Efficient Data-Flow Analysis of UML/SysML Diagrams for Optimized Model Compilation of Hardware-Software Systems

Keywords: Model-Driven Engineering, Static Data-Flow Analysis, UML, SysML, Optimizing Model Compilation.

Abstract: Growing needs in terms of latency, throughput and flexibility are driving the architectures of tomorrow's Radio Access Networks towards more centralized configurations that rely on cloud-computing paradigms. In these new architectures, digital signals are processed on a large variety of hardware units (e.g., CPUs, Field Programmable Gate Arrays, Graphical Processing Units). Optimizing model compilers that target these architectures must rely on efficient analysis techniques to optimally generate software for signal-processing applications. In this paper, we present a blocking combination of the iterative and worklist algorithms to perform static data-flow analysis on functional views denoted with UML Activity and SysML Block diagrams. We demonstrate the effectiveness of the blocking mechanism with reaching definition analysis of UML/SysML models for a 5G channel decoder (receiver side) and a Software Defined Radio system. We show that significant reductions in the number of unnecessary visits of the models' control-flow graphs are achieved, with respect to a non-blocking combination of the iterative and worklist algorithms.

1 INTRODUCTION

The evolution of current networks towards their fifth generation (5G) is dominated by considerable increases in network traffic (10x higher data rates are expected) and in the flexibility required to answer to variations in network services and performance. These two aspects are expected to significantly impact the architecture of Radio Access Networks (RANs). A promising evolution of RAN architectures is the so-called Cloud-RAN [Checko et al., 2015] that consists in moving some signal processings from a set of geographically distributed base stations to the core network. From a programmer's perspective, this implies that signals will be processed by a greater variety of platforms: from Application-Specific Integrated Circuits (ASICs), in base stations, to cloud systems equipped with both programmable and configurable components (CPUs, Digital Signal Processors - DSPs, Field Programmable Gate Arrays - FPGAs), in the core network.

The high complexity of these platforms raises the need for novel programming paradigms, such as those based on Model-Driven Engineering (MDE) [Schmidt, 2006, Selic, 2003]. As of today, the process of generating optimized implementations (i.e., hardware, software or both) from models is still an open issue. Because of the abstraction level at which they operate, modeling languages, such as UML/SysML, can express more complex control-flow interactions (e.g., hierarchical composition, dispatch/reception of signals) than traditional programming languages (e.g., functions in C). In the context of optimizing model compilers, this raises the need for novel static analysis techniques. In this paper, we present an algorithm that reduces the number of unnecessary visits due to the propagation of partial information to Control-Flow Graphs (CFGs) of functional views expressed with UML Activity diagrams, SysML Block Definition and SysML Internal Block diagrams. We demonstrate the efficiency of our algorithm for the reaching definition analysis of models denoted in DIPLODOCUS [TTool/DIPLODOCUS, 2006], a UML/SysML profile for the hardware/software co-design of embedded systems.

In Section 2 we position our contribution with respect to related work. Section 3 outlines our approach for model compilation. Section 4 illustrates our contribution. Section 5 describes the analysis of UML/SysML functional views for two telecommunication systems. Section 6 concludes this paper.

2 RELATED WORK

Static data-flow model analysis is inspired by program analysis techniques [Nielson et al., 2010] and encompasses solutions for reasoning about the value and relations (e.g., definitions, use) of data (e.g., Variables, Objects) that influence the execution of models, without actually running them. In the

context of UML, only behavioral diagrams (State Machines, Activity and Sequence diagrams) are eligible for static analysis. To the best of our knowledge, we appear to be the first to propose the use of static data-flow analysis on UML Activity diagrams and their combination with SysML Internal Block and Block Definition diagrams. Similarly, we found no related work that applies this type of analysis to optimizing model compilers.

The relevance of data-flow analysis on models is evident from the amount of work presented at international conferences (e.g., MODELS, MODELSWARD). It can be assumed that work such as [Saad and Bauer, 2013, Schwarzl, C. and Peischl, B., 2010, Yu et al., 2008, Kienberger et al., 2014, Lai and Carpenter, 2013] could profit from the algorithm in this paper to efficiently propagate the results of data-flow equations in their respective domains.
Most related work are based on the analysis of Statecharts for software testing [Kim et al., 1999, Briand et al., 2005, VERIMAG, 2018]. The authors in [Kim et al., 1999] discuss the generation of test cases, given a set of criteria to be tested, from UML State Machines. This generation is driven by data-flow analysis that identifies the pairs of definitions and uses of variables. The analysis is conducted on a control-flow graph that is retrieved by transforming the diagrams into Extended Finite State Machines where hierarchical and concurrent states are flattened. With respect to our work, communications between Classes are not considered and broadcast communications are eliminated when transforming Statecharts in Extended Finite State Machines. In [Briand et al., 2005], the main contribution is a technique that guides the coverage of UML Statecharts for test data selection in the context of fault detection. The proposed technique allows to select the best cost-effective data structure (a transition tree) based on definition-use pairs of variables. While our control-flow graph is entirely derived by UML/SysML diagrams, the authors in [Briand et al., 2005] use a special Event Action Flow Graph that represents events and actions only, where operation contracts and guard conditions are expressed in the Object Constraint Language.
The IF toolset [VERIMAG, 2018] is an environment for the modeling and validation of heterogeneous real-time systems that is built upon an intermediate representation formalism, called IF. The toolset includes a translator for input UML Statecharts and Class diagrams and a static analyzer. The latter operates on IF representations and supports live variable analysis, dead code elimination and variable abstraction. In IF, the main difference with respect to our contribution is that the functionality captured by input models is executed by software implementations: mixed hardware/software or purely hardware implementations are not considered.
The work in [Yu, 2014] describes a static analysis technique to analyze UML Class diagrams that include operations specified using the Object Constraint Language (OCL) [OMG, 2014]. The structure of a software project is captured with UML Class diagrams that are investigated against a set of scenarios representing some desired or undesired behaviors. This work addresses the needs of verification engineers rather than software developers.

Following the standardization of the Foundational Subset for Executable UML, fUML, [Seidewitz, 2014, fUML, 2018], many work analyze fUML specifications of software implementations. In [Malm et al., 2018] static *program* analysis is applied to its textual action language Alf [OMG, 2018]. The authors introduce a round-trip transformation chain that applies flow analysis to Alf specifications and back-propagates the results of this analysis to Alf programs for further investigation. The objective of their analysis is to retrieve information about loop bounds and infeasible paths in a model to estimate a worst-case execution time. In [Malm et al., 2018] a model's execution semantics influences both the construction of the Control-Flow Graph and the algorithm that visits it. On the contrary, in our work, the visit algorithm only depends on the graph's topology.
The authors in [Waheed et al., 2008] propose an approach to build a data structure that identifies all the associations between definitions and use (DU) of variables within states of an input UML State Machine. Statecharts are specified with the abstract syntax of the UML Action Specification Language [Mellor and Balcer, 2002]. An input Statechart is parsed, its control flow graph is extracted and stored in an adjacency matrix that is traversed to identify all the DU pairs. The authors also propose mapping rules that allow their approach to be reused with virtually any concrete syntax of the UML Action Specification Language. However, no effective analysis is proposed nor applied on the DU pairs (e.g., dead "code" elimination).

The work in [Aldrich, 2002] performs coverage analysis on MATLAB state diagrams in order to establish completeness and consistency with respect to design requirements. It forms the core of the Model Coverage Tool that is commercially available in the

Simulink Performance Tools developed by the Mathworks Inc. For each state diagram's block, the author retrieves the control flow of behaviorally equivalent implementation code. When modeling constructs do not have a unique code implementation, the author suggests to choose a coverage requirement that guarantees full coverage in all of the likely implementations. A fundamental difference between our work and [Aldrich, 2002] is that the latter considers analysis and coverage techniques after models have been translated in code, not as part of the code generation process itself. This can lead to discrepancies between the model's behavior and its implementation code due to optimizations performed by the code generation engine (e.g., inlining, dead code elimination).

3 MODEL COMPILATION

The methodology that we follow to generate optimized software from executable models at Electronic System Level of abstraction [Gerstlauer et al., 2009] is shown in Fig. 1. In the context of our research, we develop control software that executes as an application in the user-space of a Real-Time Operating System. This software governs the execution of data processing and transfer operations that can be implemented as both hardware and/or software modules. For this reason, we model a system with a combination of UML/SysML diagrams, rather than UML only. With respect to the C programming language[1], UML/SysML diagrams express parallelism explicitly. They offer richer constructs than concurrent languages (e.g., Synchronous Data Flow (SDF) [Lee and Parks, 1995] and Kahn Process Networks (KPN) [Kahn, 1974]) that do not capture the internal behavior of computations and communications. In Fig. 1, input specifications are created in DIPLODOCUS [TTool/DIPLODOCUS, 2006], step (1). Here, a system is captured in terms of its functionality (i.e., behavior), the architecture of its target platform (i.e., the services and topology of available resources) and the communication protocols (e.g., DMA transfers). In this phase, models are used as the primary artifact for software development. They are created, edited and debugged (e.g., formal verification, simulation, profiling) until legal specifications are obtained that respect some desired constraints (e.g., throughput, latency, power consumption). This is similar to the way code is created, edited and debugged in Integrated Development Environments (IDE) such as Eclipse CDT [Eclipse CDT, 2018].

Subsequently, model-based specifications are compiled into C code, step (2) in Fig. 1, by an optimizing compiler. The structure of the latter is inspired by those for programming languages [Torczon and Cooper, 2007] and includes a front-end for parsing and analysis, a middle-end for optimization and a back-end for code generation. To target Cloud-RAN systems, our model compiler is designed for multiprocessor architectures with heterogeneous computation, communication and storage units that can be both shared or distributed. At the output of the model compiler in Fig. 1, code becomes the primary artifact for software development as in classical software engineering. We specify to the reader that the control software generated by the compiler does not include the algorithmic part of computations and communications. For this part, we rely on external platform-specific libraries (e.g., I/O specific code, platform-specific code for OS or middleware).

The desired implementation is produced by means of a final translation, step (3) in Fig. 1. This implementation can be realized entirely in software (e.g., an application running on top of an Operating System) or in hardware (e.g., a hardware IP-based design) or both (e.g., some functionalities are executed by a general-purpose control processor and some are accelerated in hardware). Different *translators* must be used accordingly: Computer Aided Design (CAD) toolsuites (e.g., Xilinx Vivado High Level Synthesis) or traditional programming-language compilers (e.g., GNU/gcc/g++, Clang).

4 STATIC MODEL ANALYSIS

In this section, we propose a framework for solving a large class of data-flow analysis problems (e.g., reaching definitions, available expressions, live variables) for functional views expressed with UML Activity (AD) and SysML Block diagrams (i.e., SysML Block Definition and Internal Block diagrams - shortened to BDs). This framework is implemented in the optimizing model compiler's frontend of Fig. 1.

From the viewpoint of program analysis techniques, references to UML Activities via InvocationActions resemble the way procedures interact in the C language. Thus, existing techniques for program interprocedural analysis [Reps et al., 1995, Jhala and Majumdar, 2007] can be reused to examine both synchronous and asynchronous invocations of Activities. However, the execution semantics of an Activity corresponds to that of a whole C program rather than

[1] As C is the most widely used programming language for the development of signal-processing applications, we consider it the reference to which we compare our research.

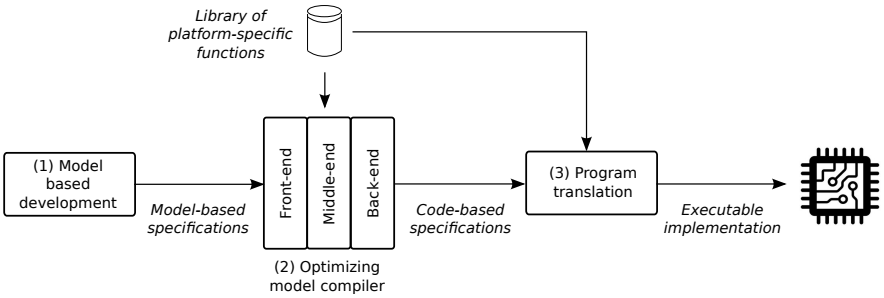
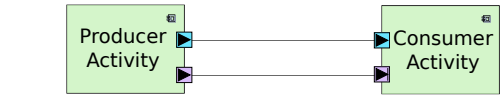


Figure 1: The software development flow of executable implementations from system-level models.

a single procedure. Novel techniques are needed to efficiently analyze the effects of modeling constructs for the exchange of data among Activities such as SendObjectActions and ReceiveObjectActions. These Actions result in numerous interactions among CFGs that increase the amount of information to be propagated when analyzing models. This is especially the case when data is exchanged through the Ports of SysML BDs. As the rules to exchange data through Ports can be specified by dedicated ProtocolStateMachines, a sound and precise analysis framework must include the CFGs corresponding to these protocols.

4.1 The Control-Flow Graph Creation

The CFG that results[2] from the composition of UML ADs and SysML BDs is a directed graph $G^* =< N^*, E^* >$. G^* is a supergraph that consists of a set of control flowgraphs $N^* = \{G_1, G_2, ..., G_n\}$. In each graph $G_i =< N_i, E_i >$, nodes N_i are the modeling constructs of an Activity and edges E_i are the Activity's ControlFlowEdges. One of these flowgraphs, G_{source}, represents the source Activity that injects samples and is the functional view's entry point[3]. At least one sink node is also present, G_{sink} that collects the samples that have been processed. E^* is the set of superedges that correspond to Relationships among SysML Blocks (the control flowgraphs in N^*).
Each Activity's CFG G_i has a unique start node (i.e., UML InitialNode) and can have multiple exit nodes (i.e., UML ActivityFinalNode and FlowFinalNodes). Remaining nodes represent the modeling statements (e.g., Actions) and predicates of an Activity (e.g., ControlNodes). In addition to the ordinary *intra*-graph edges that connect the nodes within such a CFG, special *inter*-graph edges are created for each pair SendObjectAction-ReceiveObjectAction. Here, we distinguish two cases according to the presence or absence of a ProtocolStateMachine. In case data is *not* exchanged through a Port or is exchanged through a Port that lacks protocol specifications, an asynchronous call edge is added, in the CFG, from the SendObjectAction's node to its matching ReceiveObjectAction's node. In case of the presence of a ProtocolStateMachine, instead, we add the protocol's CFG to G^* and connect it to the caller Activity's CFG by means of a pair of *synchronous* call and return edges. A return node is also added to the caller Activity's CFG as the immediate successor of the calling node, in order to retrieve the exchanged data. The resulting CFG is similar to the one of a C program with synchronous procedure calls and allows to reuse techniques from interprocedural program analysis.




```
producer_DMATransfer( _numSamples, _data[], _srcAddress, _dstAddress )
{
  array[] <- _data[];
  counter = _numSamples;                    B1
  j = 0;
  for( i = counter; i > 0; i-- ){
    DMA.transfer( array[ j ] );             B2
    j++;
  }
  return;
}

consumer_readFromDMA( _numSamples )
{
  i = 0;
  counter = _numSamples;                    B3
  while( counter > 0 ) {
    array[ i ] = DMA.read();
    i++;                                    B4
    counter--;
  }
  return array[];
}
```

Figure 2: The SysML BD for a pair of Activities (upper part) and the pseudo-code of the Ports' ProtocolStateMachines for a DMA transfer (lower part).

By way of example, the upper part of Fig. 2 shows the SysML BD for a functional view where a pair

[2]We do not describe how to create a CFG from the graph of a UML AD. Thanks to the separation between Tokens and Edges of different types, a CFG can be obtained by visiting the AD's graphical representation and filtering out undesired nodes and edges.

[3]This is similar to the CFG for the `main()` procedure in the C programming language.

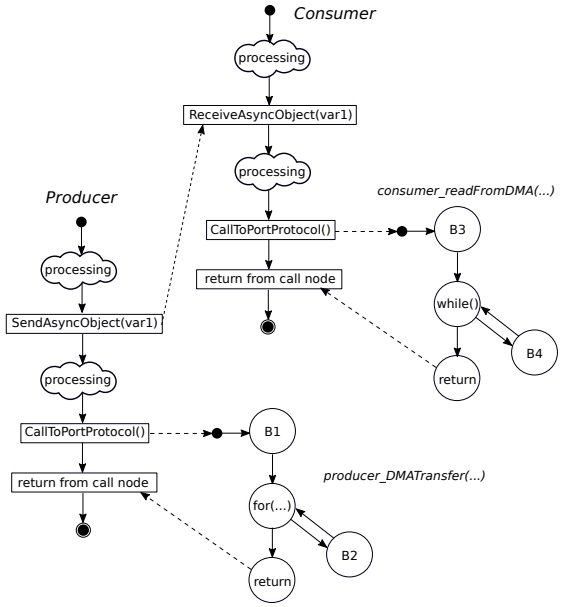


Figure 3: The CFGs for the Activities and ProtocolStateMachines in Fig. 2. Nodes B1-B4 correspond to the code snippets highlighted in gray in Fig. 2.

of producer-consumer Activities (inside the Blocks) exchange data through Ports. The blue Ports in Fig. 2 exchange data via a DMA transfer and make use of ProtocolStateMachines whose pseudo-code is shown below the diagram. The exchange of data on purple Ports in Fig. 2, instead, uses no ProtocolStateMachine. Fig. 3 shows the CFGs for the Activities (rectangular nodes) and the ProtocolStateMachines (circular nodes). In Fig. 3, dotted edges represent inter-Activity dependencies, whether synchronous or asynchronous. The dotted edge between `SendAsyncObject` and `ReceiveAsyncObject` corresponds to the Relationship between the purple protocol-less Ports in Fig. 2. Dotted lines are also used to represent the synchronous call and return edges between and Activity's CFG and the ProtocolStateMachine's CFG of its associated Port. For the sake of simplicity, in Fig. 3 we abstracted modeling statements that do not reference Activities or exchange data between Activities with cloud-shaped nodes.

4.2 The Control-Flow Graph analysis

Static analysis is computed by propagating data-flow information (facts) along the CFG's edges according to the edges' transformation functions that account for the semantics of nodes. Visitation algorithms stem from two common approaches: the iterative search and the worklist algorithms. In the iterative search (Algorithm 1), each node is visited once. If any changes occur in the output information to be propagated, then dependent nodes are visited iteratively until there are no further changes. In the worklist visit (Algorithm 2), all the edges are stored in a list. An edge is popped out and information propagated to its destination node: if any changes occur then its successors[4] are pushed into the list. This exploration repeats until the worklist is empty.

The worklist algorithm immediately propagates changes to neighboring nodes by pushing their edges into the list and examining them in the next iteration. However, a complete visitation of all nodes may require multiple visits of the same node before new nodes are considered. On the contrary, the iterative search always visits nodes once but it waits until the next visitation of the entire CFG to propagate a change.

Algorithm 1: The iterative search algorithm

```
1  changed = true;
2  while changed do
3  |   changed = false;
4  |   for ∀node n do
5  |   |   old = out[n];
6  |   |   process(n);
7  |   |   if old ≠ out[n] then
8  |   |   |   changed = true;
9  |   |   end
10 |   end
11 end
```

Algorithm 2: The worklist algorithm

```
1  worklist ← {start edge};
2  while worklist ≠ ∅ do
3  |   worklist ← worklist \ e;
4  |   old = out[e];
5  |   process(e);
6  |   if old ≠ out[e] then
7  |   |   for p ∈ succ[e] do
8  |   |   |   worklist ← worklist ∪ p;
9  |   |   end
10 |   end
11 end
```

However, the eagerness of the worklist algorithm may yield poor performance in case of inter-Activity analysis. In case of context *unsensitive* analysis, the nodes

[4] We always imply forward analysis. Predecessors must be considered in the case of backward analysis.

of an Activity's CFG are shared among different Activity's references and among inter-Activity dependencies. This results into the nodes of an Activity being visited multiple times and partial information being propagated. Ultimately, this leads to an increase in the analysis running time and processing resources. A similar issue is described in [Atkinson and Griswold, 2001] for program analysis.

To overcome this issue, Algorithms 1-2 can be combined as in [Atkinson and Griswold, 2001]. The iterative search has a more global nature in that, at each iteration, it computes data flows for all CFG's nodes. This makes it a suitable candidate to direct visitations of the entire supergraph G^*. The worklist algorithm, instead, has a more local nature as it propagates data flows locally to a node's successors only (predecessors in case of backward analysis). This makes it an ideal candidate to visit single Activities' CFGs.

Nonetheless, because of data flows from inter-Activity dependencies, this combination does not perform well enough for the analysis of UML ADs and SysML BDs. The iterative algorithm must not propagate local changes from a previous iteration to all nodes in G^*. Similarly, at each visitation, the worklist algorithm should explore a node's successors or predecessors only when information from *all* its incoming edges is available (i.e., information on both inter- and intra-Activity edges).

As a solution, in Algorithms 3-4 we propose a Combined Iterative Blocking Worklist (CIBW) search. In Algorithm 3, a first blocking worklist exploration of all graphs in G^* starts, lines 8-10. Subsequently, lines 12-30, blocked Activities are iteratively visited until no changes occur when data-flow information is propagated. Lines 12-30 in Algorithm 3, describe the iterative search on (super)nodes of G^* at the level of abstraction of the whole supergraph. Here, each node in G^* is processed only if the data-flow information of any of its successors (predecessors in the case of backward analysis) has changed as indicated by a set of pending graphs P. Each node is visited exactly once on each iteration (lines 9 and 16) in order to retain the fairness of the original iterative approach. Therefore, an Activity is not visited again before another pending Activity is visited.

An Activity's CFG is visited by a blocking version of the worklist search, in Algorithm 4. Here, a worklist of edges is created, line 2, from the set of unvisited intra-Activity and inter-Activity nodes. An edge is denoted as $e(\ n,m\ )$, where n and m are the producer and consumer nodes, respectively. At lines 4-14, exploration proceeds like in the classical worklist search (Algorithm 2). It is suspended at lines 16-19, hence the name *blocking*, in case the source node of an inter-Activity edge has not been visited yet. The Activity being analyzed is added to the pending list P. Upon completing the analysis, the Activity is removed from the pending list P, line 22. Nodes that belong to the predecessors of this Activity are marked as unvisited, line 23, if they have already propagated information to all their successors.

To avoid deadlocks due do cycles in the supergraph G^*, the blocking mechanism (line 16 in Algorithm 4) does not activate on unvisited dependencies that originate from Activities whose distance from G^*'s source is greater than the one of the currently visited Activity. This distance is computed at line 3 in Algorithm 3 by measuring the shortest path.

Algorithm 3: The CIBW algorithm

```
   Input             : G* =< N*, E* >
   Global parameters: visited[], analysis[], P
   Output            : analysis[ ]
 1 foreach n ∈ N* do
 2 |   analysis[ n ] = ⊥;
 3 |   compute_distance( n );
 4 |   foreach node ∈ n do
 5 |   |   visited[ node ] = false;
 6 |   end
 7 end
 8 foreach n ∈ N* do
 9 |   blocking_worklist( n );
10 end
11 changed = true;
12 while changed do
13 |   changed = false;
14 |   foreach p ∈ P do
15 |   |   old = analysis[ p ];
16 |   |   blocked = blocking_worklist( p );
17 |   |   if blocked then
18 |   |   |   changed = true;
19 |   |   end
20 |   |   else
21 |   |   |   if old ≠ analysis[ p ] then
22 |   |   |   |   changed = true;
23 |   |   |   |   foreach s ∈ succ( p ) do
24 |   |   |   |   |   P ← P ∪ s;
25 |   |   |   |   |   mark_nodes_as_unvisited( s );
26 |   |   |   |   end
27 |   |   |   end
28 |   |   end
29 |   end
30 end
```

Algorithm 4: The blocking worklist algorithm

```
1  Function blocking_worklist( Activity CFG b = <N_b, E_b> ):
2      worklist = create_worklist();
3      while !empty( worklist ) do
4          e( n, m ) ← worklist.pop();
5          if n ∈ N_b, m ∈ N_b then
6              if trans_n( analysis[ n ] ) ⋢ analysis[ m ] then
7                  analysis[ m ] ← analysis[ m ] ⊔ trans_n( analysis[ n ] );
8                  visited[ n ] = true;
9                  visited[ m ] = true;
10                 foreach p ∈ N_b, p ∈ succ{ m } do
11                     worklist.push( e( m, p ) );
12                 end
13             end
14         end
15         else
16             if n ∈ N_b', m ∈ N_b, b' ≠ b, distance(N_b') ≤ distance(N_b), visited[ n ] == false then
17                 P ← P ∪ b;
18                 return true;
19             end
20         end
21     end
22     P ← P \ b;
23     mark_predecessors_as_unvisited( b );
24     return false;
25 End function
```

4.2.1 Performance gain

In the CFGs that we consider, data exchanges that are associated to ProtocolStateMachines are equivalent to procedures in traditional program analysis. Thus, we evaluate the gain of the CIBW algorithm when Ports are *not* associated to ProtocolStateMachines (e.g., edge `SendAsyncObject(var1) - ReceiveAsyncObject(var1)` in Fig. 3). In Eq. 1, this is given by the ratio between the number of visits of the CFG's nodes $\mathcal{N}$.

$$g = 1 - \frac{\mathcal{N}^{blocking\ worklist}}{\mathcal{N}^{non-blocking\ worklist}} = 1 - \frac{\mathcal{N}^{bw}}{\mathcal{N}^{nbw}} \quad (1)$$

This gain can be expressed analytically only for graphs with a fixed topology (see Section 5). Nevertheless, a generic gain can be expressed, Eq. 2, in terms of the unnecessary number of visits $\mathcal{N}^u$ that are performed by the non-blocking worklist for each node n that receives an inter-Activity edge. Unnecessary visits are those that propagate partial information without considering updates from inter-Activity edges. $\mathcal{N}^u$ is zero in two cases. If n has no successors or if no path exist, from the the Activity's InitialNode to n, whose nodes operate on the same data set (Variables and/or Objects) as n, D_n. In all other cases, $\mathcal{N}^u$ is different from zero and depends on two factors: (i) the number of n's successors that operate on D_n and (ii) the type of paths (acyclic or cyclic) that these successors belong to.

$$g = \frac{\mathcal{N}^u}{\mathcal{N}^{bw} + \mathcal{N}^u} \quad where\ \mathcal{N}^{nbw} = \mathcal{N}^u + \mathcal{N}^{bw} \quad (2)$$

The value of $\mathcal{N}^u$ is given by Eq. 3, for the successors (predecessors) of a node n that receives an inter-Activity edge. These successors (predecessors) are visited either once, if they belong to a linear path, or k_p times, one per each iteration, if they belong to a cyclic path. The coefficient k_p is defined by the number of iterations that are necessary to reach the analysis' fixed point (e.g., fixed point in a lattice).

$$\mathcal{N}^u = \sum_{\forall\ path\ p\ \in\ CFG,\ i\ \in\ p} v_i^p \qquad v_i^p = \begin{cases} 1 & D_i = f(D_n),\ i \notin cycle \\ k_p & D_i = f(D_n),\ i \in cycle \end{cases} \quad (3)$$

In Eq. 3, i indexes nodes n's successors (predecessors), D_i denotes the data set on which the i-th node operates and D_n the data set onto which n operates. A path p is defined as a succession of nodes that starts either at the Activity's InitialNode or at node n. A path p can terminate at an ActivityFinalNode or at a FlowFinalNode or at n itself or at any other node m that receives a different inter-Activity edge. From this definition and from Eq. 3, it follows that the number of unnecessary visits on a given path p, $\mathcal{N}_p^u$, is comprised between L_p, in case p is a linear path, and $k_p \times L_p$, in case p is cyclic, where L_p is the number of nodes in p that operate on the data set D_n. The total number of unnecessary visits is given by the sum on all paths p, recursively if nested paths are present in the Activity's CFG.

4.2.2 Discussion

The CIBW algorithm and the supergraph G^* constitute a framework that produces sound and precise results for the class of locally-separable problems (also called "bit-vector" or "gen/kill" problems) such as

reaching definitions, available expressions and live variables. It can be used for the analysis of a composition of UML ADs, regardless of the presence of a SysML BD. It can be reused in other similar languages provided some conditions are met: (i) the absence of global variables and (ii) a pass-by-value mechanism for the exchange of Objects and Parameters among Activities. If these conditions are met, our framework also extends to profiles that allow synchronous invocations of Activities. In this case, valid paths in G^* that result from matching invocation-return pairs can be analyzed by standard *meet over all valid paths* (MVP) techniques from program analysis.

In case the above conditions do not hold, an engineer wishing to reuse our framework must (i) separate the analysis of global data from data that is local to Activities and (ii) handle the unbounded set of pending asynchronous calls. Techniques such as the one in [Jhala and Majumdar, 2007] can be leveraged to this purpose.

From an implementation viewpoint, the blocking worklist algorithm reduces processing time but limits the deallocation or reuse of the memory that stores the analysis results for a given node (i.e., data sets reclamation). This is the memory that is required to store entries in *analysis*[ ] in Algorithm 4. This limitation is balanced by the fact that data sets for CFGs issued from models are much smaller than those for CFGs issued from programs and thus require less memory. The reason for this is the higher abstraction level of constructs in modeling languages that may require multiple basic blocks in programming language in order to capture equivalent behaviors.

5 CASE STUDY

In this section, we demonstrate the effectiveness of the CIBW algorithm on reaching definition analysis (i.e., the analysis of the variables' values). For the sake of clarity, we first consider the level of abstraction of single-Activities' CFGs and ignore the system supergraph's topology. We analyze two functional views of a 5G channel decoder (receiver side, uplink SC-FDMA, single antenna case, Physical Uplink Shared channel - xPUSCH). Subsequently, we analyze the composition of the two decoders in a more complex system and present performance results that consider the system supergraph's topology.

In the DIPLODOCUS [TTool/DIPLODOCUS, 2006] functional views, the semantics of communications between Activities in given by blocking read and write Actions. The latter operate on *logical* First-In First-Out (FIFO) buffers of finite size. A read operation is blocked until the required items are in the FIFO. A write operation on a full buffer suspends until items are consumed. The results of the reaching definition analysis allow to quantify the amount of data-samples that are produced and consumed by each signal-processing operation. These values are used by the compiler's middle-end to compute a Memory Exclusion Graph (MEG) [Desnos et al., 2014]. The latter is an intermediate representation that captures the exclusion relations among logical FIFO buffers. It is used by the compiler's back-end to allocate physical memory in the output code.

5.1 Analysis of individual diagrams

The algorithm of the 5G decoder is shown in Fig. 4. We considered two functional views that are representative of most existing implementations. Both views have a Controller Activity (not shown here) that governs the execution of processing operations. In the first view, that we call *sparsely controlled* (Fig. 7 and Fig. 5), each operation executes independently and only receives updates from the Controller concerning the number of samples to process according to environmental conditions (`Update_EvtIn` and `Update_EvtIn2` in Fig. 7 and Fig. 5). This view targets platforms where control is distributed among processing elements. In the second view, that we call *centrally controlled* (Fig. 8 and Fig. 6), each execution of an operation is tightly governed by the Controller that, for each schedule, dispatches the amount of samples to process. This view targets systems where control functions are centralized to a general-purpose processor.

We denoted each decoder's view with a SysML BD containing 11 SysML Composite Block Components: 1 for each operation in Fig. 4 as well as one Source and one Sink that respectively emit and collect samples. For each operation, we created separate Activities for the processing of control information from the Controller and the processing of input/output data samples. This strategy allows to target platforms where the two Activities can be mapped to different execution units. Thus, each Composite Block Component contains 2 SysML Primitive Block Components each containing a UML AD such as the diagrams in Fig. 5-8.

Table 1 lists statistics for both views. These numbers do not include dependency relations from the whole decoder's supergraph and only consider the analysis of individual diagrams. The numbers of visits in Table 1 are expressed as a function of n_v that indicates the number of different values for the control vari-

ables that are dispatched by the Controller to ADs. In Eq.3, n_v correspond to k_p.

In the case of the centrally controlled view, applying Eq. 1 to the entries in Table 1 results in no gain for the blocking worklist. For `F_` Activities (Fig. 8), both CIBW and CINBW result in no unnecessary visits because all variables are uninitialized and no information is propagated to the successors of the first ReceiveObjectAction. `X_` Activities (Fig. 8) are visited an equal number of times by both CIBW and CINBW as no inter-Activity dependency that modifies the value of control Variables is present.

Conversely, in the case of the sparsely controlled view, Fig. 7 and Fig. 5, the Controller dispatches two different values for Variables `size` and `stop` which results in $n_v = 2$. The number of visits of the CIBW algorithm for both `X_` and `F_` Activities is given by the sum of the visits for the nodes (excluding nodes for control statements) outside the loop and those inside the loop: $4 + 4n_v$ and $2 + 2n_v$ respectively. The number of unnecessary visits for the CINBW algorithm is equal to 3 as node `Update_EvtIn2(size, stop)` can propagate the value of `size` to three successors, for a `X_` Activity (Fig. 7). It is equal to 1 for a `F_` Activity as updates on the value of `size` can only be propagated to `Update_EvtOut(size, stop)` (Fig. 5). For both types of Activities, the number of unnecessary visits does not depend on n_v because of the absence of further ReceiveObjectActions in the diagrams' loops, other than `Update_EvtIn()`, `Update_EvtIn2()`.

Without considering the topology of the 5G decoder's supergraph, the CIBW algorithm yields a gain equal to $1 - \frac{6}{7} = 14.3\%$ for each individual `F_` Activity and $1 - \frac{12}{15} = 20\%$ for each individual `X_` Activity. As it is evident from Table 1, the small number of nodes that is typical of CFGs issued from models with respect to those issued from programs justifies the limited reclamation of data sets that is possible with the CIBW algorithm.

5.1.1 Generalization

Based on our experience, the topology of the CFGs in Fig. 5-8 is representative for models of telecommunication systems. For these topologies, Eq. 4 analytically expresses a generic gain, derived from Eq. 1, for the analysis of *individual* diagrams that do not consider the system supergraph's topology.

$$g = 1 - \frac{n^{pred} + n^{loop} \times n^{it}}{n^{pred} + n^{loop} \times n^{it} + n^{succ}} \tag{4}$$

Here, n^{pred} is the number of predecessors of the ReceiveObjectAction, n^{succ} the number of its successors, n^{loop} denotes the number of nodes in the loop and n^{it} the number of iterations. The behavior of the gain g can be studied by means of the limits in Eq. 5 and Eq. 6.

$$\lim_{n^{it} \to 0} 1 - \frac{n^{pred} + \cancelto{0}{n^{loop} \times n^{it}}}{n^{pred} + \cancelto{0}{n^{loop} \times n^{it}} + n^{succ}} = \frac{n^{succ}}{n^{pred} + n^{succ}} \tag{5}$$

$$\lim_{n^{it} \to +\infty} 1 - \frac{\cancelto{0}{n^{pred}} + n^{loop} \times n^{it}}{\cancelto{0}{n^{pred}} + n^{loop} \times n^{it} + \cancelto{0}{n^{succ}}} = 0 \tag{6}$$

From Eq. 6, it evinces that when the number of iterations is large, the performance of the CIBW degenerate to that of the CINBW. This is the case when the Controller dispatches to ReceiveObjectActions a large number of values for the control variables. Conversely, in Eq. 5, the gain is determined by the number of successor nodes n^{succ} that operate on the same data sets as those received by the ReceiveObjectAction. Because of the presence of a single ReceiveObjectAction in the loop body, in Eq. 4-6, we could express the gain by means of two sets of terms: $\{n^{pred}, n^{succ}\}$ that account for the number of visits at the first iteration of the CIBW algorithm, while $n^{loop} \times n^{it}$ denotes the visits at successive iterations. We can conclude that the CIBW effectively reduces the number of visits at the first iteration only. At successive iterations, the blocking mechanism of the CIBW algorithm does not bring any advantage over the CINBW.

However, if we consider the presence of multiple ReceiveObjectActions in the loop body, the blocking worklist reduces the number of visits at all iterations. The gain can be expressed as in Eq. 7, where $n^{succ}(r)$ is the number of successors of a ReceiveObjectAction r that operate on the same data set, D_r. The term $\sum_r n^{succ}(r)$ is the sum of the successors of a given ReceiveObjectAction r, over all ReceiveObjectActions. $n^{pred}_{r_1}$ is the number of predecessors of the first ReceiveObjectAction r_1 and $n^{succ}_{r_1}$ is the number of r_1's successors.

$$g = 1 - \frac{n^{pred}_{r_1} + n^{it} \times n^{loop}}{n^{pred}_{r_1} + n^{succ}_{r_1} + n^{it} \times (n^{loop} + \sum_r n^{succ}(r))} \tag{7}$$

In this case, for a large number of iterations, the gain does not degenerate to zero, Eq. 8, as opposed to Eq. 6.

$$\lim_{n^{it} \to +\infty} g() = 1 - \frac{n^{loop}}{n^{loop} + \sum_r n^{succ}(r)} = \frac{\sum_r n^{succ}}{n^{loop} + \sum_r n^{succ}} \tag{8}$$

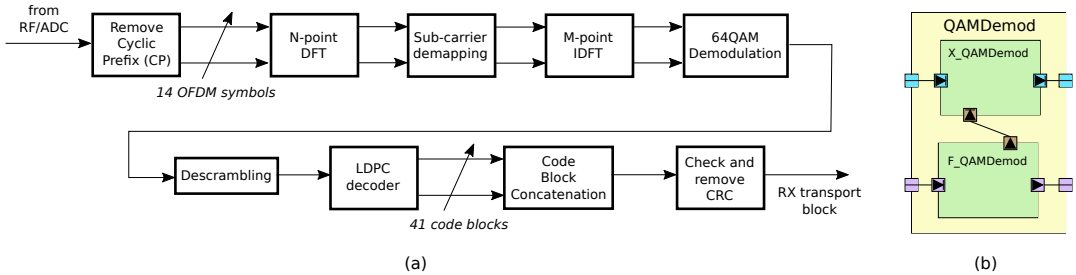


Figure 4: The block diagram of the 5G channel decoder (a). Each operation is modeled with the DIPLODOCUS SysML Blocks in (b), with data dependencies (blue Ports) and control dependencies (brown and purple Ports).

Table 1: Statistics for reaching definition analysis on the two views of the 5G decoder.

	Sparsely controlled			**Centrally controlled**		
Type of Activity Diagram	**Nb. of CFG nodes**	**Nb. of visits CIBW**	**Nb. of visits CINBW**	**Nb. of CFG nodes**	**Nb. of visits CIBW**	**Nb. of visits CINBW**
Data processing	9	$4+4n_v$	$4+4n_v+3$	5	$5n_v$	$13n_v$
Control processing	6	$2+2n_v$	$2+2n_v+1$	3	$3n_v$	$3n_v$

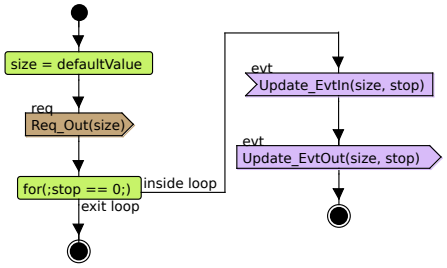


Figure 5: The UML AD for the control part of a generic operation for the sparsely controlled view.

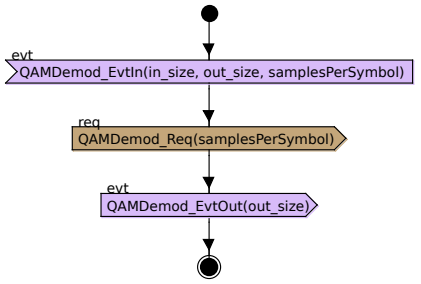


Figure 6: The UML AD for the control part of an operation (`F_QAMDemod`) for the centrally controlled view.

In Eq. 8, the value of the term $\sum_r n^{succ}(r)$ in the denominator depends on the relative position of ReceiveObjectActions. Its lowest bound is 1 and corresponds to a diagram where the loop's body has only 2 ReceiveObjectActions that are located, one after the other, at the very end of the loop's body.

5.2 Analysis of the control supergraph

Given the supergraph G^* of a system under analysis, the total gain is computed as the ratio of the number of visits for all Activities. This gain depends

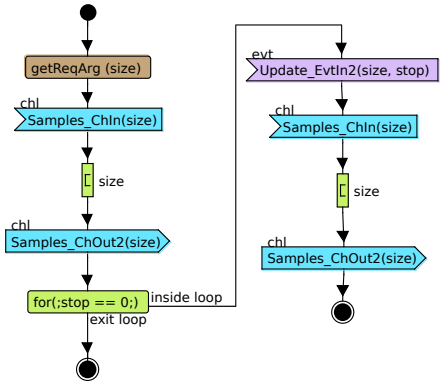


Figure 7: The UML AD for the data-processing part of a generic operation for the sparsely controlled view.

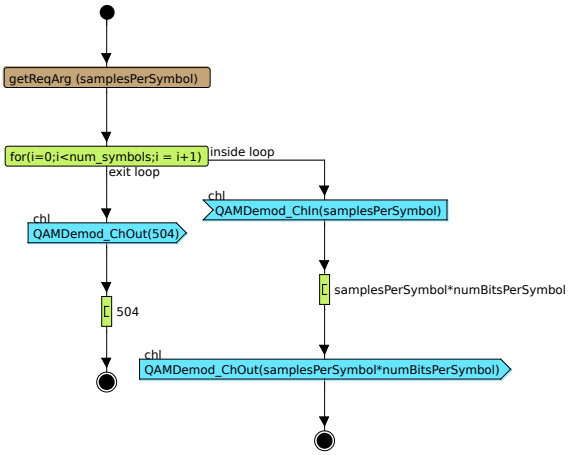


Figure 8: The UML AD for the data-processing part of an operation (`X_QAMDemod`) for the centrally controlled view.

on the supergraph's topology. When analyzing the supergraph of the 5G decoder in Fig. 4, the CIBW never blocks on incoming ReceiveObjectActions (e.g., `Update_EvtIn` in Fig. 6, `QAMDemod_EvtIn` in Fig. 5). Because of the linear dependencies among Activities, Fig. 4, when visiting Activity A_i, data-flow facts from Activity A_{i-1} are always available and the test at line 10 in Algorithm 4 always succeeds.
Fig. 9 shows the block diagram of a Software Defined Radio system that we designed to sense the frequency spectrum and opportunistically receive information on unused frequency bands. This Opportunistic Radio Sensing (ORS) system is composed of a Controller and the following algorithms:

- An energy detection algorithm called Welch Periodogram Detector (WPD) that senses the spectrum and detects when a given frequency band can be opportunistically used. It is modeled as a linear chaining of 6 SysML Composite Blocks that each contain 2 SysML Primitive Blocks interconnected as in Fig. 4b. Overall, 5 data dependencies and 10 control dependencies are present.
- Two instances of the 5G decoder in Fig. 4, modeled as described at the beginning of this section.
- An algorithm (High Order Cumulants, HOC) that searches for competing receivers with a higher priority. It is modeled as a linear sequence of 7 SysML Composite Components that each contain 2 SysML Primitive Components interconnected as in Fig. 4b. Overall, 9 data dependencies and 16 control dependencies are present.

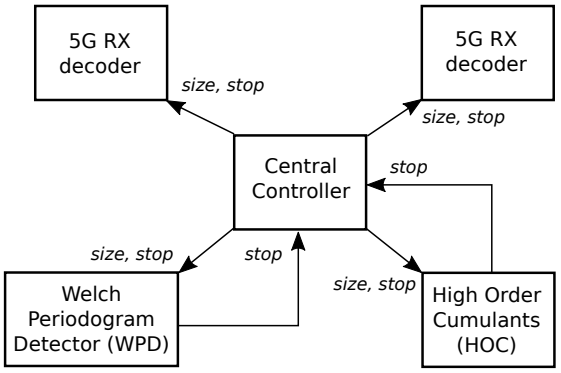


Figure 9: The block diagram of the ORS system. Edges are labeled with the Variables that they exchange.

In the control-flow supergraph of the diagram in Fig. 9, the Controller is the source node and each algorithm (5GRX, HOW, WPD) has its own sink node. Table 2 reports on the results of reaching definition analysis for the ORS system (the Controller is not included). Here, n_v refers to the number of values of Variable `size` that expresses the amount of samples to process.

Table 2: Statistics for reaching definition analysis on the specifications of the ORS system.

Signal processing algorithm	Nb. of CFG nodes	Nb. of visits CIBW	Nb. of visits CINBW
WPD	75	$75n_v$	$75n_v$
HOC	105	$105n_v$	$105n_v$
5G Decoders	270	$270n_v$	$270n_v \times 2$

In this system, the Controller first propagates Variable `stop` with a false value to the 5G decoders (to start execution). When reception cannot proceed opportunistically, the HOC and WPD algorithms communicate to the Controller to stop executing the 5G decoders. Analysis with the non-blocking worklist CINBW visits the 5G decoders twice: once to propagate `stop = false` and the second to propagate `stop = true`. On the other hand, the CIBW algorithm suspends analysis of the Controller on the incoming dependencies from HOC and WPD. Thus, it propagates to the 5G decoders both true and false values in a single visitation. The resulting total gain is given by $g = 1 - \frac{450n_v}{720n_v} = 37.5\%$.

6 CONCLUSIONS

In this paper we presented a framework to perform static data-flow model analysis on functional views denoted by UML Activity and SysML Block diagrams. These are transformed into control-flow graphs that also include the behavior of ProtocolStateMachines for the exchange of data through Ports. We proposed a visiting algorithm that combines both iterative and worklist searches as well as a blocking mechanism that reduces the number of unnecessary visits that result from the propagation of partial information among diagrams.
In future work, we will testbench a richer set of applications that includes platform-dependent communication protocols with a more complex semantics (e.g., the DMA transfers in Fig. 3).

REFERENCES

Aldrich, W. (2002). Using Model Coverage Analysis to Improve the Controls Development Process. In *AIAA Modeling and Simulation Technologies Conference*.

Atkinson, D. C. and Griswold, W. G. (2001). Implementation techniques for efficient data-flow analysis of large programs. In *ICSM*, pages 52–61.

Briand, L. C., Labiche, Y., and Lin, Q. (2005). Improving statechart testing criteria using data flow information. In *ISSRE*, pages 104–114.

Checko, A., Christiansen, H. L., Yan, Y., Scolari, L., Kardaras, G., Berger, M. S., and Dittmann, L. (2015). Cloud RAN for Mobile Networks - A Technology Overview. *IEEE Communications Surveys Tutorials*, 17(1):405–426.

Desnos, K., Pelcat, M., Nezan, J., and Aridhi, S. (2014). Memory Analysis and Optimized Allocation of Dataflow Applications on Shared-Memory MPSoCs. *Journal of VLSI Sig. Proc. Syst. for Signal, Image, and Video Tech.*, pages 1–19.

Eclipse CDT (Visited on October 2018). `http://www.eclipse.org/cdt/`.

fUML (Visited on October 2018). `http://www.omg.org/spec/FUML/1.2.1/`.

Gerstlauer, A., Haubelt, C., Pimentel, A. D., Stefanov, T. P., Gajski, D. D., and Teich, J. (2009). Electronic System-Level Synthesis Methodologies. *IEEE TCAD*, 28(10):1517–1530.

Jhala, R. and Majumdar, R. (2007). Interprocedural Analysis of Asynchronous Programs. In *POPL*, pages 339–350.

Kahn, G. (1974). The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475.

Kienberger, J., Minnerup, P., Kuntz, S., and Bauer, B. (2014). Analysis and Validation of AUTOSAR Models. In *MODELSWARD*, pages 274–281.

Kim, Y. G., Hong, H. S., Bae, D. H., and Cha, S. D. (1999). Test cases generation from UML state diagrams. *IEE Proceedings - Software*, 146(4):187–192.

Lai, Q. and Carpenter, A. (2013). Static Analysis and Testing of Executable DSL Specification. In *MODELSWARD*, pages 157–162.

Lee, E. A. and Parks, T. M. (1995). Dataflow process network. *Proceedings of the IEEE*, 83(5):1235–1245.

Malm, J., Ciccozzi, F., Gustafsson, J., Lisper, B., and Skoog, J. (2018). Static Flow Analysis of the Action Language for Foundational UML. In *ETFA*.

Mellor, S. J. and Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Nielson, F., Nielson, H. R., and Hankin, C. (2010). *Principles of Program Analysis*. Springer.

OMG (2014). The Object Constraint Language Specification Version 2.4. `https://www.omg.org/spec/OCL/`.

OMG (Visited on October 2018). Action Language for Foundational UML (ALF). `http://www.omg.org/spec/ALF/`.

Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61.

Saad, C. and Bauer, B. (2013). Data-Flow Based Model Analysis and Its Applications. In *MODELS*, pages 707–723.

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

Schwarzl, C. and Peischl, B. (2010). Static- and Dynamic Consistency Analysis of UML State Chart Models. In *MODELS*, pages 151–165.

Seidewitz, E. (2014). UML with Meaning: Executable Modeling in Foundational UML and the Alf Action Language. In *HILT*, pages 61–68.

Selic, B. (2003). The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25.

Torczon, L. and Cooper, K. (2007). *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition.

TTool/DIPLODOCUS (2006). `http://ttool.telecom-paristech.fr/diplodocus.html`.

VERIMAG (2018). IF: Intermediate Format and Verification Tool set. `http://www-verimag.imag.fr/article58.html?lang=en`.

Waheed, T., Iqbal, M. Z., and Malik, Z. I. (2008). Data Flow Analysis of UML Action Semantics for Executable Models. In *ECMDA-FA*, pages 79–93.

Yu, L. (2014). *A Scenario-based Technique to Analyze UML Design Class Models*. PhD thesis, Colorado State University, Department of Computer Science.

Yu, L., France, R. B., and Ray, I. (2008). Scenario-Based Static Analysis of UML Class Models. In *MODELS*, pages 234–248.