Generating Valid Interface Definition Language from Succinct Models

Harold Owens II
Indiana University-Purdue University Indianapolis
Dept. of Computer and Information Science
Indianapolis, IN USA
Email: owensh@cs.iupui.edu

James H. Hill
Indiana University-Purdue University Indianapolis
Dept. of Computer and Information Science
Indianapolis, IN USA
Email: hillj@cs.iupui.edu

***Abstract*—Source code generation from models (*e.g.*, domain-specific models) for distributed real-time and embedded (DRE) systems is intended to alleviate tedious, error-prone, and time-consume tasks associated with manually hand-crafting the same code. When generating code from models for DRE system programming languages that accidentally support circular dependencies, *e.g.*, the Interface Definition Language (IDL) and C++, it is necessary to resolve circular dependencies in order to generate valid and usable code. Moreover, it is important to do some automatically instead of requiring modelers to construct models that do not contain *any* circular dependencies, which is *hard*.**

This paper provides two contributions to research on source code generation from models for DRE systems. First, it presents A-Circle, an algorithm that automatically removes circular dependencies when generating source code from models for programming languages that inherently enable circular dependencies. Secondly, this paper quantitatively evaluates A-Circle when generating CORBA IDL files. The results show that A-Circle algorithm is able to generate IDL files in linear-time.

I. Introduction

Model-driven engineering (MDE) techniques [12], such as domain-specific modeling languages (DSMLs) [7], use graphical notations to capture abstractions (*i.e.*, models) and semantics (*i.e.*, constraints) of the target domain. DSMLs also shield end-users, such as software system developers and testers, from both inherent and accidental complexities of the target domain that are typically tedious and error-prone to complete using either *ad hoc* techniques (*e.g.*, using handcrafted scripts to generate, modify, and validate configuration files) or manual processes (*e.g.*, defining interface definitions and deployment descriptors that contain valid interactions and connections between components).

A critical step in using DSMLs is *model interpretation*. During model interpretation, model interpreters transform constructed models into concrete artifacts, such as source code, XML configuration files, and models for other DSMLs. For example, in the domain of distributed real-time and embedded (DRE) systems, DSMLs have been used to generate a variety of concrete artifacts ranging from dense XML configuration files [1], [2] to actual source code [6].

Although model interpretation can automate many tasks when designing and implementing DRE systems, code generation from models is a *hard* task depending on the target programming language. For example, many modern-day DRE systems are implemented with software architectures and middleware that use Interface Definition Language (IDL) [10], [11]. One of the major difficulties of code generation for IDL, and similar programming languages like C++, is handling circular dependencies. A *circular dependency* [8] is when two different type definitions reference each other thereby creating a cyclic order. Failure to resolve circular dependencies in auto-generated source code can result in source code that fails to compile.

This paper therefore presents experience and results on using DSMLs for designing and implementing DRE systems to generate both valid and succinct IDL. The main contributions of this paper are:

- It presents *Automated Circular Dependency Resolver (A-Circle)*, which is an algorithm that automatically removes circular dependencies when generating source code from models for programming languages that accidentally allow circular dependencies;
- It presents results for integrating A-Circle with an interpreter from a representative DSML that generates IDL source files; and
- It presents an empirical study that evaluates A-Circle runtime performance in terms of number of generated IDL files, number of modules processed, and percentage of dependencies between declared elements.
- It presents an empirical study that evaluates A-Circle runtime performance in terms of percentage of circular dependencies between declared elements.

Finally, runtime analysis of applying A-Circle to a representative DSML shows that A-Circle is able to identify and resolve circular dependencies in $O(n)$ where n is the number of elements in the IDL model.

Paper organization. The remainder of this paper is organized as the follows: Section II motivates the need for A-Circle in the context of a representative DSML for DRE systems; Section III discusses the design and implementation of A-Circle; Section IV presents results from applying A-Circle to the representative DSML; Section V compares A-Circle to related works; and Section VI provides concluding

remarks and lessons learned.

II. Motivating Example: The Platform Independent Component Modeling Language

The *Platform Independent Component Modeling Language (PICML)* [2] is an open-source DSML designed and implemented in GME. PICML enables DRE system stakeholders, *e.g.*, DRE system developers and testers, to define and implement component-based DRE systems at a high-level of abstraction without being too bogged down with tedious and time-consuming low-level details, such as manually handcrafting and validating IDL files and dense XML-based deployment descriptors.

PICML accomplishes this task by providing DRE system stakeholders with intuitive graphical modeling elements that enable them to handle complex DRE system development tasks (*e.g.*, multi-aspect visualization of components and the interactions of their subsystems, component deployment planning, and hierarchical modeling of component assemblies). Lastly, PICML's model interpreters can transform constructed models into many different kinds of concrete artifacts, such as configuration files for analytical tools and tarballs that can be installed into implementation repositories that facilitate remote lookup for deployment.

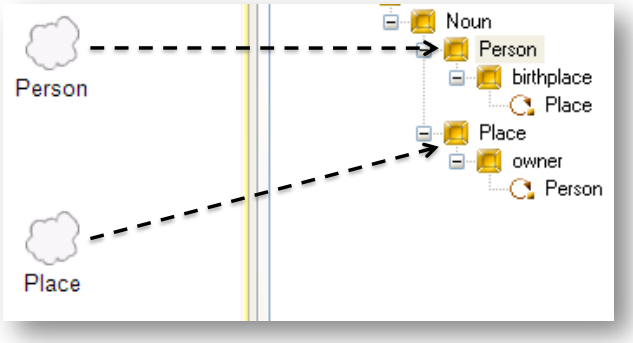

Figure 1. Example interface definition in PICML.

As stated above, one key aspect of PICML is the specification of a component-based DRE system's interfaces and attributes. This is done using graphical elements that represent IDL constructs, *e.g.*, module, exception, constant and type definitions. From such models, one of PICML's model interpreters is able to generate IDL files. Although one of PICML's goals is to generate IDL from constructed models, this is not a trivial task. For example, Figure 1, illustrates the interface definition aspect of an example PICML model. Likewise, Listing 1 highlights invalid code that can result from the model in Figure 1 based on a naïve implementation of an IDL generator.

Compilation of the IDL presented in Listing 1, however, would fail because the IDL definition contains a circular dependency. More specifically, the `Person` interface provides a method named `birthplace()` that returns an interface of type `Place`. Likewise, the interface `Place` provides a method named `owner()` that returns an interface of type `Person`. Because the method in either interface returns an interface to the other interface, this is a circular dependency. The IDL compiler will therefore generate an error because interface `Place` isn't defined before interface `Person` is processed.

```
1  module Noun {
2    interface Person {
3      ::Noun::Place birthplace ();
4    };
5
6    interface Place {
7      ::Noun::Person owner ();
8    };
9  };
```

Listing 1. Invalid IDL code generated from example model above.

Because generation of IDL from PICML models can result in invalid IDL source code, there are two critical challenges that must be addressed:

Challenge 1: Identifying and resolving circular dependencies among model elements. As shown in Figure 1 and Listing 1, it is easy to generate invalid IDL from a model. It is therefore necessary to ensure that generated IDL is valid in not only its syntax, but also handles circular dependencies as well. Moreover, it should handle such concerns in a timely manner, *i.e.*, use an efficient algorithm that scales well with the size and complexity of the model. Section III discusses how A-Circle's algorithm addresses this challenge.

Challenge 2: Generating valid IDL from succinct models. The purpose of using a DSML, such as PICML, is to increase the level-of-abstraction and reduce the *problem-implementation gap* [5] for the target domain. In the case of PICML, it should increase the level-of-abstraction so DRE system developers are not too concerned with low-level implementation details. DRE system developers should only be concerned with modeling their system as accurately as possible while capturing the essence of the design. It is therefore the responsibility of the model interpreter to infer low-level implementation details, such as resolving the circular dependencies. Section III discusses how A-Circle resolves this challenge.

This problem is not only contained to IDL, but also to other programming languages similar to IDL, such as C++, that want to support code generation from models. The remainder of this paper therefore discusses how A-Circle addresses the two challenges outlined above, and enables generation of valid IDL source code from PICML models.

III. The Design and Implementation of A-Circle

This section discusses the design and implementation of A-Circle. This section also discusses how A-Circle addresses the challenges introduced in Section II

A. Design Alternatives for A-Circle

Before discussing the details of A-Circle, it is necessary to understand the different design alternatives that influence A-Circle's design and implementation. This helps highlight the advantages and disadvantages of each approach, which may be beneficial for application in other domains.

When designing and implementing A-Circle, the following design alternatives were considered:

- **Approach 1: Avoid all circular dependencies.** Circular dependencies can be completely avoided by requesting the modeler to construct models without circular dependencies. This can be done by adding a constraint that checks for circular dependencies and alerting the modeler if the constraint fails. The advantage of this approach is implementing the model interpreter is a trivial process. This is because the model interpreter does not require intelligence that guarantees the generated IDL does not contain circular dependencies.
  This approach, however, has several disadvantages. First, it is *hard* for modelers to construct models that contain no circular dependencies for large and complex systems that result in 10's of 100's of generated IDL files. Secondly, this approach places more effort on the modeler and contradicts the goal of DSMLs reducing the level-of-complexity for its target domain.
- **Approach 2: Manually resolve circular dependencies.** Since it can be *hard* to avoid circular dependencies—especially in large models—modelers can be asked to explicitly resolve circular dependencies themselves in the model. This can be done by providing modelers with an abstraction for declaring a type before it is actually defined in the model—similar to forward declarations in IDL and C++.
  The advantage of this approach is that the interpreter is able to generate valid IDL with the assistance of the modeler. This can result in a more robust implementation of the model interpreter. Furthermore, if other interpreters want to parse the model, then they do not have to implement a circular dependency resolver.
  This main disadvantage of this approach is it adds an extra level-of-complexity to the DSML. This is because modelers have to focus on low-level implementation details. In this case, it is resolving circular dependencies. Finally, this approach is similar to the approach currently used when manually writing IDL source files by hand, which is tedious and error-prone.
- **Approach 3: Automatically revolve circular dependencies.** The final approach is to implement the logic for resolving circular dependencies inside the model interpreter. The main advantage of this approach is it does not require developer to posses domain-knowledge for resolving circular dependencies, which can be a *hard* task. The disadvantage of this approach is that each model interpreter that wants to parse the corresponding model must implement its own circular dependency resolver. If not done correctly, each implementation may have different runtime complexities or function incorrectly

The purpose of a DSML, such as PICML, is to reduce complexity of a given domain by increasing the level-of-abstraction and reducing the amount of tedious and error-prone process. Because of this fact, A-Circle uses Approach 3 as its design choice when resolving circular dependences in generated IDL code from PICML models. Approach 3 was selected because it requires less modeling effort and domain-knowledge from the modeler. The modeler can therefore focus on modeling the DRE system. Approach 3 provided the necessary functionality for resolving Challenge 1 and Challenge 2 presented in Section II. The remainder of this section therefore explains in detail how A-Circle addresses these challenges.

B. The Foundation for Generating Basic IDL from Models

Identifying and resolving circular dependencies is *hard* for source code generators, such as the IDL generator in PICML. A naïve approach for resolving circular dependencies in generated IDL source code is to forward declare all type definitions, where applicable, as shown in Listing 2.

```
1  module Noun
2  {
3    interface Idea;
4    interface Place;
5    interface Thing;
6    interface Person;
7  };
8
9  module Noun {
10   interface Idea {
11     ::Noun::Place location ();
12   };
13
14   interface Place {
15     attribute string name;
16   };
17
18   interface Thing {
19     attribute string name;
20   };
21
22   interface Person {
23     ::Noun::Place birthplace ();
24   };
25 };
```

Listing 2. Example of forward declaring all elements in generated IDL source code.

Although this approach is feasible, *i.e.*, the generated IDL source files compiles, it does not generate optimal IDL source code because it forward declares `::None::Place` and `::Noun::Thing`, which are not dependent on any other definition. Moreover, this approach does not address Challenge 2 presented in Section II. For example, a model could contain 1000 elements an have no (circular) dependency between elements. This approach would forward declare all 1000 model elements, which is unnecessary.

A better approach is for the model interpreter to only forward declare what needs to be forward declared—similar to how DRE system developers proceed manually. First, this requires identifying dependencies between each declaration, which can be accomplished using depth first search or breadth first search.

More specifically, a IDL model can be represented as a directed graph $G(V, E)$ where V is the set of declared elements in the model (*i.e.*, model elements that represent type definitions). Likewise, E is set of directed edges between to vertices in G such that $E_{i,j} \in E$ means $V_i \in V$ depends on $V_j \in V$. In other words, element V_j must be declared before V_i in the generated IDL file.

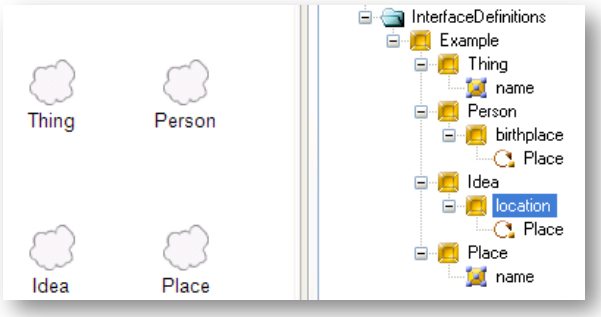

Figure 2. Simple IDL model where all type definitions are defined at the global scope.

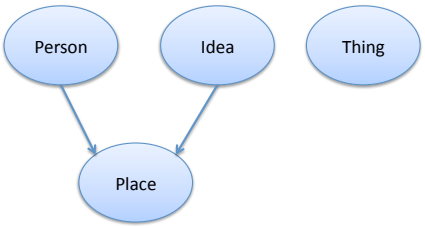

Figure 3. Directed graph for the IDL model shown in Figure 2.

For example, Figure 2 illustrates a model where each type is defined at the global scope (*i.e.*, not within a module, which is discussed later in this section). The example in Figure 2 results in the directed graph illustrated in Figure 3. Using this directed graph, it is possible to generate valid and succinct IDL source code by topologically sorting the graph and visiting each node in order. Upon visiting each node, the correct IDL source code is generated to the target source file. Listing 3 shows the resulting IDL for using this approach.

```
1  interface Thing {
2    attribute string name;
3  };
4
5  interface Place {
6    attribute string name;
7  };
8
9  interface Idea {
10   ::Noun::Place location ();
11 };
12
13 interface Person {
14   ::Noun::Place birthplace ();
15 };
```

Listing 3. Resulting IDL source code for the model in Figure 2. This IDL is based on topologically sorting the directed graph in Figure 3 and processing the nodes in order.

C. Handling Type Definitions in Child Modules

The discussion above applies only to elements declared within global scope (*i.e.*, not within a module). IDL and IDL modeling tools allow DRE system developers to define types within modules, which are analogous to namespaces in C++ or packages in Java. The approach outlined above for generating valid IDL, however, does not map to models where elements are defined in modules. This is because the dependency graph cannot be treated as a flat space.

```
1  interface Place { };
2
3  module Noun {
4    interface Idea {
5      ::Place location ();
6    };
7
8    interface Person {
9      ::Place birthplace ();
10   };
11 };
```

Listing 4. Textual model representation of IDL that contains packages.

For example, consider the IDL in Listing 4. As shown in this listing, there are two elements where element `Place` is declared outside of module `Noun` and element `Idea` is declared inside module `Noun`. The directed graph for this example would show that element `Idea` depends on element `Place`. The resulting generated IDL could declare `Place` and `Person` outside of module `Noun` if the hierarchy is not taken into account. Likewise, the model interpreter could generate verbose code by wrapping each element in its on individual scope (shown in Listing 5) which is not considered optimal code generation.

```
1  interface Place { };
2
3  module Noun {
4    interface Idea {
5      ::Place location ();
6    };
7  };
8
9  module Noun {
10   interface Person {
11     ::Place birthplace ();
12   };
13 };
```

Listing 5. Verbose IDL that can result from models that contain modules.

To ensure optimal code generation for IDL models that contain modules, it is necessary to preserve the element hierarchy in such a way that the code generation avoids

what is illustrated in Listing 5. This is achieved by using directed subgraphs where each module has a subgraph for its contained model elements, as shown in Figure 4. Each module is also treated as element in its parent module (or directed graph). Finally, if a type depends on another type that is defined in another module, instead of adding a edge between the two types—as originally done—an edge is added between the two ancestor elements that are siblings in the declaration hierarchy.

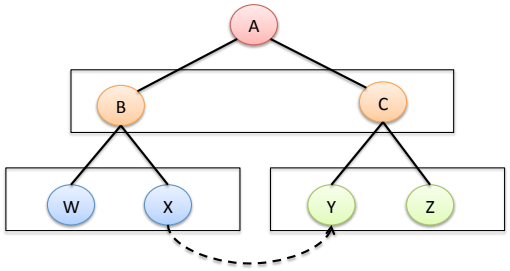

Figure 4. Conceptual overview of elements that belong to a subgraph.

For example, as illustrated in Figure 4, X depends on Y, but X and Y are declared in different subgraphs. This implies that the module containing Y needs to occur before the module containing X (*i.e.*, B depends on C). Once the subgraphs for each module is constructed, it is possible to generate IDL source code.

Algorithm 1 General algorithm building dependency subgraphs from models that contain modules.

procedure BUILDSUBGRAPH(SG, M)
  SG: map of all subgraphs
  M: current model element
  g: current subgragh

  for all $e_i \in M$ **do**
    if $is\_module(e_i)$ **then**
      $SG[e_i] \leftarrow BuildSubGraph(SG, e_i)$
      $add\_vertex(g, e_i)$
    else if $is\_reference(e_i)$ **then**
      $add\_edge(g, e_i)$
    else
      $add\_vertex(g, e_i)$
    end if
  end for

  return g
end procedure

Algorithm 1 presents the algorithm for constructing the subgraphs discussed above. As shown in this algorithm, A-Circle visits each element e_i in the current model M. If the current element is a module, then A-Circle creates a subgraph for that module and stores it in the map SG that contains all the subgraphs. A-Circle then adds e_i as a vertex to the current subgraph g.

If the current element e_i is a reference (*i.e.*, refers to another element declared in the global model), then A-Circle adds an edge between two existing vertices using the following rules:

- If the parent of the current and referenced element are siblings, then add a directed edge between the parent of the current element and the referenced element;
- Otherwise, locate ancestors for the current and referenced element that are siblings and create a directed edge between the current element's located ancestor and the referenced element's located ancestor.

Finally, A-Circle returns the constructed subgraph g to the caller. As shown in the algorithm above, A-Circle is able to complete this process in $O(n)$ time where n is the number of elements in the complete model.

Algorithm 2 General algorithm generating IDL source files from models that contain modules.

procedure GENERATEIDL(SG, M)
  M: current model element
  SG: map of all subgraphs
  g: current subgraph

  $g \leftarrow SG[M]$
  $g' \leftarrow topological\_sort(g)$

  for all $e_i \in g'$ **do**
    $GenerateIDLTag(e_i)$

    if $is\_package(e_i)$ **then**
      $GenerateIDL(SG, e_i)$
    end if
  end for
end procedure

Once all subgraphs are constructed, it is necessary to generate IDL using them. Algorithm 2 therefore shows A-Circle's algorithm for generating valid IDL from models that contain modules. As shown in this algorithm, A-Circle first gets the subgraph g for the current model M from the collection of subgraphs SG. Next, A-Circle topologically sorts the subgraph to ensure that elements are visit in order of their dependencies. Finally, for each element e_i in the topologically sorted graph g', A-Circle generates the appropriate IDL tag.

Likewise, if the current element is a module, then A-Circle recursively calls the generate IDL algorithm while passing the mapping of subgraphs and the current element. This allows the A-Circle to ensure that scope hierarchy is preserved during the source code generation. As shown in

Algorithm 2, the runtime complexity of this algorithm is $O(n)$ where n is the number of vertices in SG.

D. Handling Circular Dependencies in Type Definitions

The previous sections discussed how to generate valid and optimal IDL for trivial models (*i.e.*, models where all types are defined with the global scope) and models that contains modules and dependencies between elements in different packages, but no circular dependencies. The final problem to address with generating valid and optimal IDL source code from succinct models is handling circular dependencies between elements. This, however, adds another layer of complexity to the solution because it requires extra processing of the original model. In the case of IDL, this means preprocessing the model to determine what type definitions need to be forward declared.

As previously discussed, a circular dependency occurs when two model elements are mutually dependent on one another. More specifically, $e_{i,j} \in E$ and $e_{j,i} \in E$. Based on this definition, one of the edges is considered a *back edge*. To resolve the circular dependency, it is necessary to remove the back edge from the graph. Once the back edge is remove, the destination node of the back edge is forward declared. Algorithm 3 gives the general algorithm for resolving circular dependencies with generating IDL from models.

Algorithm 3 General algorithm for resolving circular dependencies when generating IDL from models.

```
procedure ResolveCircularDependency(G, v)
    G: current subgraph
    v: current vertex
    F ← ∅
    E ← get_edges(G, v)

    for all e_i ∈ E do
        v = dest_vertex(e_i)
        if is_back_edge(e_i) then
            G ← remove_edge(G, e_i)
            F ← F ∪ v
        end if

        F ∪ ResolveCircularDependency(G, d)
    end for

    return F
end procedure
```

Algorithm 3 presents A-Circle's approach for resolving circular dependencies when generating IDL source files from succinct models. As shown in this algorithm, first A-Circle get all the outgoing edges for the current vertex in the subgraph. For each edge, A-Circle then checks if the edge is a back edge. If the current edge is a back edge, then A-Circle removes the back edge and inserts the destination vertex (or element) into a list F of forward declarations. This process continues until all vertices (or elements) in the subgraph are visited.

Once Algorithm 3 identifies and resolves circular dependencies in the IDL model, the next step is to generate the IDL source file. This is accomplished using Algorithm 2 in Section III-C. The only extension is that when each module (or subgraph) is processed, elements identified for forward declaration are forward declared. Likewise, if an isolated module hierarchy must be defined to forward declare an element, then both algorithms are able to handle this case.

IV. Results

This section empirically evaluates A-Circle's algorithm against different IDL models.

A. Experimental Setup

In order to evaluate A-Circle's algorithm, we implemented A-Circle into PICML's IDL generator model interpreter. We also developed a Perl script named *IDLAutoGen* to auto-generate valid IDL files for testing A-Circle. IDLAutoGen is configurable and accepts the following input parameters:

- **File Size** (F) – number of logic file types generated;
- **Package Size** (P) – number of logic modules generated per file type; and
- **Circular Dependency Distribution** (D) – percentage of circular dependencies between declared model elements.

The IDL files generated by IDLAutoGen are then processed by the `IDL_to_PICML` text-to-model tool that creates a PICML model from a collection of IDL files. After the PICML model is created, it is imported into GME and transformed back to IDL using PICML's IDL interpreter that implements A-Circle's algorithm.

All experiments were performed on an AMD Athlon X2 5400 system configured with Windows XP SP3 and 4GB RAM. This system's integrated development environment is typical for modelers using GME and PICML to generate distributed application. Table I lists the version information for the applications used during the experimental evaluation.

Table I
Version information for software used to evaluate A-Circle's algorithm.

Application	Version
GME	10.8.18
UDM	3.26
Boost Graph Library	1.43
CoSMIC	0.8.2
Microsoft Visual Studio	2008SP1

B. Experimental Results

Figure 5 illustrates the runtime of A-circle when generating files that contains one package (P=1) each with no circular dependencies (D=0%) among model elements. Likewise, Figure 6 illustrates the runtime of A-circle when generating packages within a single file (F=1) with no circular dependencies (D=0%). Finally, Figure 7 illustrates the runtime of A circle when generating packages that contains circular dependency based on a circular dependency distribution (10%–50%) and number of files (F=1) and number of modules (P=30).

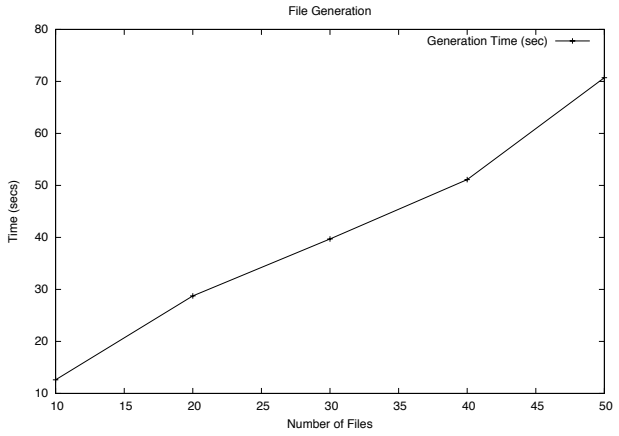

Figure 5. A-Circle's execution time to generate IDL files based on the number of files in the model.

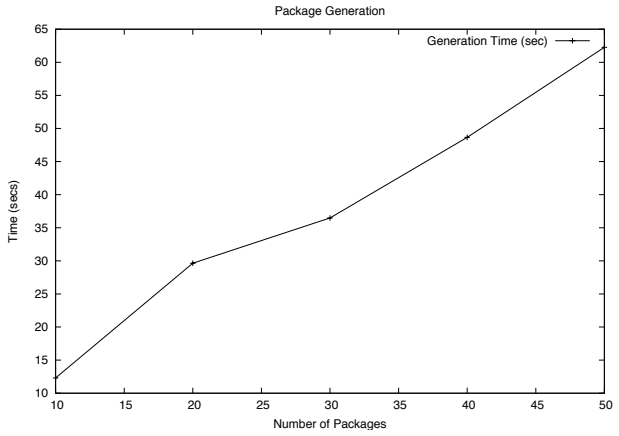

Figure 6. A-Circle's execution time to generate IDL files based on the number of modules in the model.

Figure 5 and Figure 6 show that A-Circle execution time increases somewhat linearly with respect to the number of files and packages generated. We believe it is not a completely linear increase because the experiments where run on a laptop, and not in an isolated environment. Nonetheless, the empirical results are inline with the runtime complexity of A-Circle's algorithm of $O(n)$. Moreover, these empirical result confirm our analysis and implementation of A-Circle's algorithm.

Figure 7 illustrates the runtime of A circle when generating packages that contains circular dependency based on a circular dependency distribution (10%–50%) and number of files (F=1) and number of modules (P=30).

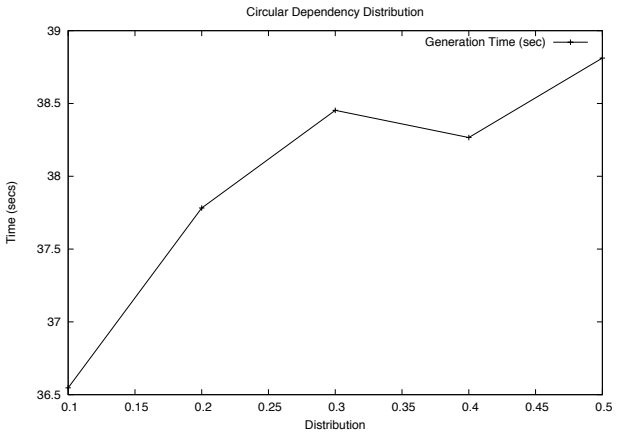

Figure 7. A-Circle's execution time to generate IDL files based on the percentage of circular dependencies between declared elements.

Figure 7, however, shows results different from Figure 5 and Figure 6. As shown in Figure 7, the results are less linear than the results illustrated in the previous two figures. Although it can be argued that the experiment's environment is playing a part of these results, this is not always the case. The results in Figure 7 shows a linear progression with increased circular dependency distribution until circular dependency distribution 40%. Although, these results still show a linear equation, we believe that the Perl script algorithm may generate less circular dependencies at 40% than what we would expect. This may be the cause for the slight decrease in runtime at circular dependency distribution 40% compared to circular dependency distribution 30%.

Although it can be argued that the experiment's environment is playing a part of these results, this is not always the case. The results in Figure 7 shows a linear progression with increased circular dependency distribution until circular dependency distribution 40%. Although, these results still show a linear equation, we believe that the Perl script algorithm may generate less circular dependencies at 40% than what we would expect. This may be the cause for the slight decrease in runtime at circular dependency distribution 40% compared to circular dependency distribution 30%.

V. Related Works

Code generation and optimization can be performed during different phases of the model transformation process. Moreover, such transformations can operate on succinct or verbose models. For example, Charfi et al. [3] promotes the idea of model compiler where the model is compiled directly

to binary code. Until such model transformation is possible, optimizations will need to be performed at different stages of model transformation. A-Circle's approach therefore is an implementation of this need because it main purpose is to optimize source code generation from succinct models, thereby reducing transformation effort at last phases of the transformation process.

Charfi et al. [4] presents work that performs optimized source code generation from models. The aim of their work is to optimize source code generation from a model based on size of the final assembly code generated by a compiler. A-Circle approach differs in this approach because it aims at optimizing source code generation from model by reducing the number of lines of source code in the compilation unit.

Merilinnai et al. [9] also presents work that performs source code optimizations when generating from a model. Unlike Charfi's work, Merilinnai allows the end-user to select what metrics, such as performance and reliability, are used to optimize the source code generation. A-Circle's approach differs from this work in that the modeler is not able to select how to optimize the code generation. It is believed that A-Circle can leverage such concepts when generating is source code, but there is currently no value in having such options.

VI. Concluding Remarks

This paper presented the design and implementation of the *Automated Circular Dependency Resolver (A-Circle)*. A-Circle is an algorithm that automatically removes circular dependencies when generating source code from models for programming languages that accidentally allow circular dependencies. In addition, this paper showed how A-Circle was applied a representative DSML that generates IDL files from models. Based on experience gained from designing and implementing A-Circle, the following is a list of future research directions:

- **Generalization of problem.** The current implementation of A-Circle is bound to the PICML metamodel. This makes it hard to apply A-Circle to other application domains system it has to be re-implemented each time. Future research therefore will focus on generalizing A-Circle's implementation for usage in other application domains.
- **Optimize performance based on file characteristic.** The initial results showed A-Circle's performance for standard metrics—yet simple metrics. Future research therefore will develop more complex metrics for evaluating A-Circle's performance. Moreover, such analysis will be used to identify characteristics and optimize A-Circle's implementation for those characteristics. This will ensure that A-Circle always performs optimally regardless of the size and complexity of the parsed model.

PICML is freely available for download in open-source format from the following location: www.dre.vanderbilt.edu/cosmic.

References

[1] K. Balasubramanian. *Model-Driven Engineering of Component-based Distributed, Real-time and Embedded Systems*. PhD thesis, Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, Sept. 2007.

[2] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt. A Platform-Independent Component Modeling Language for Distributed Real-Time and Embedded Systems. In *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS05)*, San Francisco, CA, March 2005.

[3] A. Charfi, C. Mraidha, S. Gérard, F. Terrier, and P. Boulet. Does Code Generation Promote or Prevent Optimizations? In *IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, 2010.

[4] A. Charfi, C. Mraidha, S. Gerard, F. Terrier, and P. Boulet. Toward Optimized Code Generation Through Model-based Optimization. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*, 2010.

[5] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, pages 37–54, 2007.

[6] J. H. Hill, S. Tambe, and A. Gokhale. Model-driven Engineering for Development-time QoS Validation of Component-based Software Systems. In *Proceedings of 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 07)*, pages 307–316, Tucson, AZ, Mar 2007.

[7] Á. Lédeczi, Á. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.

[8] K. McMillan. The Circular Dependency Rule. http://santos.cis.ksu.edu/smv-doc/language/node17.html.

[9] J. Merilinna and T. Raty. A Tool for Quality-Driven Architecture Model Transformation. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2010.

[10] Object Management Group. *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 1: CORBA Interfaces*, OMG Document formal/2008-01-04 edition, Jan. 2008.

[11] R. E. Schantz and D. C. Schmidt. Middleware for distributed systems - evolving the common structure for network-centric applications, 2001.

[12] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.