,RU.INTEGER)); r.assert(f);
\end{verbatim}
Or we can define a Java class and specify that its objects have the potential to become Jess facts by using the `defclass` keyword. We have to call the `(reset)` function in Jess in order to synchronize the external Java objects, with their internal Jess facts. Other than this the coordination between Jess and Java is fairly smooth.

**Rules**

The rules we define in OOTR can be fairly easily translated in Jess/CLIPS rules, because of their close connection. To make the translation we make use of the fact we can bind the value of any object that matches a rule to a variable that is available in the rule. For example we can take the example OOTR rule from the OOTR chapter:

\[
\begin{align*}
\text{[tok=pwrdby,lnks=nextObjs]}: (\text{pwrdby}<>\text{nil}) \Rightarrow \\
\text{[tok=\text{nil}]} <\text{nextObjs.distribute-token(pwrdby)}>
\end{align*}
\]

becomes

\[
\begin{align*}
\text{(defrule power-1} & \text{ ?self <- (object (is-a PowerObject) \)} \\
& \text{(tok ?pwrdby=\text{nil}) \)} \\
& \text{(lnks ?nextObjs))} \\
\Rightarrow & \text{(send ?self put-tok nil)} \\
& \text{(distribute-token(?nextObjs,?pwrdby)} \\
\end{align*}
\]

As one can see the translation is fairly straightforward. The trick of simulating `self` by binding any matched object is key to the successful translation from OOTR. We also note that it is possible to set priorities for rules using the `salience` parameter. This allows us to override the standard rule selection algorithms and is important for guaranteeing that certain performance commitments are made.

**Performance Considerations**

The Rete algorithm[9] is the reason that OOTR programming is potentially more efficient than a naïve implementation in a traditional imperative language like C++. Essentially, what the Rete based system offers is a data driven notification scheme similar to the Gang-of-Four (GOF) observer pattern[10]. This coupled with the lightweight fine-grained concurrency of the OOTR rules is what makes the resulting systems computationally efficient.

**OOTR and Electric Power Networks (PLN)**

The example we will discuss is that of the network model for the power distribution system for the Indonesian state power company (PLN). This work was done while the author was working for M3i Systems Inc. in 1994-1995. M3i is a Montreal based Company[11] which was a spin-off from the Provincial power utility Hydro-Québec; and its main business is supporting electrical power utilities worldwide by supplying Distribution Management Systems (DMS) and Control Rooms. The Company’s special expertise is in setting up integrated Simultaneous Control And Data Acquisition (SCADA) systems, along with the electronic control rooms, and the multiple, tiled projection displays to visualize and control these systems.

**Network Models**

The main function of a DMS is to monitor the state of a power network (grid) and allow operator intervention in the event that there are network problems. The main goal of this intervention is to restore the functioning of the network, or to smoothly degrade the networks function if it is not possible to restore its proper functioning. Another function of the DMS is to model changes to network for testing or maintenance purposes.

The network model is an exact representation of all the different components of the real power network, and serves to model the state of the current network, and to model potential actions, either for testing purposes, e.g. to determine which switches have to be opened in order to isolate a portion of the network so that repairs may take place; or to actually effect those changes in the real network. The network model of the actual network is along with the SCADA system is the heart of a DMS, and its flexibility and speed is critical for real-time control by the utility’s operators. Network models exist not only in the electrical power industry, but also in other public utilities such as Thames Water(UK), NIPSCO(USA) and PG&E(USA).

**OOTR Model of PLN Network**

In the PLN Network, power enters the system at a CB (these names are specific to the PLN network), and is carried over a LINE power line to an LBS substation, from there it exits on a LINE to a GD which is a 4-bar...
switch and then on to a Feeder which is a step-down transformer which sends power out onto the local transmission network. There were other objects such as GHSs, POLYs and PTSs that performed similar roles but were actually part of a DMS originally designed by Cégelec, a French utility company. In all, there were about 60,000 of these objects, up until this point all the network models had been coded in C.

Defining the Network Model in CLIPS

The new PLN network model was based on a network model developed for a previous system, but unfortunately there were major revisions that had to be done and performance problems that had to be addressed. The main development was to be in C++ running under OS/2. In order to rapidly prototype a simple reference model, and as a chance to test out some of the author’s early PhD work, it was decided to prototype the network model using an early version of OOTR implemented with the expert system shell CLIPS.

A simple model of a network containing about 20 items was coded up quickly and the basic idea was validated. A more complicated, but incomplete (with respect to the number of network objects), model that accurately reflected the PLN documentation was then coded, still with the intention that this would serve as a reference model in order to better understand the actual model that would be coded in C++. At this time, it was noted that the performance was fairly good, and it was decided to do some benchmarking of the OOTR network model. A simple model achieved approximately four thousand rule-firings a second, and this scaled fairly linearly with the number of objects (CLIPS was compiled by Borland C++ under OS/2 running on a Pentium 100 machine with 32Mb of RAM). The performance of these simple models compared favorably with that of the C++ prototype, and it was decided to scale the minimal network model up to the size of the production model, as noted previously has approximately 60,000 objects. One of the important feature of this approach is that the program used to specify the behavior of the network was approximately 250 line long in comparison with about 5,000 lines for the legacy C++ system.

A program was written to extract object information from the PLN database, and write out representations of these objects to an ASCII file. This file was parsed using YACC++, and C++ method calls were associated with the productions of the grammar that defined the representation of the objects in the ASCII file. These C++ methods served as a wrapper for the external CLIPS C calls that were used to build the CLIPS internal object knowledge base. External communication was achieved by using C calls from the SCADA system and the operators control system to update the state of named objects. Once the knowledge base had been updated, an external run call was made to start the rule engine. When a CLIPS rule fired on an object, passing a token representing the power to its outgoing links, actions on the right hand side of rules were signaled by external C calls that sent messages over an OS/2 pipe to the MOSAIC (display) system that was running on another computer. The MOSAIC system updated the state of the network on the tiled projection display. The CLIPS based network model, proved to be very stable with respect to performance even when new rules and objects were added to the system (the increase in processing time was almost linear for both rules and objects). After extensive testing it was decided that it would form the network model in the first release of the system that was shipped to PLN in early 1995. Interestingly, this first system release shipped on time, the first occasion this had happened for any M3i production system.

Lessons Learned

CLIPS and Jess, like LISP, are interpreted languages with fairly flexible type systems. It is this ability to rapidly turn around changes and experiment with a working program that is key to successful rapid prototyping. Furthermore, OOTR is a natural way to express many of the dynamic characteristics of complex systems like the PLN and MEB networks. It should be noted, however, that rapid prototyping is not always a successful development technique. It requires an overall system architect who has a strong feeling for where the development of the system is going, and the ability to push the development forward. Without this oversight, the spirals of the spiral development model can become so tight that process degenerates into hacking. In many cases, a modified waterfall method, with a fairly small number of phases from which partial products can be spun out can be more successful.

The other interesting result from this experience was the efficiency of the Rete algorithm in a production system. This results partly from the basic model of computation, which is data-driven, but also because the asynchrony inherent in the rewrite model of computation is very lightweight in comparison to multiprocess or multithread concurrency management.

We have also used OOTR techniques in the design of a system for navigating in VR environments[12, 13] and in the simulation of Air Traffic Management in a Freeflight environment. In these cases, OOTR has proved to be a powerful tool for defining a program, and executing it efficiently.

---

1 A model containing 5,000 objects propagated a token through the whole network in 1.2 seconds; a similar model with 10,000 objects took 2.5 seconds to complete.
Conclusion
We have taken programming solutions for real-world applications that were originally motivated by theoretical concerns, and used them to develop theoretical results that will have direct, and positive impact on the development of future applications. In future developments we would like to continue to keep this theory-practice spiral tight.

Some of the research that we feel would help further lay the theoretical and practical framework for OOTR are issues such as Quality of Service (QoS). The fact that OOTR rules can have dynamic salience (i.e. dynamic priority in the rule-firing order), would allow us to adjust the priorities of various rules; and understanding the interactions of dynamic salience, and the already existing conflict resolution strategies would allow us to guarantee that fairness conditions would continue to hold in a hard real-time environment. In general, we see OOTR as having the possibility of changing the way we specify the dynamic behavior of a program. We have shown how that we can separate the applicative and imperative features of our calculus, and made it possible to define the dynamic behavior of a program in a fashion that is robustly asynchronous. This gives us a much more declarative means of specifying the behavior of our programs – not having to deal with the overhead of managing concurrent updates means that, for example, the production program for the PLN system network model fits on four pages (250 lines). Furthermore, there are simple tests for modularity for any further rules that are added to the system, which makes the systems easy to update, and easy to tie back to the specifications. The updateability and traceability of OOTR programs makes them ideal for defining and building large programs in a piece-wise fashion, which is one of the goals of any successful Software Engineering methodology.

References
[12] http:\www.m3isystems.qc.ca