CHAPTER 2

Review Of The CiviC Framework

This chapter will present the findings of a source code analysis of the C-based framework. The framework is first described on a high-level (section 2.1) and from a users perspective (section 2.2). The chapter concludes by describing the implementations of it's four main usable components, which are: the abstract syntax tree and it's generation (section 2.3), traversals (section 2.4), phases (section 2.5) and validation (section 2.6).

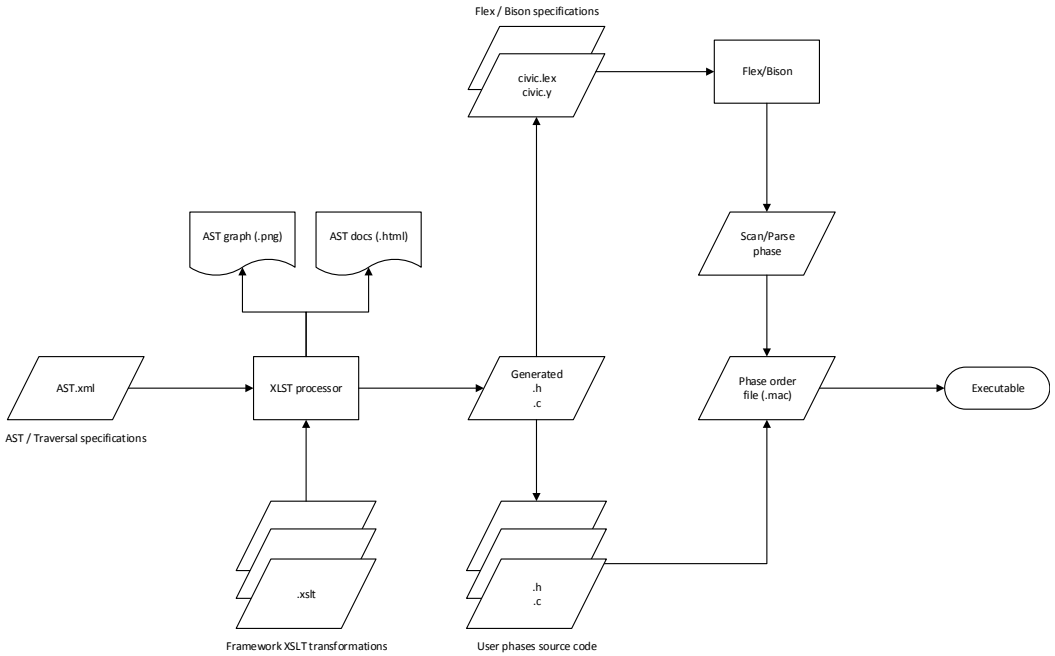

Figure 2.1: *Schematic overview of the CiviC framework*

2.1 Architecture

On a high level the framework aims to provide a structured approach for phase-based transformations on a language agnostic *abstract syntax tree* or AST. Central to the framework is a specification of the AST and a code generator which produces interaction code and traversal control mechanisms. Interaction code here means an interface through which nodes can be created or modified and assembled into a tree. The framework recognizes the distinct compilation *phases*, such as semantic analysis or code generation, and presents them as an ordered set of *sub-phases* or traversals. By piping the output of a sub-phase into the next one, incremental AST transformation is achieved. Each traversal can be configured to target any set of nodes, although handlers are always defined on the node level; one cannot target 'all expressions' for instance.

The framework is written in C and XSLT and runs on most *nix distributions, provided they support the required packages or suitable replacements[1]. A schematic overview of the architecture is presented in figure 2.1. The framework relies on an XSLT processor to transform a specification of the AST and traversals into C header and source files which can be used by the rest of the compiler. This process is triggered on each build and replaces key files if the specifications are altered.

2.2 Workflow

Changes to the AST or the addition of a new traversal are propagated by building the compiler. The build process produces updated integration code as well as documentation in two forms: 1) as an HTML document detailing every AST node, it's attributes and valid children, and 2) as an image depicting the AST graphically, with lines connecting nodes where applicable.

Debugging is assisted through command line arguments which can specify to only compile up to a certain point. The command line also allows configuration of tracer options, which - assuming the user supplemented their code with the required statements - enables detailed output of the call stack.

Important locations of user code:

1. Specification of AST and traversals (`AST.xml`)
2. Traversal header and implementation files (user directory with `*.c, *.h` files)
3. Pipeline configuration file (`phase.mac`)

A default scanner/parser phase is provided, using the Flex scanner generator and Bison parser generator. Though not formally part of the framework, they perform adequate and won't be replaced, adding to the above list:

4. Lex configuration for Flex (`civic.l`)
5. YACC configuration for Bison (`civic.y`)

[1] gcc, gzip, flex, bison, xsltproc, dot, indent

2.3 Abstract Syntax Tree

Nodes in are specified through an `<node>` XML element under the `<syntaxtree>` inside the `AST.xml` specification file. Every node has a name and two optional lists for named child nodes (or *sons*) and *attributes*, following the template shown in listing 2.1. The difference between an attribute and a child node is that child nodes can be traversed, and always point to node references, whereas attributes are simple variables or references which are never automatically traversed. The `<targets>` element inside either a child or attribute element used to specify validation attributes and will be described in section 2.6.

Listing 2.1: *Traversal specification template*

```
<node name="[node_name]">
    <sons>
        <son name="[child]">
            <targets>
                <target mandatory="[yes/no]">
                    <node name="[child_name]"/>
                    <phases><all /></phases>
                </target>
            </targets>
        </son>
    </sons>
    <attributes>
        <attribute name="[attr]">
            <type name="[attr_type]">
                <targets>
                    <target mandatory="[yes/no]">
                        <phases><all /></phases>
                    </target>
                </targets>
            </type>
        </attribute>
    </attributes>
</node>
```

In C, a node is represented by the `NODE` structure, holding the type and references to two union structures representing child nodes and attributes. Node type is represented as an enumeration of node identifiers generated each build, similarly for the child node and attribute union structures. Access to properties requires selecting the correct union, which is cumbersome and error-prone. To alleviate this *access macros* are generated for every property on every node which can be used to get or set an attribute or child node. Due to their use on both the left- and right-hand side, they cannot ensure they apply to the node passed in[2].

Overall this representation resembles that of a (simulated) tagged union, though it's strength is reduced by failing to provide access safety. Tagged unions in languages with no compiler support rely entirely on defensive programming, and this approach cannot guarantee it fails when an incompatible macro is used on a node. Conversely, adding get/set identifiers to every macro isn't a great idea either as it doubles the amount of macros, and hinders readability.

Creating nodes is a straightforward process as constructors are generated from the specification. These accepts child node references as well as any property marked mandatory. Releasing nodes is a recursive operation which requires a traversal and is covered in the next section. An example of node construction and manipulation through macros is given in listing 2.2.

[2] Assignment to conditionals is illegal in C

Listing 2.2: *Typical operations demonstrating generated access macros and constructors*

```
/* Get 'to_type' attribute on cast node of type CAST. */
if (CAST_TO_TYPE(cast) == TYPE_void) {

    /* Access common attribute through NODE. */
    CTIerrorLine(NODE_LINE(cast), "Illegal cast (void)");

    /* Access and return child property 'to' */
    return CAST_TO(cast);
}

...

/* Construct binop expr '5 < 1' */
node *binOp = TBmakeBinOp(BO_lt, TBmakeInt(5), TBmakeInt(1));
```

2.4 Traversals

Traversals are defined through both code and specification. A new traversal is specified by adding a `<traversal>` element to the set of traversals inside `AST.xml`. It requires a unique prefix, name used in debugging, mode of operation and the path to the accompanying header file. This header file contains declarations of node transformation functions, also referred to as *handlers*. A traversal must define a handler for each node type it wants to process.

Listing 2.3: *Example of a handlers and initation of traversal*

```
node *TCKmonop(node *monop, info *info)
{
    MONOP_EXPR(monop) = TRAVdo(MONOP_EXPR(monop), info);
    return monop;
}

node *TCKvar(node *var, info *info)
{
    info->last_type = type_from_declaration(VAR_DEC(var));
    return var;
}

node *TCKdoCheck(node *node, info *info)
{
    info *info = MakeInfo();

    /* Announce traversal, start and */
    TRAVpush(TR_tck);
    node = TRAVdo(node, info);
    TRAVpop();

    info = FreeInfo(info);
}
```

The operation mode signals the traversal mechanism which types are required. Two operation modes are supported: 1) *child mode*, and 2) *user mode*. In *child mode*, a list of types supported by the traversal must be supplied along with the specification. Nodes not handled by the traversal are automatically traversed until a handler on the traversal is found. By contrast, *user mode* assumes the traversal has declared handlers *for every node type* and aborts the build process if a handler is missing. This is useful in eliminating type repetition for catch-all functions or as a safety mechanism against forgotten node types.

Handlers are uniquely identified by concatenating the traversal's unique prefix with the type of the node it handlers. A handler function takes two arguments: a reference to the current node

and a state discussed later. The function returns a reference to the node which replaces the input node. This allows node removal by returning nil, property updating when returning the input and node replacement when returning any other node.

Traversals are started from a special handler: the *entry point*. The entry point is responsible for the preparation, initiation and cleanup of a traversal. If a state is to be used it must be initiated now with the static `MakeInfo()` function. A traversal is initiated by pushing it's identifier onto the traversal stack and calling the `TRAVdo` function with the root node and optional state. This hands control to the traversal mechanism which walks the tree and invokes applicable handlers along the way, eventually returning an output node. The function performs a lookup based on the active id and node type to find a suitable handler. In child mode, types without handler are routed to a generated catch-all function which uses a large switch statement to determine which child nodes should be called.

After control is returned to the entry point, it may decide to run the traversal again or pop the stack and perform a cleanup. Traversals may be nested by pushing an identifier inside while still active, though care must be taken as not all traversals support this kind of initiation. Specifically, this is only possible if no state structure is used, as the allocation and de-allocation is always static to the owning traversal. Invoking a nested traversal should be done by calling a separate entry point with accompanying parameters.

Generated Traversals Three traversals are generated on every build:

1. Copy traversal, which creates a deep copy of the structure (child nodes) and a shallow copy of all attributes
2. Free traversal, which releases all nodes in a tree and it's attributes
3. Check traversal, responsible for asserting tree validity based on checks discussed in section 2.6

Attributes are copied and freed based on their specification in the previously mentioned `<ast-types>` element, which identifies three situations: 1) node references, 2) strings and 3) value types and other references. Node references as attributes are not followed and only the reference is copied, freeing a node attribute only nulls the reference but leaves the structure intact. Strings are copied and freed using string library functions, and all other value and reference types are copied using straight C assignment. The check traversal will be discussed in section 2.6.

2.5 Phases

Phases are simple yet powerful constructs, providing a logical grouping of sub-phases which are to be sequentially executed. A sub-phase is essentially a reference to a traversal entry point and accompanied by a description only shown during debugging. For all intents and purposes sub-phases are synonymous with traversals.

The pipeline is configured with helper macros in the special `phase.mac` file, which when resolved by the preprocessor on compilation automatically feed the result of a traversal into the next. Debugging and conditional control flow hooks are also wired up at this stage.

2.6 Tree Validation

As explained in section section 2.3 all nodes share a structure, meaning that any tree or node structure is valid in the eyes of GCC. Integrity checks are supported by the framework through additional node specification and the check traversal from section 2.4.

This generated traversal validates all constraints set in the AST specification and aborts compilation upon discovering an invalid tree. Constraints can be set in the `<targets>` element optionally placed inside a child declaration or type element of an attribute (see listing 2.1). Both child node and attribute properties support the *mandatory* constraint, which invalidates the tree if the property is found to be empty.

While a type is always required on an attribute, child node properties allow any node by default unless specified further. The `<node name="..."/>` element signals the check traversal that there is only one node type valid for the child position. Often this is not sufficient as a child node may be any one from a set of nodes. Expressions are a great example of this; a binary operation node may hold two variable nodes, a ternary node, perhaps a nested binary node, etc. To describe this relation the framework uses *nodesets*, defined as elements under `<nodesets>` in the AST specification. They consist of a listing of nodes and must have a unique name, which may be referenced with the `<set name="..."/>` element.

Some properties may be empty until a certain phase and become required after that. This is supported by allowing multiple `<target>` elements on properties, which allows the specification of a begin and end range. An example might be a variable reference node with a linked symbol only required after context analysis (listing 2.4).

Listing 2.4: *Example of attribute with a range specifier*

```
<attribute name="Symbol">
    <type name="Symbol">
        <targets>
            <target mandatory="yes">
                <phases>
                    <range from="ctx_analysis" to="optimization" />
                </phases>
            </target>
        </targets>
    </type>
</attribute>
```