Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37235

Towards Incremental Cycle Analysis in ESMoL Distributed Control System Models

Joseph Porter, Daniel Balasubramanian, Graham Hemingway, and János Sztipanovits

TECHNICAL REPORT

ISIS-11-106

Abstract

We consider the problem of incremental cycle analysis for dataflow models in the Embedded Systems Modeling Language (ESMoL). We give a general form of a cycle enumeration algorithm that makes use of graph hierarchy to improve analysis efficiency. Our framework also stores simple connectivity information in the model to accelerate future cycle analyses when additional components are added or modifications are made. Finally we give a mapping from a term algebraic model of the ESMoL component model and logical dataflow sublanguages to the analysis framework, and an evaluation on a fixed-wing aircraft controller model. This is part of a larger effort to integrate cycle analysis into the ESMoL tool suite to aid well-formedness checking during model construction.

1 Introduction

High confidence embedded control system software designs often require formal analyses to ensure design correctness. Detailed models of system behavior include numerous design concerns, such as controller stability, timing requirements, fault tolerance, and deadlock freedom. Models for each of these domains must together provide a consistent and faithful representation of the potential problems an operational system would face. This poses challenges for structural representation of models, as components and design aspects are commonly tightly coupled. The ESMoL language is built on a platform which provides inherent correctness properties for well-formed models. The properties include functional determinism, deadlock freedom, and timing determinism. We also rely on decoupling methods such as passive control design (decoupling controller stability from network effects) and time-triggered models of computation (decoupling timing and fault tolerance from functional requirements) and on compositional and incremental analysis to enable rapid prototyping in our design environment. As design paradigms become more fully decoupled and analysis becomes faster (and therefore cheaper), we move closer to the goal of "correct by construction" model-based software development.

In compositional analysis for graphical software models, sometimes the nature of the analysis does not easily lead to a clean syntactic decomposition in the models. Examples include end-to-end properties such as latency, and other properties which require the evaluation of particular connections spanning multiple levels of components. One approach for dealing with such properties in hierarchical dataflow designs is the creation of interface data for each component which abstracts properties of that component. Hierarchical schedulability models defined over dataflows are a particular example[14] – each composite task contains a resource interface characterizing the aggregate supply required to schedule the task and all of its children. Extensions to the formalism allow the designer to efficiently and incrementally evaluate whether new tasks can be admitted to the design without recomputing the full analysis[5]. One goal is to see whether this approach can be generalized to other properties that do not easily fit the compositional structure of hierarchical designs.

One particular syntactic analysis problem concerns synchronous execution environments and system assembly. In dataflow models of computation we are often concerned with so-called “algebraic” or delay-free processing loops in a design model. Many synchronous formalisms require the absence of delay-free loops in order to guarantee deadlock freedom [1] or timing determinism [10]. This condition can be encoded structurally into dataflow modeling languages – for example Simulink [16] analyzes for algebraic loops and attempts to resolve them analytically. In the Ptolemy dataflow design environment, such causality loops complicate scheduling requiring fixed-point iteration to ensure convergence of results[19]. In our work we only consider the structural problem of loop detection in model-based distributed embedded system designs.

We propose a simple incremental cycle enumeration technique with the following characteristics:

- The algorithm uses Johnson’s simple cycle enumeration algorithm as its core engine[7]. Johnson’s algorithm is known to be efficient [11]. We use cycle enumeration rather than simple detection in order to provide useful feedback to the designers.

- The algorithm exploits the component structure of hierarchical dataflow models to allow the cycle enumeration to scale up to larger models. A small amount of interface data is created and stored for each component as the analysis processes the model hierarchy from the bottom up. The interface data consists of a set of typed graph edges indicating whether dataflow paths exist between each of the component’s input/output pairs. Each component is evaluated for cycles using the interface data instead of the detailed dataflow connections of its child components.

- The interface data facilitates incremental analysis, as it also contains a flag to determine whether modifications have been made to the component. We refer to the flag and the edges as an *incremental interface* for the component. This is consistent with the use of the concept in other model analysis domains, such as compositional scheduling analysis[5]. In order for the incremental method to assist our development processes, the total runtime for all partial assessments of the model should be no greater than the analysis running on the full model. Because the amount of interface data supporting the incremental analysis is small, the method should scale to large designs without imposing onerous data storage requirements on the model.

- The technique will not produce false positive cycle reports, though it may compress multiple cycles into a single cycle through the interface abstraction. Fortunately, full cycles can be recovered from the abstract cycles through application of the enumeration algorithm on a much smaller graph.

Zhou and Lee presented an algebraic formalism for detecting causality cycles in dataflow graphs, identifying particular ports that participate in a cycle.

[20]. Our work traverses the entire model and extracts all elementary cycles, reporting all ports and subsystems involved in the cycle. Our approach is also inspired by work from Tripakis et al, which creates a richer incremental interface for components to capture execution granularity as well as potential deadlock information[18]. Their approach is lossless, in that it retains sufficient detail to faithfully represent dataflow structure and execution granularity. It is much more complex in both model space and computation than our approach. Our formalism does not aim to pull semantic information forward into the interface beyond connectivity. In that sense our approach is more general, as it could be applied to multiple model analysis problems in the embedded systems design domain.

The KPASSA model analysis tool described by Boucaron et al [3] performs task graph scheduling analysis for latency-insensitive synchronous designs. Their formal model leans heavily on loop structures, and as such one component of their tool relies on an implementation of Johnson's cycle enumeration algorithm[2]. Their formal model is specific to a particular model of computation, and their application of cycle checking is only one small component of that solution.

2 Background

2.1 ESMoL Component Model

As the ESMoL language structure is documented elsewhere[13], we only cover details relevant to incremental cycle checking. ESMoL is a graphical modeling language which allows designers to use Simulink diagrams as synchronous software function specifications (where the execution of each block is equivalent to a single bounded-time blocking C language call). These specifications are used to create blocks representing ESMoL component types. ESMoL components have message structures as interfaces, and the type specification includes a map between Simulink signal ports and the fields of the input and output message structures. The messages represent C structures, and the map graphically captures the marshaling of Simulink data to those structures.

Once software component types and interfaces have been specified, ESMoL designers instantiate those components into a distributed deployment model. ESMoL allows the separate specification of the logical data flow, the mapping of component instances to hardware, timing information for tasks executing those components, and timing for messages sent over a time-triggered communication bus. Code generated from the models conforms to an API for time-triggered execution. A portable virtual machine implementation of the API allows execution in simulation, hardware-in-the-loop, and fully deployed configurations[6].

ESMoL deliberately provides an unusual degree of freedom in creating software component types. A designer can include Simulink references from any part of an imported dataflow model, and instantiate them any number of times within the type definitions. The partition of functions into ESMoL types allows the designer to control the granularity of functions assigned to distributed tasks. Tasks can distribute functions over a time-triggered network for performance, or replicate similar functions for fault mitigation. This level of flexibility requires automatic type-checking to ensure compatibility for chosen configurations. Beyond interface type-checking, structural well-formedness problems arise during assembly such as zero-delay cycles. Model analysis must ensure well-formedness.

2.2 Cycle Enumeration

To implement cycle enumeration we use the algorithm Johnson proposed as an extension of Tiernan's algorithm [17] for enumerating elementary cycles in a directed graph[7]. Both approaches rely on depth-first search with backtracking, but Johnson's method marks vertices on elementary paths already considered to eliminate fruitless searching, unmarking them only when a cycle is found. Johnson's algorithm is polynomial ($O((n+e)c)$, where n, e, and c are the sizes of the vertex, edge, and cycle set, respectively), and is still considered the best available general cycle enumeration method[11]. We created an implementation of Johnson's algorithm in C++ using the Boost Graph library [15].

2.3 Hierarchical Graphs

For formally describing our incremental approach we use the algebra of hierarchical graphs introduced by Bruni et al[4]. We repeat here their first definition: a *design* is a term of sort $\mathcal{D}$ generated by

$$\begin{aligned} \mathbb{D} &::= L_{\bar{x}}[\mathbb{G}] \\ \mathbb{G} &::= \mathbf{0} \mid x \mid l < \bar{x} > \mid \mathbb{G} \parallel \mathbb{G} \mid (\nu\bar{x})\mathbb{G} \mid \mathbb{D} < \bar{x} > \end{aligned} \tag{1}$$

Here term $\mathbb{G}$ represents a hierarchical directed graph, $\mathbb{D}$ is an edge-encapsulated hierarchical graph, x is a vertex, $\bar{x}$ is a list of vertices in $\mathbb{G}$ (for which $\lfloor\bar{x}\rfloor$ is the corresponding set), $l \in \mathcal{E}$ (edge labels of $\mathbb{G}$, where edges can have n-ary connectivity), $L_{\bar{x}} \in \mathcal{D}$ ($\mathcal{D}$ are the design labels of $\mathbb{G}$ and $\bar{x}$ are interface vertices in L), $\mathbb{G} \parallel \mathbb{G}$ is parallel graph composition which merges vertices with common names, $(\nu\bar{x})\mathbb{G}$ restricts the interface of graph $\mathbb{G}$ to exclude vertices in $\lfloor\bar{x}\rfloor$, and the notation $\mathbb{D} < \bar{x} >$ maps the vertices from the interface of $\mathbb{D}$ to the vertices listed in $\bar{x}$ (renaming vertices internal to the design for the external interface). Finally $[\![\mathbb{G}]\!]$ indicates the graph corresponding to the term $\mathbb{G}$.

Intuitively, Equation 1 is a grammar defining a simple textual notation for describing typed hierarchical graphs. Within the formalism we can compare equivalence between algebraic descriptions of two hierarchical graphs using reduction rules and a normal form (as in Bruni[4]), though equivalence is beyond the scope of this publication. The algebraic properties are for future use. The other main attraction of this particular formalism is that the notation allows the definition of interface symbols which correspond easily to port objects in a dataflow language, and the hiding of those interfaces as we specialize types.

The notation is a compact shorthand for much larger diagrams or mathematical descriptions. The design sorts $\mathbb{D}$ correspond to composite types in our dataflow language (which may have children), and the edge-encapsulation means that graph edges are not visible outside the component that contains them. The specification for a composite element is $L_{\bar{x}}[\mathbb{G}]$, which means that an element of the sort $\mathbb{D}$ has type L with interface vertices in the list $\bar{x}$ and a corresponding internal graph $\mathbb{G}$ defining the details of the component. The internal graph may also include subcomponents. Gluing of subgraphs (contained by the design sorts $\mathbb{D}$) only occurs at common vertices. When a composite element from $\mathbb{D}$ of a particular type is used as a child element to form a larger (parent) graph, vertices from the child are possibly renamed in the parent, hence the notation $\mathbb{D} < \bar{x} >$. In a parallel composition, vertices with the same name x are glued together.

Note that the term *design* in the formalism is synonymous with the concept of a *component* in a modeling hierarchy, with a type and a set of interface vertices. Unfortunately in the realm of graph theory the term *component* has a different meaning. The term-algebraic graph formalism also includes a definition of *well-typedness*, where types defined on the vertex set are only connected if their types are compatible. Finally the authors define *well-formedness* for hierarchical graphs which includes *well-typedness* as a condition. We do not define the entire formalism here, only enough to understand the essence of the connections between the terms and the graphs that they represent.

3 Incremental Cycle Analysis

Our intention is to support a design and analysis work flow that includes incremental analysis steps. For example, a designer may analyze part of the design before integrating it into a larger part of the system. In our work flow, we envision storing the results of that first analysis along with some interface data to reduce the cost of the second analysis. The same should hold true for the system design. We should be able to analyze the system design efficiently, calculating incremental analysis interfaces. When the system models are revised, whether by adding, removing, or modifying components we can isolate the effects of the change on the cost of the analysis. Cycle analysis is a useful example, but our aim is to tackle this problem more generally.

Formal Model

Let $\mathbb{G}$ be a well-formed hierarchical graph (as in Bruni [4]). To get more comfortable with the notation, first note that graph $\mathbb{G}$ itself (without hierarchical structure) can be given as:

$$\mathbb{G} = (\| \, x) \, \| \, (\|_{(u,v)\in\mathcal{E}} \; l < u, v >) \tag{2}$$

which is the parallel composition of the graphs induced by individual edges of $\mathbb{G}$, merged at their common vertices.

Let $C(\mathbb{G})$ be the set of elementary cycles in $\mathbb{G}$, and let $P(\mathbb{G}, u, v)$ be the subgraph of $\mathbb{G}$ containing all of the paths from vertex u to vertex v.

Consider a design of type W. Let $W^p_{\bar{x}}[G]$ represent a parent design object in a graph hierarchy with interface vertices $\bar{x}$, and let $W^{c_i}_{\bar{x}_i}[G]$ be the children of W^p ($[\![W^{c_i}]\!] \subset [\![W^p]\!]$). Then neglecting vertex hiding and renaming to simplify the illustration, we have the following:

$$W^p_{\bar{x}}[\mathbb{G}] = W^p_{\bar{x}}[(\|_h\ x_h) \parallel (\|_{(j,k)}\ l < j,k >) \parallel (\|_i\ W^{c_i}_{\bar{x}_i}[\mathbb{G}])] \tag{3}$$

Eq. 3 describes the design W^p in terms of its design children W^{c_i}, internal vertices x_h, and edges $l < j,k >$. Fig. 1 gives a simple example of a hierarchical graph with one child component and one cycle including vertices and edges within the child component.

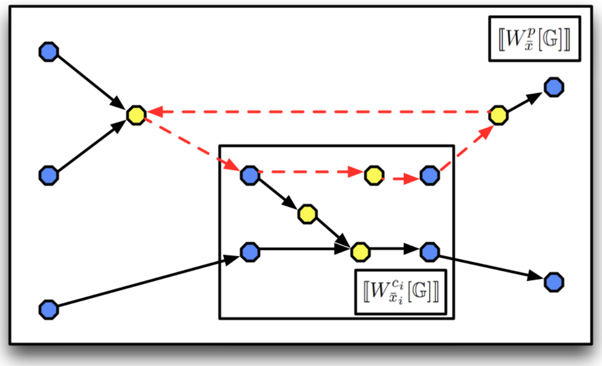
$[\![W^p_{\bar{x}}[\mathbb{G}]]\!]$
$[\![W^{c_i}_{\bar{x}_i}[\mathbb{G}]]\!]$

Figure 1: Generic hierarchical graph model

We introduce a new label l_c into the sort for edges ($\mathcal{E}$), which is used to connect vertices at the boundaries of a design, abstracting the interface connectivity of the design. Introduce a new mapping $A : \mathcal{D} \to \mathcal{D}'$ from the designs of $\mathbb{G}$ to designs in a new graph $\mathbb{G}'$. $\mathbb{G}'$ is identical to $\mathbb{G}$, but adds the new edge label. This is the interface that we will use for incremental cycle analysis.

$$\begin{aligned} A(W^{c_i}_{\bar{x}_i}[\mathbb{G}]) =& W^{c_i}_{\bar{x}_i}[(\|_h\ x_h) \parallel (\|_{(j,k)\in\lfloor\bar{x}_i\rfloor \wedge P(\mathbb{G},j,k)\neq\emptyset}\ l_c < j,k >) \\ & \parallel (\|_{(j,k)}\ l < j,k >) \parallel (\|_m\ W^{c_m}_{\bar{x}_m}[\mathbb{G}])])] \end{aligned} \tag{4}$$

In this abstraction function the child designs are replaced by a much simpler connectivity graph. We introduce two functions to support the algorithm:

$$R(A(W^{c_i}_{\bar{x}}[\mathbb{G}])) = W^{c_i}_{\bar{x}_i}[(\|_{x\in\bar{x}_i}\ x) \parallel (\|_{(j,k)\in l_c}\ l_c < j,k >)] \tag{5}$$

$$S(W^p_{\bar{x}}[\mathbb{G}]) = W^p_{\bar{x}}[(\|_h\ x_h) \parallel (\|_{(j,k)\in\lfloor\bar{x}\rfloor}\ l < j,k >) \parallel (\|_i\ R(A(W^{c_i}_{\bar{x}_i}[\mathbb{G}])))] \tag{6}$$

$R(\cdot)$ and $S(\cdot)$ map designs in $\mathbb{G}$ to an abstracted design which only has connectivity edges for each child design. In other words, when analyzing a component of $\mathbb{G}$ we use the incremental interface data for each child component rather than its full details. This is a useful abstraction for cycle detection: we can exploit the graph hierarchy to enumerate simple cycles more efficiently. Figure 2 gives an example of the transformation defined by $A(\cdot)$, $R(\cdot)$, and $S(\cdot)$. The child graph is replaced by its abstracted correspondent, which only preserves connectivity between interface vertices.

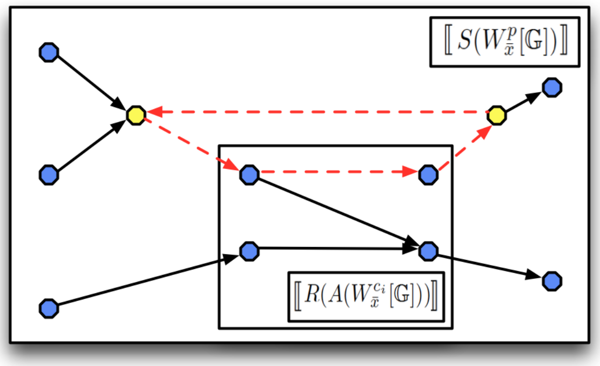
$[\![S(W_{\bar{x}}^{p}[\mathbb{G}])]\!]$
$[\![R(A(W_{\bar{x}}^{c_i}[\mathbb{G}]))]\!]$

Figure 2: Abstract hierarchical graph model

Algorithm Description

Assume we have a function FINDALLCYCLES : $\mathbb{G} \rightarrow \mathbf{2}^{\mathbb{G}}$ which enumerates all elementary cycles in a graph $\mathbb{G}$, returning sets of subgraphs. Then Algorithm 1 adapts the general algorithm FINDALLCYCLES to the hierarchical graph structure described above. We assume that $\mathbb{G}$ has a unique root design, and that we have a function $modified : \mathbb{D} \rightarrow boolean$ which indicates whether a particular hierarchical component has been modified since the last run. New components in the model are considered modified by default.

Algorithm 1 Hierarchical cycle detection

```
1:  $cycles \leftarrow []$
2:  $ifaces \leftarrow \{\}$
3:  function FINDHCYCLES( $[\![W^p_{\bar{x}}[\mathbb{G}]]\!]$ )
4:      for all $W^{c_i}_{\bar{x}_i}[\mathbb{G}] \in W^p_{\bar{x}}[\mathbb{G}]$ do
5:          FINDHCYCLES($[\![W^{c_i}_{\bar{x}_i}[\mathbb{G}]]\!]$)
6:      end for
7:      $modified(W^p_{\bar{x}}[\mathbb{G}]) \leftarrow (modified(W^p_{\bar{x}}[\mathbb{G}]) \vee (\vee_{c_i} modified(W^{c_i}_{\bar{x}_i}[\mathbb{G}]))$
8:      if $modified(W^p_{\bar{x}}[\mathbb{G}])$ then
9:          $T \leftarrow [\![S(W^p_{\bar{x}}[\mathbb{G}]]\!])$
10:         $cycles \leftarrow [cycles;$ FINDALLCYCLES($T$)]
11:         $ifaces[p] \leftarrow A(T)$
12:     end if
13: end function
14: FINDHCYCLES($\mathbb{G}$)
```

- (Lines 1-2) We initialize a list to contain the resulting cycles, and an associative list to contain the interface data.
- (Lines 3-6) The algorithm performs a depth-first search on the hierarchical graph, recursively visiting all of the child components ($W^{c_i}_{\bar{x}_i}[\mathbb{G}]$) for the current (parent) component ($W^p_{\bar{x}}[\mathbb{G}]$) which is currently under evaluation.
- (Line 7) The modification status is propagated up the hierarchy as the algorithm progresses. Each component which has a modified child will also be marked as modified.
- (Lines 8-12) If the current component has been modified, we use the previously computed incremental connectivity interface for each subcomponent to check for cycles in the parent component – the connectivity graph interface is substituted for each subcomponent. The cycles are accumulated as the algorithm ascends to the top of the model, and a connectivity interface is created for the current component as well.

The runtime for the extended algorithm should be slightly worse than Johnson's algorithm in the worst case, as it must also compute the interface graphs. In the average case the cycle checking proceeds on graphs much smaller than the global graph, offsetting the cost of finding paths in each subgraph. Further, if the incremental interface edges are stored in the model following the analysis, then scalability is enhanced when incrementally adding functions to a design. Cycle analysis is then restricted to the size of the new components together with the stored interfaces.

4 ESMoL Language Mapping

Now to map ESMoL logical architecture models onto this cycle-checking formal model we use the following rules:

$$\begin{aligned}
\mathbf{Subsys} &::= L^{Subsys}_{\bar{i},\bar{o}} [\![\mathbf{Dataflow}]\!] \\
\mathbf{Dataflow} &::= \mathbf{0} \mid x \mid l_D < \bar{x} > \mid \mathbf{Subsys} < \bar{x}, \bar{x} > \\
&\quad \mid \mathbf{Dataflow} \parallel \mathbf{Dataflow} \mid (\nu \bar{x})\, \mathbf{Dataflow} \\
\mathbf{MsgType} &::= M_{\bar{e}, e_{ext}} \qquad (7) \\
\mathbf{SysTypeDef} &::= \mathbf{Subsys} < \bar{i}, \bar{o} > \mid l_S < x, x > \\
&\quad \mid \mathbf{SysTypeDef} \parallel \mathbf{SysTypeDef} \mid \mathbf{MsgType} < \bar{x}, y > \\
\mathbf{SysType} &::= L^{Sys}_{\bar{y_i}, \bar{y_o}} [\![(\nu \bar{x})(\nu \bar{e})\, \mathbf{SysTypeDef}]\!] \\
\mathbf{LogicalModel} &::= \mathbf{SysType} < \bar{i}, \bar{o} > \mid \ l_L < o, i > \\
&\quad \mid \mathbf{LogicalModel} \parallel \mathbf{LogicalModel}
\end{aligned}$$

Briefly (from the bottom rule to the top), logical models consist of component blocks (**SysType**) whose interface ports connected by edges. Component blocks are specified by Simulink dataflow blocks (**Dataflow**) whose interface ports are connected either to other Simulink dataflow blocks or to fields in message instances. Each message instance (**MsgType**) inside a system component type block also has an interface vertex (y) which faces outward, and all other vertices are hidden within the component $(\nu\bar{x})(\nu\bar{y})$**SysTypeDef**. At the logical architecture model level, data is exchanged via messages which aggregate the individual dataflow connections within the components. **Dataflow** blocks are built up from connections between functional vertices and between the interfaces on composite subsystem blocks (**Subsys**). These each correspond to sorts in the ESMoL term algebra.

Let i, o, and e be vertex sorts corresponding to input ports, output ports, and message elements respectively. Let s, c, f, and d be edge sorts (of l_D, above) representing signal edges, connectivity edges (as described above to support the incremental interface), f for Simulink primitive function blocks, and d for delay blocks. The f function edge sorts are n-ary, so each function block can have an arbitrary but finite number of input and output connections. For l_S define the sorts (given with their interfaces) $l^{b,b} < o, i >$, $l^{m,b} < e, i >$, and $l^{b,m} < o, e >$. These represent the three different connection types in a **SysType** specification, for connecting between ports of Simulink blocks (from outputs to inputs) ($l^{b,b}$), from message elements to Simulink input ports ($l^{m,b}$), and from Simulink output ports to message elements ($l^{b,m}$).

Finally we give an encoding of terms representing ESMoL models into the more general hierarchical graph algebra:

$$\begin{aligned}
x &= x \\
L^{Subsys}_{\bar{i},\bar{o}}[\![\textbf{Dataflow}]\!] < \bar{x}, \bar{x} > &= L_{\bar{x}}[\![\mathbb{G}]\!] < \bar{x} > \\
M_{\bar{e},e_{ext}}[\![]\!] < \bar{x}, y > &= L_{\bar{x}}[\![\mathbb{G}]\!] < \bar{x} > \\
L^{Sys}_{\bar{i},\bar{o}}[\![(\nu\bar{x})(\nu\bar{e})\textbf{SysTypeDef}]\!] &= L_{\bar{y}}[\![\mathbb{G}]\!] < \bar{x} > \\
l_D < \bar{x} > &= l < \bar{x} > \\
l_* < x, x > &= l < \bar{x} > \\
(\nu\bar{x})\textbf{Dataflow} &= (\nu\bar{x})\mathbb{G}
\end{aligned} \tag{8}$$

The encoding assigns the various layers of hierarchy from the ESMoL component type system to hierarchical designs in the graph. Edges from all layers map to (possibly generalized) edges in the new graph, and ports map to vertices.

The final piece is the application to finding delay-free loops. For a given ESMoL model, simply remove all delay edges (sort elements d). Then invoke the algorithm. For the results, if a cycle is found in a component we can construct a more detailed cycle model by substituting paths from the connectivity edge sort with their more detailed equivalents in the descendants of the component (recursively descending downwards until we run out of cycle elements). Call this subgraph the *expanded cycle*. Repeating the cycle enumeration algorithm on these structures should yield the full set of elementary cycles, and still retain considerable efficiency as we are only analyzing cycles with possible subcycles, which can be a relatively small slice of the design graph.

5 Evaluation

5.1 Fixed-Wing Aircraft Example

Our case study covers cycle analysis of the control design for a fixed-wing aircraft. The Simulink model (Fig. 3) shows the four controller blocks and the sensor data handler. The particulars of the control architecture are not important for this example, but Kottenstette covers them in detail[8]. The controller has five software functions which are specified as Simulink model blocks, and a dynamics component (the Cessna plant block). The **MDL2MGA** model importer creates a structural replica of the Simulink model in the ESMoL modeling language. We use subsystems from the replica to specify the function of synchronous software components. Fig. 4 illustrates one possible configuration of the fixed wing controller components. In this particular configuration (Fig. 4) the entire dataflow is included in one type definition, which means that the entire system will execute together as a single synchronous function with all blocks firing at the same rate. This particular configuration is useful for illustration, but is not the most practical implementation choice.

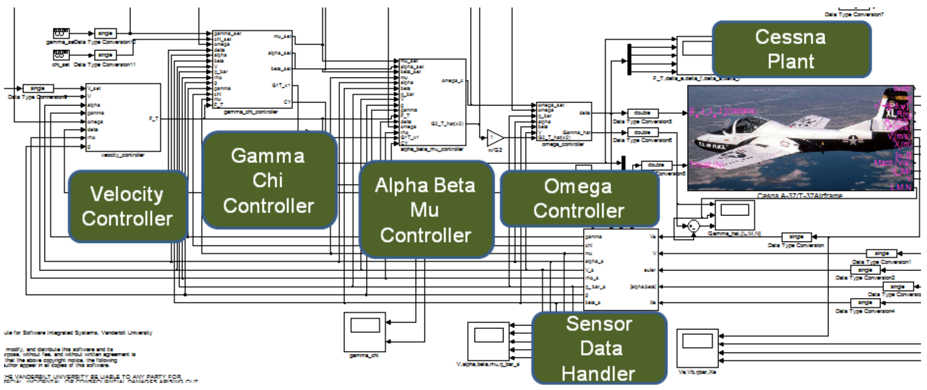

Figure 3: Simulink Fixed Wing Controller Model

5.2 Incremental Analysis Results

Table 1 contains data from the analysis of the fixed wing model. The first pass was performed incrementally, with each subcomponent of the top level model analyzed first. Then the top level is analyzed using the stored path edges in the lower models. The table reports two run times for the analysis of each component – the first is the processing time required to find the abstract cycles only, and the second is the full analysis which finds the expanded cycle for each abstract cycle (enumerating possibly multiple cycles per abstract cycle). The table also displays the number of hierarchical components visited and the number of individual model elements visited, together with the number of abstract cycles found and the total number of cycles. The table row labeled *top level (incremental)* contains the results for the analysis of the top level of the model once the individual path interfaces had been created for each of its subcomponents. The second pass (labeled *top level (full)*) analyzed the entire fixed wing model at once, reporting the same quantities. Our assessment of the scalability of the approach is inconclusive for three reasons – 1) the model size is moderate, so overhead is likely large enough to be a significant factor in all of the run times, 2) we would need a comparison with time taken to process a fully flattened model, including the flattening traversals, and 3) we need to find larger models for our test set. The analyzer found 18 abstract cycles and 54 detailed cycles at the top level for both passes. The *velocity_controller* component also contained a single abstract cycle (consisting of two detailed cycles). Note that we analyzed for all cycles rather than only delay-free cycles to assess scalability. Total runtime was roughly equivalent between the full and incremental methods for this particular model. The results so far are promising but inconclusive as far as improved performance.

Figs. 5 and 6 display a subset of the *velocity controller* component which

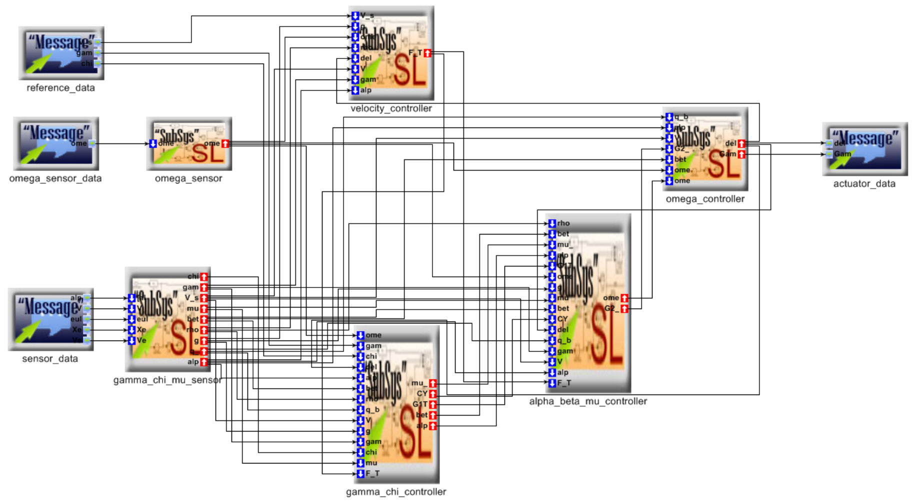

Figure 4: Synchronous data flow for Fixed Wing Controller

contains a cycle, along with the expanded cycle for the component, in order to illustrate the cycle refinement in greater detail. The abstract cycle search discovered the presence of a cycle within the component, but part of the cycle lies within a subcomponent (*anti_windup_control*). The cycle detection for *anti_windup_control* created a single path edge in the interface between the *In1* port and the *Out2* port, which corresponds to two paths within *anti_windup_control*. The full cycle as shown (Fig. 6) is constructed in the analyzer, and then one more pass of Johnson's algorithm resolves the two cycles within the full cycle graph as reported in Tab. 1. The extracted cycle graph is much smaller (13 elements) than the corresponding fully flattened *velocity controller* model, which would contain 60 elements.

6 Conclusion

The current implementation is integrated into the ESMoL tool suite for the Generic Modeling Environment[9], but thorough scalability testing requires larger models.

One interesting observation is the generality of the approach. Algorithm 1 very nearly captures a generic procedure for bottom-up incremental syntactic analysis of hierarchical graphical models. Algorithm 2 proposes such a generic template. A complete study of such generic incremental structural analysis techniques should include consideration of the effects of the component processing order on the accuracy of the result. Note that two small contributions may emerge from this observation 1) we have a structure to which we can adapt

Component	**Abstract Run Time (s)**	**Full Run Time (s)**	**Hier. Comps.**	**Total Elts.**	**Abstract Cycles Found**	**Total Cycles Found**
alpha_beta_mu_controller	0.9	0.9	9	80	0	0
gamma_chi_controller	1.6	1.6	7	134	0	0
gamma_chi_mu_sensor	1.3	1.3	8	100	0	0
omega_controller	0.9	0.9	9	80	0	0
velocity_controller	0.6	0.8	6	60	1	2
Top level (incremental)	2.3	55.1	1	21	18	54
Totals	7.6	60.6			19	56
Top level (full)	7.9	60.5	42	554	19	56

Table 1: Cycle analysis comparisons for the fixed wing model.

some other model analysis techniques for incremental operation, if an appropriate component interface can be found for the particular analysis in question, and 2) this approach could lead to a tool for efficiently specifying such analyses, from which we could generate software code to implement the analysis.

Algorithm 2 Hierarchical cycle detection

```
1: $results \leftarrow []$
2: $ifaces \leftarrow \{\}$
3: **function** ANALYZE( $[\![W^p_{\bar{x}}[\mathbb{G}]]\!]$ )
4:     **for all** $W^{c_i}_{\bar{x}_i}[\mathbb{G}] \in W^p_{\bar{x}}[\mathbb{G}]$ **do**
5:         ANALYZE($[\![W^{c_i}_{\bar{x}_i}[\mathbb{G}]]\!]$)
6:     **end for**
7:     $modified(W^p_{\bar{x}}[\mathbb{G}]) \leftarrow (modified(W^p_{\bar{x}}[\mathbb{G}]) \vee (\vee_{c_i} modified(W^{c_i}_{\bar{x}_i}[\mathbb{G}]))$
8:     **if** $modified(W^p_{\bar{x}}[\mathbb{G}])$ **then**
9:         $T \leftarrow$ ANALYZESTRUCTURE($W^p_{\bar{x}}[\mathbb{G}]$)
10:        $results \leftarrow [results;$ COLLECTRESULTS($T$)$]$
11:        $ifaces[p] \leftarrow$ CREATEINTERFACE($T$)
12:    **end if**
13: **end function**
14: ANALYZE($\mathbb{G}$)
```

Two immediate applications of this generic incremental method in ESMoL embedded control system designs are 1) automated sector analysis for passivity and/or stability [12] and 2) quantization interval analysis for data precision and overflow. Both represent a static analysis of possible system behaviors that can be encoded syntactically and have a natural component interface that can be easily defined. In both cases component interface data requirements are small, and computation is fairly efficient.

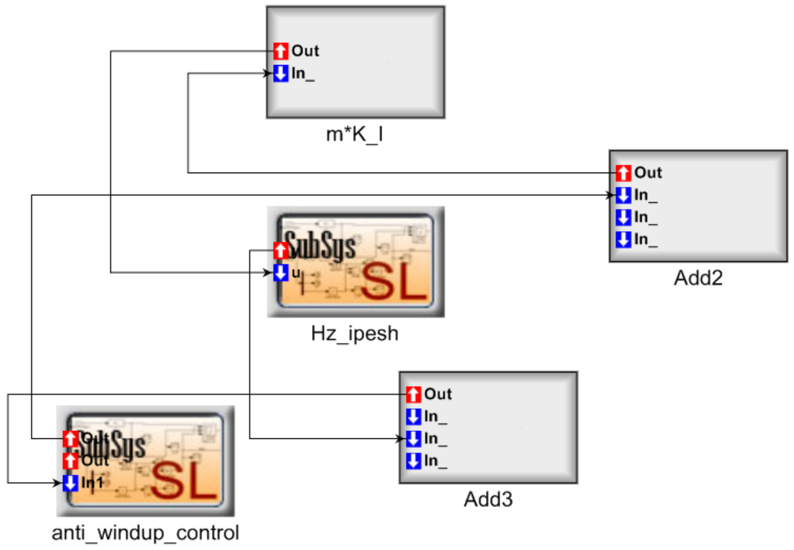

Figure 5: Detail of the components involved in the cycle found in the velocity controller.

7 Acknowledgements

This work is sponsored in part by the National Science Foundation (grant NSF-CCF-0820088) and by the Air Force Office of Scientific Research, USAF (grant/contract number FA9550-06-0312). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

References

[1] Benveniste, A., Caspi, P., di Natale, M., Pinello, C., Sangiovanni-Vincentelli, A., Tripakis, S.: Loosely time-triggered architectures based on communication-by-sampling. In: EMSOFT '07: Proc. of the 7th ACM & IEEE Intl. Conf. on Embedded Software. pp. 231–239. ACM, New York, NY, USA (2007)

[2] Boucaron, J., Coadou, A., De Simone, R.: Throughput and FIFO Sizing: an Application to Latency-Insensitive Design. Research Report

RR-6919, INRIA (2009), `http://hal.inria.fr/inria-00381644/PDF/RR-6919.pdf`, RR-6919

[3] Boucaron, J., de Simone, R., Millo, J.V.: Formal methods for scheduling of latency-insensitive designs. EURASIP J. Embedded Syst. 2007, 8–8 (January 2007), `http://dx.doi.org/10.1155/2007/39161`

[4] Bruni, R., Gadducci, F., Lafuente, A.L.: An Algebra of Hierarchical Graphs and Its Application to Structural Encoding. Scientific Annals of Computer Science 20, 53–96 (2010)

[5] Easwaran, A.: Advances in hierarchical real-time systems: Incrementality, optimality, and multiprocessor clustering. Ph.D. thesis, Univ. of Pennsylvania (2008)

[6] Hemingway, G., Porter, J., Kottenstette, N., Nine, H., vanBuskirk, C., Karsai, G., Sztipanovits, J.: Automated Synthesis of Time-Triggered Architecture-based TrueTime Models for Platform Effects Simulation and Analysis. In: RSP '10: 21st IEEE Intl. Symp. on Rapid Systems Prototyping (Jun 2010)

[7] Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. 4(1), 77–84 (1975)

[8] Kottenstette, N.: Constructive non-linear control design with applications to quad-rotor and fixed-wing aircraft. Tech. Rep. ISIS-10-101, Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN (11 2010)

[9] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., IV, C.T., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. Workshop on Intelligent Signal Processing (May 2001)

[10] Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. of the IEEE 75(9), 1235–1245 (1987)

[11] Mateti, P., Deo, N.: On algorithms for enumerating all circuits of a graph. SIAM J. Comput. 5(1), 90–99 (Mar 1976)

[12] Porter, J., Hemingway, G., Kottenstette, N., Karsai, G., Sztipanovits, J.: Online stability validation using sector analysis. In: EMSOFT '10: Proc. of ACM Intl. Conf. on Embedded Software. Scottsdale, AZ (Oct 2010)

[13] Porter, J., Hemingway, G., Nine, H., vanBuskirk, C., Kottenstette, N., Karsai, G., Sztipanovits, J.: The ESMoL Language and Tools for High-Confidence Distributed Control Systems Design. Part 1: Language, Framework, and Analysis (Sep 2010)

[14] Shin, I.: Compositional Framework for Real-Time Embedded Systems. Ph.D. thesis, Univ. of Pennsylvania, Philadelphia (2006)

[15] Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Professional (Dec 2001)

[16] The MathWorks, Inc.: Simulink/Stateflow Tools. http://www.mathworks.com

[17] Tiernan, J.C.: An efficient search algorithm to find the elementary circuits of a graph. Commun. ACM 13, 722–726 (December 1970), `http://doi.acm.org/10.1145/362814.362819`

[18] Tripakis, S., Bui, D., Geilen, M., Rodiers, B., Lee, E.A.: Compositionality in Synchronous Data Flow: Modular Code Generation from Hierarchical SDF Graphs. Tech. Rep. UCB/EECS-2010-52, Univ. of California, Berkeley (2010)

[19] UCB: Ptolemy II. http://ptolemy.berkeley.edu/ptolemyII

[20] Zhou, Y., Lee, E.: Causality interfaces for actor networks. ACM Trans. on Emb. Computing Systems 7(3) (Apr 2008)

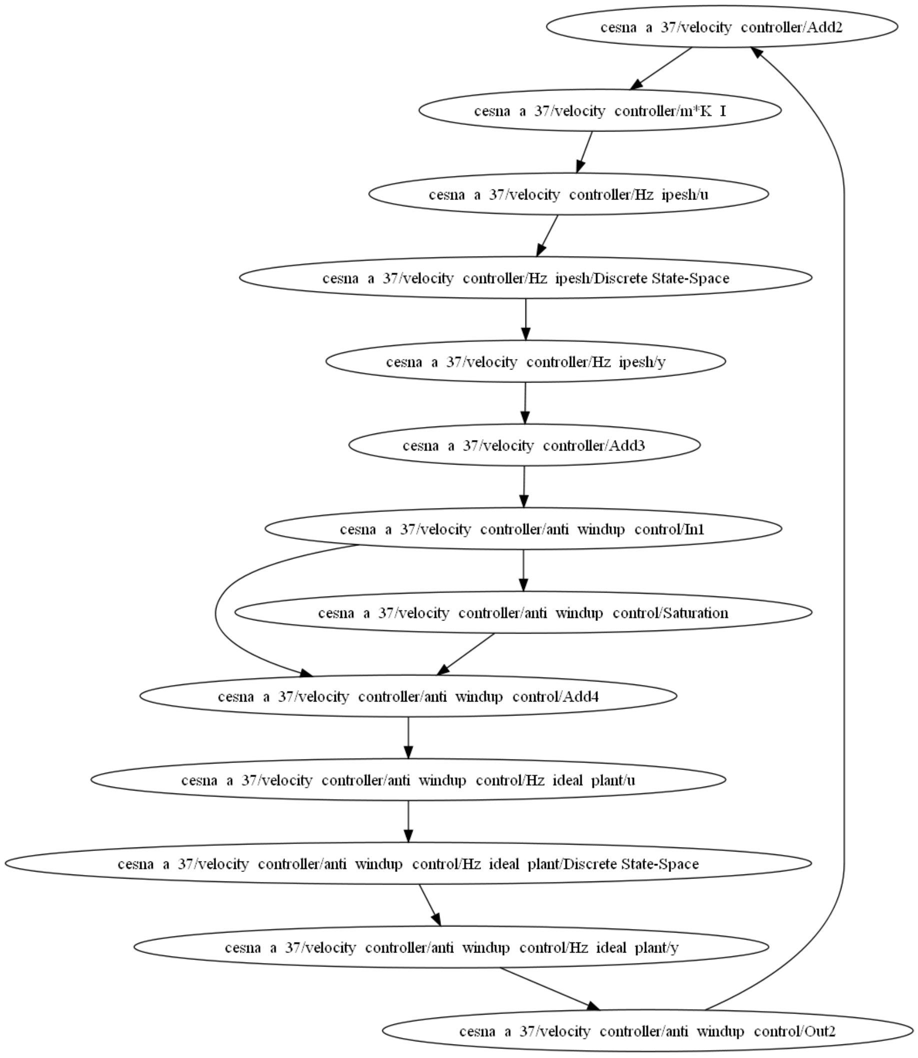
cesna a 37/velocity controller/Add2
cesna a 37/velocity controller/m*K I
cesna a 37/velocity controller/Hz ipesh/u
cesna a 37/velocity controller/Hz ipesh/Discrete State-Space
cesna a 37/velocity controller/Hz ipesh/y
cesna a 37/velocity controller/Add3
cesna a 37/velocity controller/anti windup control/In1
cesna a 37/velocity controller/anti windup control/Saturation
cesna a 37/velocity controller/anti windup control/Add4
cesna a 37/velocity controller/anti windup control/Hz ideal plant/u
cesna a 37/velocity controller/anti windup control/Hz ideal plant/Discrete State-Space
cesna a 37/velocity controller/anti windup control/Hz ideal plant/y
cesna a 37/velocity controller/anti windup control/Out2

Figure 6: Full cycle for the velocity controller.