Power Network Control Using Object-Oriented Term-Rewriting

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

Abstract
In this paper, we describe the use of the expert system shell CLIPS to define the network model in the Distribution Management System (DMS) for the Indonesian State Power system (PLN). Although, the DMS is implemented by a mixed CLIPS/C++ system, the rules defining the behavior of the system are developed using the framework of Object-Oriented Term-Rewriting (OOTR) to guarantee the asynchronous characteristics of the program’s execution. Another advantage of using OOTP is that we can describe both the power propagation in the network, and the alarm conditions which have to be monitored in a uniform fashion. We discuss the performance of the system, and the general approach of using OOTR as part of a rapid-prototyping methodology for power networks and other reactive systems.

Keywords: Power Networks, Intelligent Control, Modeling and Simulation

Introduction
Power networks are a classic example of a large-scale reactive system; in these kinds of systems there may be a very large number of inputs that are arriving in real-time from a SCADA subsystem without any predictable pattern or structure. In traditional Object-Oriented programming practice, state change in the network model 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 prone to errors; which 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. We discuss the application of these techniques in the definition and implementation of a large-scale network model for the Indonesian State Power System (Perusahaan Listrik Negeri – PLN), and show how we can use these techniques to define both network behavior and alarm conditions in a uniform fashion.

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[GR93], or Jess[Jess]. Actual experience in designing the PLN network model has shown that this programming paradigm is very effective for developing network prototypes that can help define and understand the system’s behavior, but are efficient enough to serve in customer-deployed systems. The paradigm is also flexible enough to model diverse tasks such as navigating in VR environments[SBC98,SMC99], and simulating a Freeflight Air Traffic Management system[Ste99].

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[Der87, OS89]. 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\[CM89\], and we adapt, *grosso modo*, the standard definitions from term-rewriting theory\[DJ90\]. An **unconditional object rewrite rule** is a triple \((l, r, m)\), written \(l \Rightarrow r < m>\) 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 \(FV(r) \cup FV(m) \subseteq FV(l)\), where \(FV(t)\) is the function that returns the free variables of a term \(t\). Our first example of a rule is similar to the one defining the propagation of a token through a network. 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:

\[\text{[token=tok, links=lnks]} \Rightarrow \text{[token=nil, power=tok]} < \text{lnks.distribute_token(tok)}>\]

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

\[\text{[spindle=7, token=120KV, links=[\ldots=<GH17, LBS42>\ldots]}] \rightarrow \text{[spindle=7, token=nil, power=120KV, links=[\ldots=<GH17, LBS42>\ldots]} < \text{[\ldots=<GH17, LBS42>\ldots].distribute_token(120KV)}>\]

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 \(\text{nil}\) tokens. 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:

\[\text{[token=tok, links=lnks]}: (\text{tok} \neq \text{nil}) \Rightarrow \text{[token=nil, power=tok]} < \text{lnks.distribute_token(tok)} >\]

More formally, we define a **conditional object rewrite rule** as a quadruple \((l, c, r, m)\), written \(l: (c) \Rightarrow r < m>\), we have \(FV(c) \cup FV(r) \cup FV(m) \subseteq FV(l)\) and we impose the restriction that the condition \(c\sigma\), where \(\sigma\) is a substitution, is in ground normal form. We often write the condition \(c\) as \(c_1, c_2, \ldots, c_n\) where \(,\) indicates conjunction. To effectively calculate the relation = we will use the notion of joinability \(\downarrow_R\) to capture this notion of equality. We have shown 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\[Ste99\]. One of the key concepts from traditional term-rewriting theory is that of a **critical-pair**. A critical pair in OOTR occurs when a rule can match with the left-hand side of another rule; this indicates a potential rewriting ambiguity. If there are no critical pairs in a set of rules (a testable condition), then we can guarantee that rule firings will not interact, and the order of application of the rules does not matter. This allows us to add in rules to change behavior, or to detect alarm conditions, safely because we can test if there are any adverse interactions with existing rules. For example we can add in the rule:

\[\text{[token=tok], : (\text{len(tok)} \geq 2) \Rightarrow [\ldots<\text{self.error("Multiple power sources for ", self.name)}>\] \]

Which will indicate if more than one feeder powers an object, without creating any problems.

**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.

```
(defclass TOKEN (is-a USER)
  (role concrete)
  (pattern-match reactive)
  (slot Type (create-accessor read-write)
    (default nil))
```

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[GJS96] a number of ways. From Java to Jess we can create and update an object in the following manner:

```java
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);
```

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 above, and translate it into the CLIPS/Jess rule:

```
(defrule power-1
  ?self <- (object (is-a PowerObject)
     (tok ?pwrdbys=nil)
     (lnks ?nextObjs))
  )
```

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[For82] 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[GHJV95]. This coupled with the lightweight fine-grained concurrency of the OOTR rules is what makes the resulting systems computationally efficient.

**OOTR and the PLN Power Network**

The original motivation for the development of the theory of OOTR grew out of its use in defining 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[M3i] 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, 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.

**OOTR Model of the 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 features 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

---

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.
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 power network. 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.

**Conclusion**

We used OOTR define the dynamic behavior of the PLN Network in a fashion that is robustly asynchronous. This gives us a much more declarative means of specifying the behavior of our program – 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, such as the PLN network model in a piece-wise fashion, which is one of the goals of any successful Software Engineering methodology.

**References**


[GHJV95] Gamma, E., R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.


[M3i] http:\www.m3isystems.qc.ca


