A Framework for a Rule-Based Form Validation Engine

Luis Blando
Operations Systems Lab.
GTE Laboratories, Inc.
40 Sylvan Road, MS-40
Waltham, MA 02454 USA
+1 781 466 3296
lblando@mit.edu

ABSTRACT

Automatic form validation enables telecommunication carriers to process incoming service order requests more effectively. Validation rules, however, can be nontrivial to test and ultimately depend on the carrier's internal software systems. Traditionally, these validation checks are spread throughout an application's source code, which makes maintaining and evolving the system a very complex task. Our approach to solving this problem involves decoupling the rules, giving them a simple, easy-to-understand representation, and creating an engine to apply these rules to incoming forms automatically. This paper presents our approach in detail, explains its parallelism and briefly presents the differences with other common rule-based engines.

Keywords: Rule-Based systems, Form Validation. Frameworks. Parallel Programming.

1 INTRODUCTION

The OMT/SIGS project at GTE Laboratories is aimed at automating order processing. The orders are loosely called Local Service Requests (LSRs) or Local Service Orders (LSOs) [1] and may arrive into GTE's Order Centers via fax, or Electronic Data Interchange (EDI) transmissions. Whenever an LSR is received, it has to go through a series of validation checks to make sure it complies with open market standards and GTE validation[1] rules. Traditionally, the know-how about these checks is hard-coded in the application that does the checking. This is usually done in the same programming language as the application itself is written in. This approach has the disadvantage of being highly inflexible to potential change in the edit rules, spreading the 'validation knowledge' into many places (modules) in an application and possibly replicating that knowledge in many modules. Thus impedes a fast and accurate system response when these rules change.

The LSR Edit Engine, or EE hereafter, has been designed to address these problems. Its purpose is to enable easy modification of the edit rules using a high level language that end-users (i.e.. GTE business rule experts) can manipulate directly.

2 TYPES OF VALIDATION CHECKS

In order to establish a common vocabulary, it is important to define the kinds of edits the EE will handle. The four different and loosely named kinds of errors the EE will check for are:

1. **"Meta" Checks/Errors:** These checks are the first that are performed on the request. They are designed to ascertain that we have received all the information (i.e. forms) required for a particular request type and that no "prohibited" data has been sent. Only after these checks have been validated can the rest of them be started.
2. **Syntactic Checks/Errors:** These are performed at a field level. Only the field data is necessary to perform this check. For instance: "*Only numeric values allowed*" or "*First byte must be a zero*" would be examples of syntactic checks/errors.
3. **Semantic Checks/Errors:** The semantic checks involve field dependencies. The differentiating factor is basically the inability to perform the check with just the field data (i.e.. more fields are needed). For instance: "*if Field1 = A then Field2 must be present" or "if ACT = 2 then REFNUM should be one of 1,2, or 3*" would qualify as semantic checks.
4. **Domain Checks/Errors:** Domain Checks involve

[1] In the telecommunication carriers' world, these validation are sometimes referred as "edits". Consequently, This paper will use both terms interchangeably.

pieces of information that can not be derived from the document being validated alone. They are a proper superset of the semantic checks. For instance, "*check that the value of the CCNA field represents a valid GTE Customer*" or "*Validate the BILL_STATE field against GTE's State Table*" are examples of domain checks.

The bulk of the edits are standard, based on a particular version, called “issue” in the industry, of the LSOG [1] document. Other edits are company-specific and are present to support specific inter-company agreements with regards to the format of electronic orders. The EE is designed to support more that one issue of the LSOG document at a time. The EE assumes there is a superset document that encompasses both the LSOG specs plus any GTE-defined edits). The EE can accommodate multiple versions of these documents simultaneously.

3 LOGICAL FLOW OF A VALIDATION

For this discussion, we will assume a strictly sequential thread of control. It will be expanded later to parallelize many of the checks. We will also assume, as is indeed the case, that the incoming request is already stored in a common data store for retrieval.

The following sequence of steps needs to be taken to validate any given request:

1. The LSO type (request type) needs to be determined first, so that we can tell what are the components (forms) that are required, prohibited, or optional. As hinted in the previous paragraphs, a couple of fields in the LSR form act as metadata for the request itself. The two fields are: "request type" (`lsr.reqtyp`) and "activity code" (`lsr.act`). A given combination of `lsr.reqtyp/lsr.act` uniquely determines the LSO type. If the combination found is not valid, an exception is immediately raised and processing stops.
2. Once the LSO type has been validated, a set of special rules is searched to find the one that applies to this particular request type. These rules are called "metarules" and they determine two items:
   (a) The forms that are required, prohibited, or optional. The validation engine thus needs to make sure that the "required" forms are indeed present in the request, the "prohibited" are not, etc. Should this condition not be met, an exception is raised and processing stops.
   (b) Once we have determined that this is a valid LSO type and is a "complete" request, the meta checks phase is complete and we are ready to check each of the forms. Each metarule thus contains a list of sets of rules that need to be validated. These "lists of rules" are called rulesets and there's usually one ruleset per form (though this restriction is for convenience only).
3. At this stage, we have already finished the meta checks phase and we have identified a set of rulesets that need to be checked. Thus a big loop begins. For each ruleset, the following sequence takes place:
   (a) Check each and every rule in this ruleset. If the rule checks out OK, do nothing. If it fails, collect its error code.
   (b) Regardless of whether the previous rule failed or not, go to the next one until you exhaust this ruleset.
   (c) Once all the rulesets have been checked, collect all the error codes found and report them.

One very important point to note is that, while checks (1) and (2a) are "halting" checks, in the sense that their failure signals the end of processing for this particular request, the rest are non-fatal checks that allow processing to continue and, furthermore, they must all be carried out. This rationale follows logically from two business requirements:

(a) Do not attempt to validate an incomplete or malformed request.

(b) Check all the rules on a well-formed request, so that we find everything that's wrong at once.

4 THE EDIT ENGINE LANGUAGE (EEL)

In order to construct the validation rules we designed a very simple high level language so that it can be used directly by the business analyst that is creating or modifying the rules. The EEL is tailored for doing field validations. Instead of presenting here a full version of a BNF grammar for it, the following shows an example of an actual validation file.

```
issue "1" 1 {
  meta {
    E01METH00101:
     if ( in(lsr.reqtyp,"EB","MB") and
          in(lsr.act,"N","T","V") )
     then ( required(lsr) and required(eu) and
       required(rsl) and prohibited(lp) and
       prohibited(port) and prohibited(inp) and
       prohibited(lpnp) and required(dsr) )
     do ( cross_form_set, lsr_set, eu_set,
```

```
        rsl_set, dsr_set, dl_set);
   }
   // -----------------CROSS_FORM SET
   set cross_form_set {
      E01METH00001:
       if ( in(lsr.reqtyp,"EB","MB") and
            in(lsr.act,"N","T","V","A") )
       then ( (lsr.pon == eu.pon) and
              (lsr.pon == rsl.pon) and
              (lsr.pon == dsr.pon) );
      E01METH00002:
       if (in(lsr.reqtyp,"EB","MB") and
           in(lsr.act,"C","M","R","B","D","S"))
       then ( (lsr.pon == eu.pon) and
              (lsr.pon == rsl.pon) );
   }
   // ---------------------- LSR SET
   set lsr_set {
    p E01FLDH00003: if (notempty(lsr.ccna))
       then (externalvalidate(tbhit,lsr.ccna));
     E01FLDH00012: always (required(lsr.sc));
     E01FLDH00013: if (notempty(lsr.sc[1..2]))
                   then (lsr.sc[1..2] == "GT");
     E01FLDH00014:always(required(lsr.dtsent));
     E01FLDH00015:
       if (notempty(lsr.dtsent[1..8]))
       then (isdate(lsr.dtsent[1..8]));
     E01FLDH00016:
       if ( notempty (lsr.dtsent[9..10]  ) )
       then ( (lsr.dtsent[9..10] >= "00") and
              (lsr.dtsent[9..10] <= "23") );
   }
}
```

Each validation file contains one or more `issue{...}` blocks. Each version of the LSOG document is fully-specified inside an this block. For each issue, first a `meta{...}` set is presented and then a number of rulesets (`set foo{...}`) are described. Within a set, a number of rules are described. A rule contains the following:

- An optional tag to denote this rule is spawnable on its own [`p`]
- The rule name (this will be transformed to a class name later) followed by a '`:`'
- A command (either `if (expr) then (expr)`, or `always (expr)`)
- In the case of metarules, a `do( <set_list>)`

Metarules differ from simple rules only in that they contain a `do( <ruleset_list> )` statement at the end. Furthermore, metarules are always "`if`" commands.

Expressions can have a number of representations. There are several 'basic functions', such as `in(...)`, `notin(...)`, `length(...)`, `required(...)`, `isdate(...)`, `istime(...)`, etc, that make rule-writing simple. Data elements (i.e. form fields) are represented by a qualified pathname with the pattern: `form[.form][.field]`. Parts of fields are extracted using an index notation `[a]` or `[a..b]`, where a is the index of the first character and b the index of the last character to be taken into account (starting from one).

The EEL file is then compiled into a number of C++ source files, which are themselves compiled and linked in with the system. The compiler was designed and built using a novel technique, called Adaptive Object Oriented Programming, using Demeter/Java [5]. Details about this are beyond the scope of this paper, but the reader is encouraged to get more information from http://www.ccs.neu.edu/home/lblando/misc/eelc.html.

5 EDIT ENGINE IMPLEMENTATION

This section presents an overview of the different live objects in the system and their behavior when processing a form. Figure 1 shows a runtime snapshot of the EE framework. Note that this is not a standard object graph (for one, it has inheritance links in it!)

First, the `Controller/LSRController` object is responsible for the communication with the outside system. The `Controller` part (at the framework level) takes care of providing the behavior logic while the `LSRController` subclass handles the specifics of the communication mechanism selected (a proprietary mechanism called TONICS IPC [7] in this case). The `Controller` object contains an inbound queue, an outbound queue, and an auxiliary queue for receiving BESi responses. BESi stands for Back-End System Interface, and attempts to encapsulate any third-party system that we interface with to carry out a validation.

Notice that the existence of this last queue (`besQ`) at the framework level is debatable. Our rationale was that, no matter which 'instantiation' of the framework we are dealing with, domain checks always need to communicate with a backend system (and thus receive the response). The connection between the rule object and the Workflow Manager (WFM) cloud depicts one problem with this design. The rule objects (generated from the EEL source) need at some point to contact back-end systems for domain checks. The sensible thing would have been to rely in an abstract framework-level interface for messaging or, better yet, standardize on a distributed object architecture and thus remote-enable the rule objects to send IIOP requests themselves. However, our proprietary middleware selection forced us into this design, since there's basically only one entry point for all messages to the process -the `besQ`-.

Therefore, we decided to somewhat pollute the architecture by allowing the rule objects to directly use the proprietary middleware layer while the response comes back from the `besQ`. This is clearly a compromise approach but it will enable us to move to a true distributed objects architecture (ie. CORBA) in the future, since the model that the Rule objects follow is logically equivalent to that of a synchronous remote invocation in CORBA (namely, send request, wait, and automatically be awaken).

The `Controller` object (there's only one in the system) maintains a table with "issue numbers" on one axis and "checkers active" on the other. Or, there's a list of `Checker` objects per issue. This table is used by the `Controller` object to determine if it has a `Checker` object available to handle an incoming request or if the latter has to wait.

The relationship between requests and `Checker` objects is very simple. Each request is given to an (appropriate) `Checker` object and the object works on the request until it finishes. Once the `Checker` object is done with a request it puts the result back in the outbound queue and signals the `Controller`. It relinquishes control to the `LSRController`, which in turn gives the message (after appropriate marshaling, of course) to TONICS IPC to give back to the WFM.

The `Checker`, in turn, contains a number of `RuleSet` objects. One of them is the "`meta`" `RuleSet` object. Remember the meta checks need to be done first? Well, the `Checker` object accomplishes exactly this using the `RuleSet` object. In other words, it relinquishes control to the `meta RuleSet` and waits until it is finished.

After one metarule validates and we get a list of rulesets to work on, the `Checker` signals all these rulesets to start working and these do so in parallel. Thus, paralellism is first encountered at this point for any given request. Note that this is the first place we could (reasonably) have parallelism, since the semantics of metachecks is strictly sequential and, furthermore, ordered.

Each `RuleSet` object has been given the "`Validate()`" order. All these objects have a list of `Rule` objects. They go through the list, sequentially, telling each rule object to validate itself and collecting the errors in a bag if they fail. Once they reach the end of the rules list, they signal so to the `Checker` and go back to sleep until the next order comes along.

This simple minded scenario is disrupted by parallel rules. Initial analysis resulted in the realization that the biggest bottleneck in the system would come from the IO delays in contacting and, more importantly, waiting for the results of, the back end systems. Therefore, we introduced another level of parallelism in our design, that of "spawnable rules". As the name implies, an spawnable rule is a rule that gets its own thread [4]. The thread is created the moment the `RuleSet` comes across a rule that returns `true` to the `Spawn()` invocation. At that time a new thread is created and the rule is left alone to do its work. Once the rule has finished the thread dies. Notice that this is a departure from the EE model of having the threads alive 'sleeping' until work comes their way. We selected this on-demand approach to reduce the complexity of the system, as well as the total thread load on the operating system.

Given the above loop of control, it is imperative that the `RuleSet` encounters the spawnable rules first in the set so that it spawns them as soon as possible and then goes on to work on the 'regular' rules. While it would be relatively simple to have the `RuleSet` order the rules according to their "spawnability", we have not implemented that functionality yet. Therefore, we request that spawnable rules be put first in the ruleset in the EEL file. (This optimization can also be introduced in the EEL to C++ mapping phase.)

The atomic check is performed by a `Rule` object. We have one object per check. The EEL to C++ compiler generates a class per edit that subclasses the `Rule` abstract class. In other words, for each EEL rule, a new class name is generated that subclasses `Rule`. Each `Rule` object implements a handful of methods that perform the required checks (notice that there are many auxiliary methods found at the `Rule` framework class that these subclasses use). The fact that we have one class name per EEL edit rule does not mean that we will have only one instance of that class. We might have many, for example, when we have multiple `Checker` objects that handle the same issue number. To clarify a little bit, the following EEL rule:

```
R2: always in(lsr.a, "A", "B", "C");
```

mutates into the following class:

```
#include <rules.h>
class R2 : public Rule {
 public:
   R2();
   virtual bool Validate(Form*);
 private:
   static vector<string> invar_0;
};
```

The rules `met001`, `met002`, `fld01`, `fld02`, etc in Figure 1 are examples of these classes. The validation checks

are complicated because of the existence of "repeating fields". Repeating fields are vectors of values of the same kind. For example, the field `rsl.sd.*.lna` means that the `RSL` (Resale) form contains a section called `SD` (Section Details) that contains zero or more `LNA` fields. The above field name references each and every one of these `lna` fields. If we want to address a specific row, we would say for instance `lsr.sd.#1.lna` to denote the 2nd row (indices are zero-based). Field "nesting" is supported to any level at the EE layer. It is important to note that the semantics of the validation process do not allow for any inter-row checks (ie. "`rsl.sd.#<i>.lna` must be the same as `rsl.sd.#<i+1>.lna`"). Furthermore, a rule that uses the wildcard needs to hold true for all the rows in the repeating field. Lastly, when two distinct repeating fields are part of the same EEL rule, the number of rows for both at each repeating level must be the same. To recap, the rules are:

1. No variable inter-row validations allowed. Specific checks (ie. the first row must be equal to the second) are however permitted. The only values allowed to denote an index within a repeating field are either a `*`, meaning "do this check for each and every one of the rows at this repeating level", or a '`#n`', meaning "check the row number n", which should be 0-based. Examples:

`always in(rsl.sd.*.lna,"A","B","C");`	Ok
`always(rsl.sd.i.lna==rsl.sd.(i+1).lna);`	Error
`always(rsl.sd.#0.lna==rsl.sd.#1.lna);`	Ok

2. No proper subsets of repeating fields can be specified. In other words, either exactly one instance (row) is specified (by using the `#n` notation) or all the instances are (by using the `*` notation). This means we cannot specify checks like "the first five rows must be empty". Instead, we have to say the same thing by enumerating all the first five rows using the `#n` notation.

3. The number of instances (row) at each level of nesting must be exactly the same if two distinct fields are present in a rule. For instance, the rule

   ```
   always port.sd.*.fd.*.ftr==rsl.sd.*.fd.*.ftr;
   ```

   implies that the number of `port.sd` sections must be the same as the number of `rsl.sd` sections. Furthermore, it means that, for each `port/rsl.sd` section, the number of `fd` instances must be same for both. Please note that the behavior is unspecified if this condition is not met (it is inefficient to do bounds-checking for every rule).

Last, but certainly not least, we have neglected to specify the means by which each of these rule objects access the data they need to validate. This is accomplished by the `Form/LSRForm` object (api layer and instantiation, respectively) which in turn delegate to the LSO database object. The LSO database object is a third-party product (as far as the EE is concerned) that provides database independence. It features a rich API that clients use to retrieve and store data.

6 PARALLELISM WITHIN THE EDIT ENGINE

This section will summarize the parallelism that is present within this application. In all cases, please refer to Figure 1.

Server Behavior:
The `Controller/LSRController` object behaves like a standard server in a client/server situation, with thread-pooling instead of thread-spawning. There are three separate threads within that object that control the message queues, the timer, and the response queues, respectively. Synchronization primitives are needed since all these three threads alter the state of the same objects. There's master/slave style synchronization between the `Controller` and any given `Checker` object.

"fork() paralellism":
The `Controller` object contains many `Checker` objects. Each `Checker` object runs in its own thread. As soon as the `Controller` receives a request, it gives control to the `Checker` object who does not return it until it is done. Since there's little or not synchronization necessary between the `Checker` and the `Controller` objects, and that each `Checker` works on an entire request, we've termed this *fork()* parallelism because it reminds us of the standard daemon/server situation where a new process is fork()ed the moment a new connection is received. Please bear in mind that our situation is not exactly the same, as we do not spawn a new thread for a `Checker` object, but rather the threads are pre-allocated (and sleeping). It is at this level where the switch from SMP to Dist. Memory should be made. In other words, we could have each of these `Checker` objects in a different computer (communicating through, for example, CORBA). There's barrier-style synchronization between the `Checker` object and the many `RuleSet` objects

RuleSet parallelism:
This level of parallelism was added to exploit multiple processors on the same machine. It follows from the semantics of the application that checks at the `RuleSet` level are independent and thus can proceed in parallel. Therefore, for the case when there's only one request in the queue we need to make sure we're utilizing all the processors and thus the need for this level of parallelism. Again, all these threads are pre-allocated, dormant, waiting for a signal to be waken up. There's master/slave-style synchronization between the `RuleSet` and the (maybe) spawned `Rule` threads.

On-Demand parallelism:
At the last level of our hierarchy we have the Rule objects. Since these rules are mostly CPU-bound and we have more than likely already exhausted the number of available processors with the threads used by the `RuleSet` parallelism, it didn't make sense to make each `Rule` have its own thread. The overhead would be enormous. However, there are certain rules for which we must allocate a new thread. These rules take an inordinate amount of time because they need to contact other back-end systems and thus are IO-bound. For these rules, we spawn a thread when we reach the rule. That thread acts as a client to a server connection and ultimately dies after returning the result to the `RuleSet` thread that spawned it. There's no synchronization (in general) among sibling `Rule` objects. Bear in mind that rules are performing mostly read-only operations and thus they can proceed in parallel.

7 TYPIFICATION OF PARALLELISM

In this section, we explore two available alternatives for parallel programming, and contrast their approaches with that of the EE.

CILK
We first consider MIT's Cilk. In their own words [3]:

> *Cilk is an algorithmic multithreaded language. The philosophy behind Cilk is that a programmer should concentrate on structuring his program to expose parallelism and exploit locality, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Thus, the Cilk runtime system takes care of details like load balancing, paging, and communication protocols. Unlike other multithreaded languages, however, Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance.*

`Cilk` introduces the concept of DAG-consistency. Since it works off a directed acyclic graph when scheduling work among different threads. From our application, we clearly fit in the class of problems they can handle. In other words, we need not worry about memory consistency issues (except in a few places, such as the different threads within the `Controller` object. In those, we could simply use synchronization primitives). The first benefit of using the `Cilk` approach is simplicity in the code itself. Even though we still need many synchronization primitives, they are better localized (ie. At the same 'parallelism level') and thus will not be as hard to implement or as costly to execute. Secondly, we have gained distributability by virtue of using `Cilk`. That is to say, we can move to a distributed memory parallel situation without changing a line of code. Of course, the practical feasibility of such an approach remains an unanswered question.

Last, but certainly not least, we've gained automatic load balancing and true thread pooling. This is a benefit that cannot be overlooked. Since each `Cilk spawn()` doesn't necessarily create a new thread, but rather works off a base thread pool, we have true pooling. More importantly, still, is the fact that `Cilk`'s work-stealing algorithm is perfectly suited for this kind of application.

The case when we have a very long `RuleSet` coupled with many small ones would be handled perfectly by `Cilk`. While one of the processors is working on the tail of the long `RuleSet`, the other CPUs will pick the heads of the stack (i.e. a whole small `RuleSet` at a time) and run with it. This in effect implies true load balancing and true thread pooling.

TOP-C
`TOP-C` is a parallel programming language, based on C, that follows the master/slave paradigm. In the words of its author [2],

> *The system is based on two key concepts: tasks in the context of a master/slave environment; and global, shared environment with lazy updates. Task descriptions (inputs) are generated on the master, and assigned to a slave. The slave executes the task and returns the result to the master. The master may update its own private data based on the result, or it may update data on all processes. Such global updates take place on each slave after the slave completes its current task. A SPMD (Single Program Multiple Data) style of programming is encouraged.*

Following the master/slave programming paradigm, there is one first-class process in `TOP-C`, and a bunch of second-class ones. This type of processing is perfect for writing simple communication servers, since you might have the master be the main thread of control, that `listen()`s on the socket and 'spawns' slaves once a connection is established.

TOP-C is also very simple to use. It introduces only a handful of primitives. The control of the flow is fixed. The master uses `get_task()` to find out what the next work item is, and it then calls `do_task()` in the servers. The master then uses the `get_task_result()` to retrieve responses. The `update_environment()` call is used to make changes to the global state that is replicated in all the servers. This is the only way the programmer has to change the environment in order to guarantee correct execution.

The TOP-C model seems not to fit our application problem very well. We have several levels of parallelism in the EE. Since TOP-C seems to have only two possible levels, we are at a loss when we need to extend this model. One possibility would be to have "nested tasks", in which each `do_task()` is actually a master to some other set of sub slaves.

8 FUTURE WORK

There are a number of problems with the model described in the previous sections. While the model does offer a number of benefits, its implementation makes it difficult to partition the system into pieces that can be distributed. More strictly, the application can be easily partitioned at the `Checker` level. However, the synchronization code between the `Controller` and the `Checker`(s) will need to be rewritten.

Another problem is that we are doing static load-balancing among threads. In other words, since each `RuleSet` has one thread, we are betting on the sets begin of more or less equal size. However, if we have one huge set and a number of smaller ones, we might end up with many processors idling waiting for the large set. A more dynamic method of load-balancing is necessary. Doing true thread-pooling, where the work each set needs to do is put into a common queue, and the threads pick the work from there might be a good alternative.

9 COMPARISON TO OTHER ALTERNATIVES

With the help of one of the providers of rule-engine toolkits, we conducted an extensive study to evaluate the feasibility of using a third-party rule engine, based in the RETE [3] algorithm, instead of ours. We jointly concluded that our problem domain does not fit the target problems for which a RETE-based solution is viable. Space considerations prevent us from extending on this topic. However, you can find more information at http://www.ccs.neu.edu/home/lblando/ilog.doc.

10 CONCLUSIONS

We have presented a novel approach to form validation in detail. We have briefly considered alternative approaches for parallel programming as well as using a different algorithm to perform the validation. The EE has been in operation for more than a year now with excellent results. The framework has proven reusable and adaptable, having been modified several times. This papers aims at presenting our experiences in building a successful real-world rule-based application.

11 ACKNOWLEDGEMENTS

I would like to thank Kevin Qian of GTE for his initial encouragement to write this paper, Karl Lieberherr of Northeastern University for his support with Demeter/Java, and Christie Labomme of GTE for her help in writing this paper. Finally, I would like to thank Tony Confrey, Mohammad Azzam, and Shadman Zafar of GTE for their support of the Edit Engine project.

12 REFERENCES

[1] Bellcore. LSOG Specification. Bellcore Laboratories, 1996. http:www.bellcore.com

[2] Cooperman, G. TOP-C: A Task-Oriented Parallel C Interface. *5-th International Symposium on High Performance Distributed Computing (HPDC-5)*, IEEE Press, 1996, pp. 141--150

[3] Forgy, Charles. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*,19, pp 17-37, 1982

[4] Lewis, Bil, Berg, Daniel. Multithreaded Programming with Pthreads. *Sun Microsystems Press*, 1996.

[5] Lieberherr, Karl. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, 1996.

[6] Randall, Keith. Cilk: Efficient Multithreaded Computing. *PhD Thesis*. MIT Dept. of Electrical Engineering and Computer Science, June 1998.

[7] Shukla, R., McCann, J. TOSS: TONICS for Operation Support Systems: System Management Using the World Wide Web and Intelligent Software Agents. *NOMS '98, Proceedings of the IEEE Network Operations and Management Symposium, IEEE*, New York, NY, USA, 1998, Vol. 1, p. 100-109

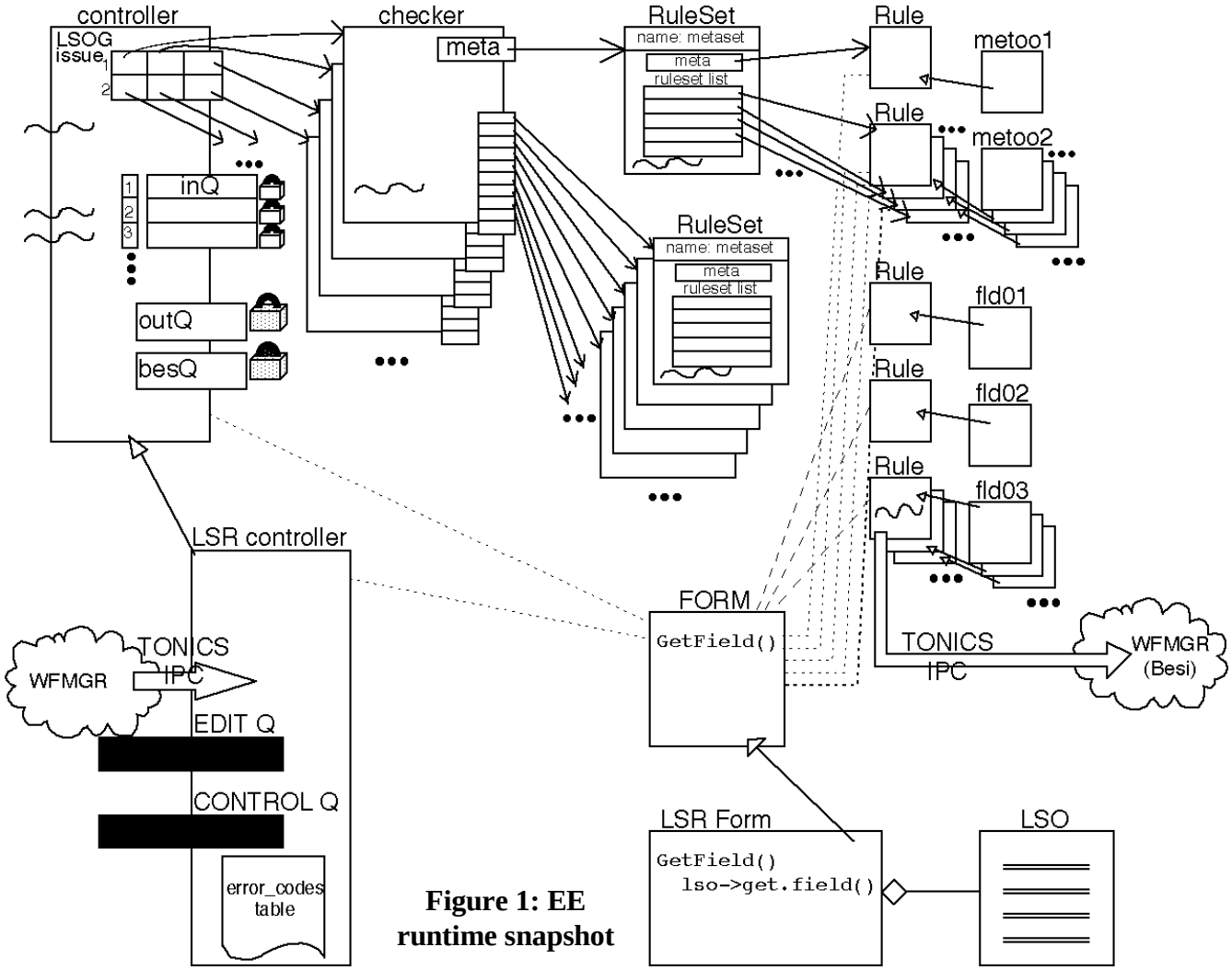

Figure 1: EE runtime snapshot

Notes:

- This picture is a mixture of an object diagram and a class diagram. It shows the different runtime structures at work within the EE.
- The classes Controller, Form, and Rule are abstract template-method classes, used to adapt the EE framework to the specific application. Thus, in this implementation the classes LSRController, LSRForm, and {met001, met002…, fld01, fld02…} extend the abstract classes.
- All the metXXX and fldXX classes are automatically generated from the high-level rule specification by a custom-built compiler.
- The WFMGR process is in charge of executing the Domain Check rules. A Domain Check rule communicates with the WFMGR process via TONICS IPC, a proprietary middleware protocol.
- The Form abstraction enables the framework to be adapted to other data sources. So far it has been modified to retrieve the information via IIOP/CORBA. The LSRForm class in the picture, however, acts as a proxy for getting the information from an Informix database (LSO).