Towards Incremental Cycle Analysis in ESMoL Distributed Control System Models

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

Institute for Software Integrated Systems,
Vanderbilt University,
Nashville TN 37212, USA,
`jporter@isis.vanderbilt.edu`,
WWW home page: `http://www.isis.vanderbilt.edu`

Abstract. We consider the problem of incremental cycle analysis for dataflow models in the Embedded Systems Modeling Language (ESMoL). This is an example of a syntactic property which does not lend itself to compositional analysis. 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. An extended version of this work gives 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[7]. Integrated cycle analysis aids well-formedness checking during model construction in the ESMoL tool suite.

Keywords: model-based design, embedded systems, model analysis

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 embedded software design language is built on a platform which provides inherent correctness properties for well-formed models. We 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[9] – 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[3]. 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 [5]. This condition can be encoded structurally into dataflow modeling languages – for example Simulink [11] analyzes models for algebraic loops and attempts to resolve them analytically. 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 for ESMoL:

– The algorithm uses Johnson's simple cycle enumeration algorithm as its core engine[4]. Johnson's algorithm is known to be efficient [6]. We use cycle enumeration in order to provide detailed feedback to 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 connectivity 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[3]. 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. [14]. Our method traverses the entire model and extracts all elementary cycles, reporting all ports and subsystems involved in each 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[13]. Their approach 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.

2 Background

A number of frameworks and techniques contributed to our solution:

ESMoL Component Model: As the ESMoL language structure is documented elsewhere[8], 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. ESMoL and its tools provide design concepts and functions for specifying logical architecture and deployment, and performing scheduling analysis.

In ESMoL a designer can include Simulink references from parts of an imported dataflow model, and instantiate them any number of times within the type definitions. ESMoL 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.

Cycle Enumeration: To implement cycle enumeration we use the algorithm Johnson proposed as an extension of Tiernan's algorithm [12] for enumerating elementary cycles in a directed graph[4]. 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 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 considered the best available general cycle enumeration method[6]. We implemented Johnson's algorithm in C++ using the Boost Graph library [10].

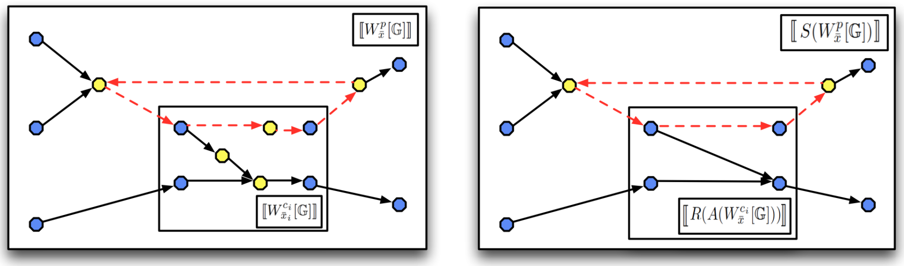
Hierarchical Graphs: As a notation for describing our incremental approach we use the algebra of hierarchical graphs introduced by Bruni et al[2]. We will only give a summary of some of the notation here for brevity. The interested reader can refer to [7] (and [2]) for a more detailed account.

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

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[2]), 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 rule $\mathbb{D}$ corresponds to composite types in our dataflow language (which may have other composites as children). The specification for a composite element is $L_{\bar{x}}[\mathbb{G}]$, which means that an element of $\mathbb{D}$ has type L and 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 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. Finally $[\![\mathbb{G}]\!]$ indicates the graph corresponding to the expression $\mathbb{G}$.

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.


$[\![W^p_{\bar{x}}[\mathbb{G}]]\!]$
$[\![W^{c_i}_{\bar{x}_i}[\mathbb{G}]]\!]$
$[\![S(W^p_{\bar{x}}[\mathbb{G}])]\!]$
$[\![R(A(W^{c_i}_{\bar{x}}[\mathbb{G}]))]\!]$

(a) Generic hierarchical graph (b) Abstract graph

Fig. 1: Simple graph example

Let $\mathbb{G}$ be a well-formed hierarchical graph (as in Bruni [2]). To get more comfortable with the notation, first note that graph $\mathbb{G}$ itself (without hierarchical structure) can be described by the expression:

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

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 set of components of type W. Let $W^p_{\bar{x}}[G]$ represent a component 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 ($[\![W^{c_i}]\!] \subset [\![W^p]\!]$) with their respective interface vertices ($\bar{x}_i$). Then neglecting vertex hiding and renaming to simplify the illustration, we have the following:

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

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

We introduce a new type of edge l_c into the model, which is used to connect vertices at the boundaries of a component, abstracting the interface connectivity of the component. Introduce a new mapping $A : \mathcal{D} \to \mathcal{D}'$ from the components of $\mathbb{G}$ to components 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.

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

In this abstraction function the child components 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 components in $\mathbb{G}$ to an abstracted component which only has connectivity edges for each child component. 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 1b 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 in the child component.

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 component, 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.

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.

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) Initialize a list to hold the resulting cycles, and an associative list to contain the component interface data.
- (Lines 3-6) Perform 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 ($W^{p}_{\bar{x}}[\mathbb{G}]$).
- (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 current 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 before returning.

4 ESMoL Language Mapping

To find delay-free loops for a given ESMoL model, we must first map a well-formed ESMoL model to the generic hierarchical graph model (as in [7]), remove all delay elements from the model, and then invoke the algorithm. For any cycle found in a component we can construct a more detailed cycle by substituting paths using the connectivity edges 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 yields the full set of elementary cycles, and should still retain considerable efficiency as we are only analyzing cycles with possible subcycles, which can be a relatively small slice of the design graph. [7] includes an example.

5 Conclusion

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. Note that two small contributions may emerge from this observation 1) we have a structure to which we can adapt 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.

6 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).

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. 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)
3. Easwaran, A.: Advances in hierarchical real-time systems: Incrementality, optimality, and multiprocessor clustering. Ph.D. thesis, Univ. of Pennsylvania (2008)
4. Johnson, D.B.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. 4(1), 77–84 (1975)
5. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. of the IEEE 75(9), 1235–1245 (1987)
6. Mateti, P., Deo, N.: On algorithms for enumerating all circuits of a graph. SIAM J. Comput. 5(1), 90–99 (Mar 1976)
7. Porter, J., Balasubramanian, D., Hemingway, G., Sztipanovits, J.: Towards Incremental Cycle Analysis in ESMoL Distributed Control System Models (Extended Version) (Apr 2011), `http://www.isis.vanderbilt.edu/sites/default/files/incr_cycle_analysis.pdf`
8. 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), `http://www.isis.vanderbilt.edu/sites/default/files/ESMoL_TR.pdf`
9. Shin, I.: Compositional Framework for Real-Time Embedded Systems. Ph.D. thesis, Univ. of Pennsylvania, Philadelphia (2006)
10. Siek, J.G., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Professional (Dec 2001)
11. The MathWorks, Inc.: Simulink/Stateflow Tools. http://www.mathworks.com
12. 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`
13. 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)
14. Zhou, Y., Lee, E.: Causality interfaces for actor networks. ACM Trans. on Emb. Computing Systems 7(3) (Apr 2008)