Slicing UML-based Models of Real-time Embedded Systems

Conference Paper · July 2018

DOI: 10.1145/3239372.3239407

3 authors, including:

Reza Ahmadi
Queen's University
4 PUBLICATIONS **6** CITATIONS
SEE PROFILE

Ernesto Posse
Zeligsoft
23 PUBLICATIONS **107** CITATIONS
SEE PROFILE

Slicing UML-based Models of Real-time Embedded Systems

Reza Ahmadi
Queen's University
Kingston, Ontario, Canada
ahmadi@cs.queensu.ca

Ernesto Posse
Zeligsoft
Gatineau, Quebec, Canada
eposse@zeligsoft.com

Juergen Dingel
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT

Models of Real-time Embedded (RTE) systems may encompass many components with often many different kinds of dependencies describing, e.g., structural relationships or the flow of data, control, or messages. Understanding and properly accounting for them during development, testing and debugging can be challenging. This paper presents an approach for slicing models of RTE systems to facilitate model understanding and other model-level activities. A key novelty of our approach is the support for models with composite components (with multiple hierarchical levels) and capturing the dependencies that involve structural and behavioural model elements, including a combination of the two. Moreover, we describe an implementation of our approach in the context of Papyrus-RT, an open source Model Driven Engineering (MDE) tool based on the modeling language UML-RT. We conclude the paper with the results of applying our slicer to a set of UML-RT models to validate our approach and to demonstrate the applications of our approach for facilitating model-level analysis tasks, such as testing and debugging.

CCS CONCEPTS

• **Computing methodologies** → **Model development and analysis**; • **Software and its engineering** → **Model-driven software engineering**; **Unified Modeling Language (UML)**; **Software testing and debugging**; *Embedded software*; *State systems*; *Specification languages*;

KEYWORDS

Model Driven Engineering, Model Slicing, Slicing, Dependency Analysis, UML-RT, UML

ACM Reference Format:
Reza Ahmadi, Ernesto Posse, and Juergen Dingel. 2018. Slicing UML-based Models of Real-time Embedded Systems. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18), October 14–19, 2018, Copenhagen, Denmark.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3239372.3239407

MODELS '18, October 14–19, 2018, Copenhagen, Denmark

ACM ISBN 978-1-4503-4949-9/18/10…$15.00
https://doi.org/10.1145/3239372.3239407

1 INTRODUCTION

Modern Real-Time Embedded software (RTE) play a fundamental role in controlling many products and devices found in, e.g., telecommunication systems, cars, and airplanes. In such complex systems, the behavior of the system depends on real-time constraints as well as on communications with the environment possibly using various protocols. Using code-centric-only approaches for developing complex RTE systems is very challenging. MDE techniques tackle this challenge by raising the level of abstraction on which the developers construct software.

Even though MDE principles including abstraction, automation, and analysis [39] can help deal with the complexity of software, models of modern industrial systems still often are large and can become overwhelming without additional support. For instance, complex RTE systems may encompass many interacting components, and the components of such systems may be composite (i.e., consist of other components). Debugging, testing, and maintaining models with composite components can be challenging, because developers may need to inspect every part and understand the way it impacts and interacts with other parts, which also can be composite.

Slicing has already proved to be an effective technique for understanding, testing, and debugging large programs [20, 25, 46, 47]. However, more work is needed to leverage it for MDE in the best possible way. While multiple approaches on, e.g., slicing state machines [15, 26], software architectures and design diagrams [24, 27, 44] have been proposed, the slicing of collections of models that capture behaviroural as well as structural aspects of the system has not been explored much.

In this paper, we present an approach and prototype tool for slicing models of RTE systems. We have built our slicer on UML-RT, a domain specific language, whose constructs are built from standard UML constructs [40]. UML-RT is a popular industrial modeling language that is used for modeling industrial systems and is supported by multiple open-source and commercial tools (Eclipse Papyrus-RT [13], IBM RSA-RTE, IBM RoseRT). Our slicer removes unrelated structural and behavioral elements from composite and non-composite components in the model. As a result, model analysis tasks such as testing and debugging are simplified. Our prototype implementation uses the open source MDE tool (Papyrus-RT). The prototype can visualize dependencies between model elements and compute slices of different, user-selected parts of the model. Since the original layout of model elements in the slice is preserved, the modeler can recognize them easily and user-friendliness is increased. To evaluate our approach, we use an industrial case study provided by one of our industrial partners as well as a set of other academic case studies with different sizes and complexities.

In the next section we briefly introduce UML-RT with an example model. Then, we explain our approach. Next, the implementation

of our slicing algorithm is presented. We then briefly describe our prototype tool and an evaluation of the approach and the tool.

2 BACKGROUND

In this section we give some context with an overview of UML-RT, as well as the current state of the art in program slicing and state-based models slicing.

2.1 Modeling in UML-RT

Constructing complex, often distributed real-time systems needs powerful, well-defined modeling constructs, as well as strong tool support. These constructs and tools can be used to design a well-defined architecture for such complex systems, which eases not only the development of the initial system, but also facilitates maintenance and evolution [40]. UML for Real-Time (UML-RT) [40] is a domain-specific language that has its roots in the well-known Real-time Object Oriented Modeling (ROOM) language [41]. Similar to languages such as SysML [19], and MARTE [17], UML-RT is a UML profile whose constructs are built from standard UML constructs. Since UML-RT focuses on modeling real-time systems, it is, compared to UML, a small language with light notation.

The main concepts of UML-RT are *capsules*, *capsule parts*, *ports*, *protocols*, and *connectors*. A capsule is an independent active class with its own control flow. Capsules may contain internal structure, defined by *capsule parts*. A capsule part is an instance of another capsule inside a capsule. Capsules that contain capsule parts are called *composite* capsules, whereas other capsules are called *non-composite* capsules. Capsules own ports, allowing them to communicate via message passing. To allow two capsules to communicate, capsule ports are typed with *protocols*, that is, a formal description of the incoming and outgoing *messages* a capsule can receive from and send to other capsules. Two ports typed with the same protocol can be formally connected through *connectors*. Ports can be of different kinds: external ports are boundary ports exposed by the capsule to be connected with other capsules. Internal ports allow a capsule to communicate with the capsule parts it contains (Fig. 1 shows an example UML-RT model that contains some of the mentioned constructs).

On the behavioural side, state machines model the behaviour of capsules. Example state machines are shown in Section 3. A UML-RT state machine is an extension of a Mealy state machine [31] augmented with extra features, including state actions, composite states, and deep-history. UML-RT state machines have minor differences with those of UML, for instance, they do not support orthogonal composite states (multiple concurrent state machines in a composite state) [35].

A capsule part in a composite capsule is typically a *fixed* part, which means the lifetime of a part depends on the lifetime of its container (e.g., a fixed part is always instantiated automatically and after its container). However, UML-RT supports dynamic models, as well. To this end, UML-RT provides facilities for creating two specific types of capsule parts: *optional* and *plug-in* [4, 40]. An optional part does not have a strong lifetime relationship with its container capsule, which means an optional part can be created after the container is created or can be destroyed before the container is destroyed. Plug-in capsules, on the other hand, are placeholders for capsule parts that are populated dynamically by capsule instances. Plug-in capsules are necessary when the exact capsule instance to be put in a capsule slot is not known statically and is determined at run-time [4, 40].

To facilitate modeling, the UML-RT Runtime System (RTS) library includes a set of services that provide utilities for, e.g., importing parts, logging messages, and setting timers. For instance, a composite capsule can dynamically create optional parts or populate a plug-in capsule slot by interacting with a *frame* port, *log* ports are used to print messages in the console mostly as a means for debugging and tracing, and *timer* ports are used to trigger transitions in periodic intervals or at a specific point in time.

2.2 A Fabric Dyeing Machine in UML-RT

To illustrate our approach to model-slicing, we will use as a running example a *Fabric Dyeing Machine (FDM)* [41]. Fig. 1 illustrates the top-level structure of this system (examples of state machines of this system are shown in Section 3). The system consists of eight capsules: *DyeingUnit (DU), DyeingSoftware* and *DyeingSystem* are composite capsules, and the rest are non-composite capsules. The *Dyeing Run Controller (DRC)* capsule part communicates with *Dyeing Specification(DS)* part to establish required parameters to set up a dyeing task, such as *running time, temperature, and dyeing level*. When the system is running, the DU part provides variable values such as, *solution tank level* and *temperature* to the DRC and accepts device commands such as *open dye valve*, and *open drain valve* from the DRC (through the three pairs of ports and connectors between DU and DyeingSoftware parts, depicted in Fig. 1). DU and DRC capsule parts are optional (optional parts are normally shaded, but not DU here to show its containing parts). The reason behind this design is to allow the container capsule to create new DU and DRC instances dynamically based on the number of dyeing tasks. The communication between the capsules and their parts happen through *opOut2* and *unitCreatePort* internal ports. In UML-RT the maximum number of instances of an optional capsule must be defined statically at design time (*_MAX* in our model is used for that purpose).

2.3 Program and Model Slicing

Program slicing is typically the computation of the set of program statements that may affect the values of a variable at a specific point in a program. The computed set of program statements is called a *slice*. The variable and the specific point of interest in the program are called the *slicing criterion*. Program slicing techniques typically construct a Program Dependency Graph (PDG) [22, 33], which is a directed graph, where each node in the graph represents a statement and each edge represents a dependency between two statements. The slice is computed by traversing the PDG by starting from the nodes associated with the slicing criterion, and marking the reachable nodes. Finally, the marked nodes are preserved and the rest are removed.

Slices computed by traversing the graph backward or forward are known to facilitate debugging and maintenance, respectively [1, 20, 38, 46]. For instance, *backward slicing* preserves statements and predicates of a program that affect a variable x at point p, and is used for debugging purposes, whereas *forward slicing* finds statements

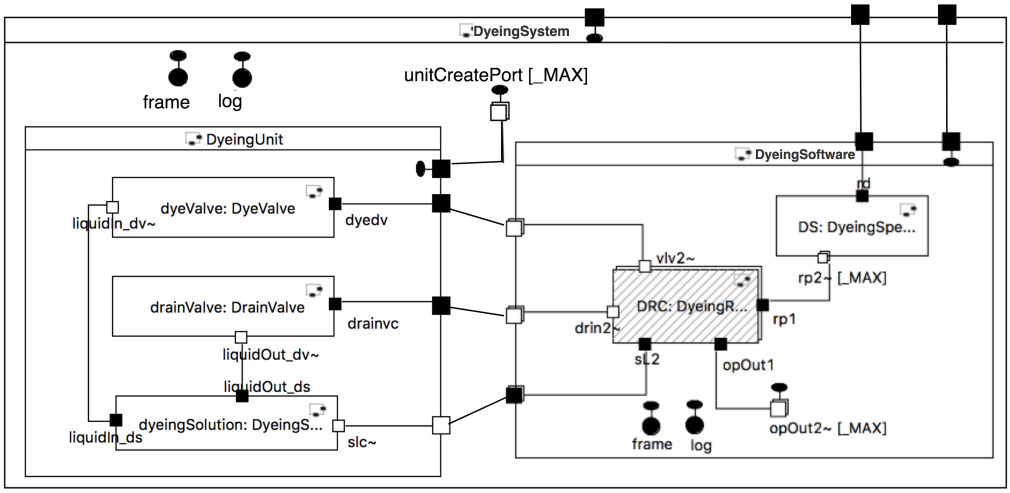

Figure 1: A *composite capsule* of a Fabric Dyeing system modeled in UML-RT

and predicates of the program that are affected by a variable x at point p, and hence can be used for identifying parts that will be affected by the modification of the program [1, 38]. Note that the direction of the graph traversal in forward and backward slicing is different, but the algorithm for both is defined by employing the same concepts [46].

In addition, program slice computation can be *static* or *dynamic*. In static program slicing, all statements that can affect the values of a variable are in the slice, which is totally independent of the program input values. Whereas in dynamic program slicing, all statements that affect the values of a variable are computed with respect to a set of given program input values [25].

As opposed to program slicing, model slicing has not been studied widely, in particular for models of reactive and real-time systems, even though MDE is becoming more prevalent. Models of RTE systems can be complex and large, since a system may consist of many interacting components. During development, testing and maintenance of such systems, developers often need to identify parts of the model that are affected by a set of inputs, or parts that send specific outputs. For instance, a developer might be interested in parts of a model that cause a component to send an *error* message or parts of a model that receive a set of messages and show them to the user. In the UML-RT model of Fig. 1, for instance, the developer would like to find out which parts of the model can potentially cause the *drainValve* capsule part to send an *error* message on the port *drainvc*. In this case we can slice the *DyeingSystem* capsule and take {*drainvc*, *error*} as the slicing criterion. Once the criterion is specified, a slicing tool preserves the structural and behavioral parts related to the criterion and removes the rest.

3 APPROACH

Before we present our algorithm for computing UML-RT model slices, we quickly summarize our approach, then we introduce various dependencies that may exist in UML-RT capsules, and then we describe few concepts related to our approach.

3.1 Approach Overview

We propose a directed graph representation for dependencies between a UML-RT model elements, where each graph node represents an element in the capsule and each edge represents a dependency between two elements. Since a UML-RT model consists of a set of capsules, we construct a Component Dependency Graph (CDG) for each capsule of a model. To construct this graph, we first construct a dependency graph for each capsule that owns a state machine (a container capsule may not have a state machine), and we call it Behavioral Dependency Graph (BDG). This graph represents the dependencies between the state machine and ports of the capsule as well as the dependencies between elements in the state machine (i.e, transitions, states and action code).

If the capsule is composite, which means it contains other capsule parts, then we construct another graph called Structural Dependency Graph (SDG) for the dependencies between structural model elements that shows dependencies between ports and connectors as well as ports and parts. For example, in a composite capsule C that contains parts $p2$ and $p1$, where $p1$ contains $p3$, the SDG of C shows whether or not the part $p2$ is reachable from the part $p3$ by traversing any number of ports and connectors.

In addition, since in composite capsules, parts may communicate with other parts by sending or receiving messages, or the container capsule may dynamically instantiate its containing parts, we create Communication and Dynamic Model Dependencies Graph (CADG) that represents dependencies between capsule-parts and parts-parts.

The CDG of a non-composite capsule is a BDG of that capsule, since all the dependencies of such capsules are defined in terms of the state machine and ports of the capsule. For composite capsules, however, the CDG is constructed by integrating the BDG (if applicable), the SDG, and the CADG of the container capsule as well as the BDGs of each part in the container capsule. Observe that parts in a composite capsule can be composite, as well, so we follow the same rule that we mentioned to construct the dependency graph of composite parts. In fact, CDG is a transitive closure of the dependencies

between capsule elements. That means, given two behavioral or structural elements of a capsule, CDG shows whether or not the first element is reachable from the second one by traversing any number of edges (typed with any kind of dependency) in the graph.

A slicing criterion specifies elements of a model that a user is interested in. Given the slicing criterion, our slicing algorithm returns (a copy of) the model with only elements that are related to, or have an effect on the elements in the slicing criterion. To this end, our algorithm traverses the edges in the CDG for the given slicing criterion. Through CDG traversal, our tool identifies the relevant model elements from a UML-RT model based on the dependencies among them to compute the slice.

Our approach is based on UML-RT, which is a UML profile and its constructs are built from standard UML constructs [40]. Similar to UML-RT models, SysML [19] and MARTE [17] models encompass both structural (are called blocks or structures) and behavioral (state machines and sequence diagrams) aspects. Moreover, these languages allow composite structures in models and use ports and connections to connect structures. Therefore, our approach can be adopted to slice other UML-based languages, such as SysML and MARTE models.

In order to express dependencies between capsule elements, we use the notion of $x \xrightarrow{d} y$, which means y is dependent on x, and the type of the dependency is d. In the following sections we elaborate different possible types of dependencies in a UML-RT capsule. Note that, in this paper we assume that in a state machine, only transitions are active elements (which means cause changes in the model, by e.g., sending a message or updating a variable). However, our approach is applicable to active states, as well.

3.2 Dependencies of a Non-composite Capsule

We identify four types of dependencies in a non-composite capsule: Control, Data, Action Code, and Transition-Port. Our definition of Control, Data, and Action Code Dependencies are similar to the definitions provided in prior work, while Transition-Port is a new type of dependency that we identify to capture the interactions of ports and state machines. We first only briefly explain Control and (different realization of) Data Dependencies in our approach followed by Action Code, and Transition-Port dependencies.

Control Dependency. Control dependency in a state machine captures the notion that one transition may affect the traversal of another transition. We adopted a notation of Control Dependency that supports possible non-termination in state machines, proposed by [9]. In a state machine, a transition t_j is control dependent on a transition t_i, iff t_i has at least one sibling t_k (outgoing transitions from the same state are siblings), where t_j is not in all the paths that start from the source state of t_i and t_k. For instance, in Fig. 2, *t5* is control dependent on *t3*.

Data Dependency. UML-RT capsules have strong encapsulation, so attributes are not accessible from other capsules and sharing data between capsules is realized via message exchanges. In addition, variables defined on transitions are locally accessible, so data dependency between transitions is defined in terms of capsule global attributes. The types of attributes can range over primitive types, user defined complex data types, or RTS data types (e.g., timers).

We adopt the Data Dependence definition proposed in [9, 26] to capture data dependencies. In a state machine, a transition t_j is data dependent on a transition t_i, iff t_i updates a variable that t_j uses. For instance, in the state machine of the *DRC* capsule shown in Fig. 2, transition *t2* is data dependent on the transition *t1*, since *runTime* is initialized on *t1* (*runTime=rt;*) and is used in transition *t2* (as an argument to set the *timer1* object). This dependency is shown on the dependency graph in the same figure.

As mentioned before, UML-RT encompasses a set of libraries and data structures dedicated to facilitating modeling real-time systems. *Timing services* library [4] is one of them. This library encompasses a set of data structures for treating timed models. For instance, the library includes the *Timing* protocol. A port that is typed with a Timing protocol acts as a timer and can trigger transitions in periodic intervals or at a specific point in time (absolute or relative to the current time). We use the notion of data dependency to capture the dependencies of timer ports to the action code on transitions. To this end, we treat timer ports as any other typical object defined in a capsule. For instance, in Fig. 2, *t3* is data dependent on the transition *t2*, since the *timer1* object is initialized on *t2* and is used as the trigger of transition *t3*. This dependency is shown on the dependency graph in the same figure. Note that timer objects in UML-RT are set using *informIn*, *informEvery*, and *cancel* to initialize a timer to timeout at a specific time, timeout in periodic intervals, or to cancel a timer to prevent a timer from sending a timeout signal, respectively.

Action Code Dependencies. Action code usually consists of simple code that is executed when a transition is fired. In UML-RT action code can be written in a number of languages, including C++ and Java. Here we assume that action code does not contain pointers and procedure definitions. Note that such assumption does not prevent us from slicing a wide range of models, since in models action code is usually a short, simple code fragment. In slicing sequential programs [46] a statement s_i is control dependent on statement s_j if s_j affects the execution of s_i, e.g., if s_i is an assignment in an *if* block, and s_j is the *if* condition, then s_i is control dependent on s_j. Data dependence, on the other hand, is defined in terms of the definition-reference relations of variables in a program [46], i.e., a statement s_i is *data dependent* on statement s_j if s_j updates a variable that s_i uses. The same notion is used to capture control and data dependencies between action code statements on a transition. For instance, in transition *t3* in Fig. 2, *timer2.informIn(dt)* is data dependent on the line *dt=5*. In addition, the whole action code of transition *t3* is control dependent on the transition's guard (i.e., *runTime > 0*).

Transition-Port Dependencies. These dependencies capture the notion of potential *interactions* between transitions and ports in a capsule. A transition depends on a port p if either it is triggered by a message from the port p, or it has a send action to p. In a capsule C, $p \xrightarrow{tpd} t$ means the transition t is dependent on the port p, and there exists such a dependency iff:

- $t \in T(SM_C)$ and SM_C is the state machine of C;

- t is triggered by a message from p, or t has a send action to p, where $p \in Ports_C$;

where $T(SM_C)$ represents the set of transitions of SM_C, and $Ports_C$ are the ports of C. Examples of such dependencies are shown in Fig. 2 (red-dotted edges). Note that a port on a capsule can be a user defined port (e.g., *opOut2*) or a system port (e.g., *log* and *frame* on capsule DS and *timer1* on capsule DRC). As an example of this type of dependency, transition *t3* in the DRC state machine is triggered by the port *timer1*, so *t3* is dependent on *timer1*. As another example, *t0* in the DS state machine sends messages to the port *log*, so *t0* is dependent on the port *log*.

3.3 Dependencies of a Composite Capsule

A composite capsule contains capsule parts and may have a state machine. As we explained before, the CDG of a composite capsule encompasses both structural and behavioral dependencies. The behavioral dependencies consist of dependencies in the state machine of the capsule (if applicable) as well as dependencies in each part of the capsule. In addition, in a composite capsule, model elements may be structurally dependent (e.g., a port needs a connector for message passing) and a capsule may dynamically instantiate capsule parts. Therefore, for composite capsules, we introduce two new dependency types: *Structural Dependencies* and *Dynamic Model Dependencies*. In addition, since in a composite capsule, capsule parts may exchange messages through their ports, we identify *Communication Dependencies* [15] between parts, as well.

Structural Dependencies. Structural dependencies represent information on structural connections between capsule parts, such as the dependency of ports on a connector (*connEnd* in Fig. 2) and ports on parts (*portOf*). Moreover, each capsule part is always dependent on its container capsule, regardless of whether the two capsules communicate or not, and thus we create a dependency between each capsule part and its container capsule (*partOf*). Therefore, during slicing if a capsule part is preserved so is its container capsule, i.e., a part and its container always coexist. Fig. 2 shows various structural dependencies in a composite capsule (black edges), where the SDG is constructed from these dependencies.

Dynamic Model Dependencies. As mentioned in previous chapters, UML-RT allows dynamically instantiating capsule parts using optional capsule parts. To this end, action code in the capsule's state machine must *incarnate* the mentioned parts (or *destroy* to remove the instances). In a composite capsule C, $t \xrightarrow{inc} part$ means *part* is *incarnation dependent* on the transition t in the state machine of C, and there is such a dependency iff:

- $part \in Parts_C$;
- $t \in T(SM_C)$ and SM_C is the state machine of C;
- the action code of t incarnates/destroys the $part$;

where $T(SM_C)$ represents the set of transitions of SM_C and $Parts_C$ are the capsule parts of C. For example, in Fig. 2, there is an incarnation dependency between *DRC* and *t0* (shown as a green-dotted edge). This dependency is due to the fact that the capsule part *DRC* is incarnated in the action code of the transition *t0*. The word *frame* in parentheses means a message is sent through the port *frame* to perform this task.

Moreover, UML-RT allows importing capsule instances at runtime and put them in available capsule slots using plug-in capsules. To this end, action code in the capsule's state machine must *import* the mentioned instances (or *deport* to remove the plug-in capsule instances). In a composite capsule C, $t \xrightarrow{imp} part$ means *part* is *import dependent* on the transition t of C's state machine, and there is such a dependency iff:

- $part \in Parts_C$;
- $t \in T(SM_C)$ and SM_C is the state machine of C;
- the action code of t imports/deports the $part$;

where $T(SM_C)$ represents the set of transitions of SM_C and $Parts_C$ are the capsule parts of C.

Communication Dependencies. In a composite capsule, we find communication dependencies between capsule parts. In a composite capsule C, $t_j \xrightarrow{com} t_i$ means t_i is communication dependent on t_j, and there exists such a dependency between two transitions iff:

- $t_i \in T(SM_i)$ and $t_j \in T(SM_j)$ where $i \neq j$;
- the trigger of t_i matches a message sent by the action code of t_j.

where SM_i and SM_j are the state machines of capsule parts i and j and both parts are owned by C. $T(SM_i)$ represents the set of transitions of the state machine SM_i. Note that it is also possible that C communicates with its parts, so SM_i and SM_j will represent the state machines of the capsule C and a part of C. For example, in Fig. 2, there is a communication dependency between transitions *t0* (in the state machine of DS) and *t2* (in the state machine of DRC), since *t0* sends the message *start* on the port *opOut2*, which is received by *t2* through the port *opOut1*.

3.4 Slice Validity and Slicing Criteria

In order to identify a slice from the model it is necessary to define a set of concepts. Below we define these concepts.

Slice of a capsule. A capsule C' is a slice of C if the sets of elements of C' are subsets of the corresponding sets of C (e.g. the parts of C' are a subset of the parts of C, etc.).

Valid slice of a capsule. If a capsule C' is a slice of C, C' is said to be *valid with respect to capsule C*, iff the set of all possible behaviors (or interactions) of C' is a subset of the set of behaviors (or interactions) of C. For example, a valid slice of capsule *DyeingUnit* preserves elements and action code of the original capsule that cause *DyeingUnit* to report its current number of active units.

Slicing criterion. A slicing criterion defines which aspects of behavior and structure of the capsule the slicing must preserve. Given a capsule C, a slicing criterion is a set of ports P in C and a set of messages M in C. For instance, a developer might be interested in determining which elements of the capsule DyeingSoftware have a role in sending the message *drainStatus* on the port *drainvc*, and hence the slicing criterion is: {*drainvc, drainStatus*}. Observe that the slicing criterion can be specified based on the vocabulary of a property of interest (e.g., the message m sent on the port p is followed by the message n on the same port), so the slice preserves

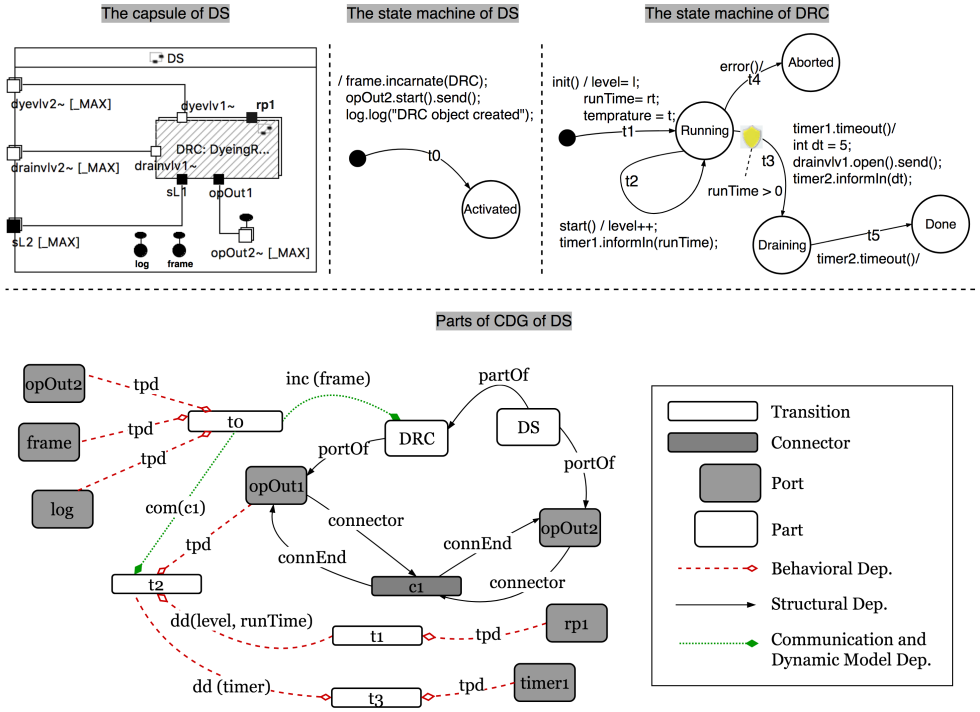

Figure 2: Top: parts of the DS capsule and state machines of DS and DRC. Down: Parts of the CDG of DS

the behavior and structure of those parts of the capsule that affect the behavior of the property [7].

Valid slice with respect to a criterion. If a capsule C' is a slice of C, C' is said to be a valid slice with respect to some criterion if:

- C' is valid with respect to C;
- the slice includes only structures and behaviors involving messages and ports from the slicing criterion. Observe that this does not mean that the behaviours in question include only such messages and ports, as other messages may be part of a behaviour that leads to a particular message from the slicing criterion being sent or received. For example, if the state machine of C has transitions $t_1 : S_1 \rightarrow S_2$[1], $t_2 : S_2 \rightarrow S_3$ and $t_3 : S_3 \rightarrow S_4$ and the slicing criterion includes a message sent only by t_1 and t_3, a slice C' would have to include t_2 as well, and so it would have a behaviour that includes other actions that may not be related to the criterion;

For example, similar to the original capsule *DS* (in Fig. 2), a valid slice of *DS* with respect to the criterion {*drainvlv1*, *open*} will contain the structure and behaviour that include actions to send the message *open* to the port *drainvlv1*. The slice will include transition *t3* and the port *drainvlv1*, since the action code of transition *t3* sends the message *open* to the port *drainvlv1*. Moreover, the slice will preserve transitions *t1* and *t2*, as well, since the action code of *t2* sets the *timer1*, which triggers *t3*, and without the transition *t1*, *t3* is not reachable. In addition, the part *DRC* is preserved in the slice, whose container will be the capsule *DS*.

[1] $t_1 : S_1 \rightarrow S_2$ means there is a transition t_1 with source state S_1 and target state S_2

3.5 The Slicing Algorithm

Given a set of incoming/outgoing messages M and a set of ports P of the UML-RT capsule C, the goal is to compute the slice C' with respect to P and M. Informally, C' is a capsule with a subset of C's ports, parts, connectors, etc. that contains all model elements (including action code) that may contribute to either the sending or reception of a message from the given set M through a port in P. Such model elements are referred to as *contributing elements* (whereas the rest of the elements are referred to as *non-contributing ones*). By removing the non-contributing elements from the capsule, the slice is constructed.

Algorithm 1 shows the algorithm to compute the slice. The algorithm starts by creating a copy of the original capsule C, creating an empty graph, and adding all the elements (e.g., transitions, ports, parts) of the capsule into the empty graph (lines 2-7). In order to produce a valid slice from the original capsule with respect to the slicing criteria M and P, we need to identify nodes in the graph that represent a transition, that sends or receives any message in M through any port in P. Nodes (that represent such transitions) are identified are marked as contributing (lines 8-14). In this part of the algorithm, *tr.InOuts* represents the set of messages that either trigger the transition *tr* or messages that the action code of *tr* sends. So far we have identified transitions (*inOutNodes*) that send or receive our desired messages through the desired ports. A capsule may include many capsule parts, but we are only interested in capsule parts that *inOutNodes* depend on (a transition depends on the part that owns it, or a part that some action code in that part enables the transition. Note that triggering transitions might be transitive. For instance, the behavior of part p_1 may trigger part p_2 that triggers

part p_n). Therefore, we first identify the communication and dynamic model dependencies between the capsule parts and update the graph with these dependencies (line 15), so now the graph contains information about reachable communicating parts. Then, we use the information in the graph to remove from the capsule any capsule part that is not reachable from the parts that own a transition in *inOutNodes* (lines 16-20). In the next step, behavioral and structural dependencies between all transitions (of the remaining capsule parts) are identified and edges are created between their corresponding nodes in the graph (lines 21-22). Then, the algorithm traverses in the graph and marks any reachable nodes from *inOutNodes* as contributing (lines 24-27). If, during the traversal performed by *TraverseGraph*, a node is reachable from the current node via any dependency, then the node is marked as contributing and the function is called again. In case the nodes are reachable via a data dependency, then the algorithm needs to collect variables involved in this dependency, so any other transitions in the graph that affect these variables are marked as well (line 38). The algorithm finally removes non-contributing nodes from the capsule (lines 28-32).

Observe that while the graph is updated by removing non-contributing nodes, C' (the slice) is automatically updated, and hence the remaining nodes in the graph represent the computed slice. The slicing algorithm only removes elements, and preserves non-contributing nodes that connect initial states (of a state machine) to a contributing node (line 29) and thus we maintain the structural integrity of the model required to preserve its executable behaviour. Moreover, since the slice is computed with respect to a slicing criterion and the resulting slice encompasses a subset of the elements and behaviors of the original capsule, the resulting slice is valid with respect to the capsule and the criterion.

As mentioned earlier, in UML-RT the behavior and the structure of models can change dynamically. We previously explained that we capture various dependencies with respect to plug-in and optional capsule parts to support the structural dynamicity of models. With respect to dynamic behavior, for instance, *DRC* (in Fig. 2) receives data, including running time, temperature and level from the Dyeing Specification capsule (via port *rp1*), where the mentioned capsule itself reads the data from a human operator. Whether or not the state machine of *DRC* enters the *Draining* state depends on the *runTime* input value, therefore it is possible that transition *t3* is never taken (due to the guard on the transition). If during slicing, there is a dependency to *t3*, even if there might be cases that *t3* never executes, our algorithm preserves *t3*. For instance, if the slicing criterion is {*drainvlv1*, *open*}, then, the slicer preserves *t3* (since transition *t3* sends message *open* to the port *drainvlv1*), even though it is possible that *t3* is not fired at run-time if the *runTime* input value does not satisfy the guard (e.g.,*runTime=0*).

Algorithm 1 Compute the slice of a composite or non-composite capsule C

```
   Input: The capsule C, the set of messages M, the set of ports P
   Output: C', which is the slice of C with respect to M and P
 1: procedure ComputeSlice(C, M, P)
 2:     C' ← C, graph ← create an empty graph
 3:     for each Element element in C do
 4:         node ← new Node(element)
 5:         mark node as non-contributing
 6:         graph.add(node)
 7:     end for
 8:     inOutNodes ← create an empty list
 9:     for each Transition tr in graph do
10:         if ∃m ∈ M, p ∈ P | (m sent/received through p) ∧ (m ∈ tr.InOuts) then
11:             inOutNodes ← inOutNodes ∪ {tr}
12:             Mark tr as contributing
13:         end if
14:     end for
15:     CreateCADG (graph) // communication and dynamic model dependencies
16:     for each Part part in graph do
17:         if part is not reachable from parts that own {inOutNodes} then
18:             C'.remove(part) // Remove the part from the capsule
19:         end if
20:     end for
21:     CreateBDG (graph) // behavioral dependencies
22:     CreateSDG (graph) // structural dependencies
23:     V ← create an empty list
24:     for each Transition tr in inOutNodes do
25:         V ← {v | v:variable used in the guard/action code of tr}
26:         TraverseGraph(tr, V)
27:     end for
28:     for each non-contributing Node node in graph do
29:         if (!ConnectsInitial2Contributing(node)) then
30:             graph.remove(node) // Remove the node from the graph
31:         end if
32:     end for
33: end procedure
34: procedure TraverseGraph(node0, V)
35:     for each non-contributing Node node in node0.dependencies do
36:         if node0 is data dependent on node with respect to V then
37:             mark node as contributing
38:             V ← V ∪ {variables used in guards or action code of node}
39:             TraverseGraph (node, V)
40:         else if node0 has other dependencies on node then
41:             mark node as contributing
42:             TraverseGraph (node, V)
43:         end if
44:     end for
45: end procedure
```

4 TOOL IMPLEMENTATION

We have implemented a prototype tool[2] to compute UML-RT model slices using the algorithm that we presented. Our tool was implemented in Java and uses the Eclipse Modeling Framework (EMF) related libraries to process a UML-RT model. We have integrated our tool in Papyrus-RT [13], an open source modeling IDE for UML-RT with an active user community. Our tool can be integrated to other IDEs that support UML-RT, including RSA-RTE [3] and Rhapsody. The schematic design of our prototype tool is shown in Fig. 3. The components of this design are explained in the following.

Model Processor implements a set of libraries on top of EMF to connect to a model and collect information on model elements, such as ports, parts, connectors, and transitions. This information is stored in some vectors to be used by the dependency graph creator.

Action Code Parser traverses the state machines of each capsule and collects the action code on each transition and sends them to the dependency graph creator. To this end, the module uses a set of regular expressions for pattern matching in the action code. For instance, to figure out whether a transition in the state machine of a capsule *incarnates* a part or not, this module searches the action code for *incarnation* patterns and finds the port and the part involved in such operation.

[2]Source code of our slicing tool and our UML-RT models can be found at: https://bitbucket.org/rezaahmadi/wmslicer

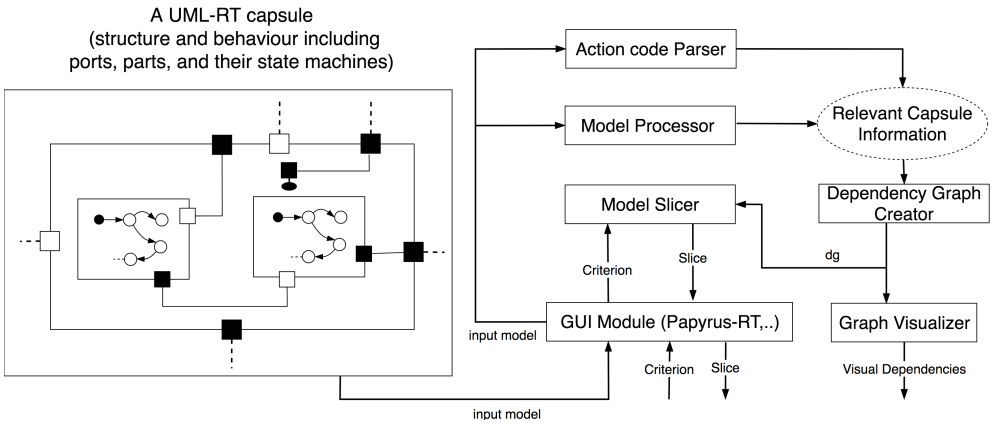

Figure 3: The overview of our prototype tool

Dependency Graph Creator constructs a graph, in which nodes represent a model element (e.g., a part or port), and edges represent a dependency. To this end, this module implements various procedures to compute structural, behavioral and communication and dynamic model dependencies and merges all the dependencies into a single graph. The mentioned procedures are referred in the Algorithm 1 as *CreateSDG*, *CreateBDG*, *CreateCADG*, respectively. For instance, to capture dynamic model dependencies between a capsule and a part, if the part is optional or plug-in, then the module searches the action code on the state machine of the capsule, and if it finds the required code for incarnating or importing the part on a transition, then an edge (typed with *dynamic model dependency*) is constructed in the graph that connects the transition and the part.

Model Slicer takes the slicing criteria from the GUI (i.e., the Papyrus-RT IDE, which is based on the Eclipse platform) and the dependency graph in order to compute the slice. The model slicer uses the algorithm that we introduced in the previous section to traverse the dependency graph and conduct the slicing. Our tool allows user interactions such as specifying a slicing criterion, visualizing dependencies between model elements, running slicing commands, as well as visualizing the computed slice.

Graph Visualizer collects information on dependencies from the dependency graph creator and presents them graphically. The model developer can interact with the component to display and inspect the dependencies of a capsule conveniently as well as to export dependencies to pdf files. Our graph visualizer uses mxGraph [5], which is an open source library for constructing interactive graphs and charts. Fig. 4 shows some dependencies of the capsule *DyeingUnit* using our visualizer. Currently some information (e.g., the type of the dependency, the message to the ports, and variable names) are shown on the edges, but more information about the dependencies can be shown to the user.

5 EXPERIMENTAL EVALUATION

We have conducted several experiments using our tool on a number of industrial and academic UML-RT models with different sizes and complexities. We will show how model-level testing and debugging can be optimized by integrating these tools with our slicer.

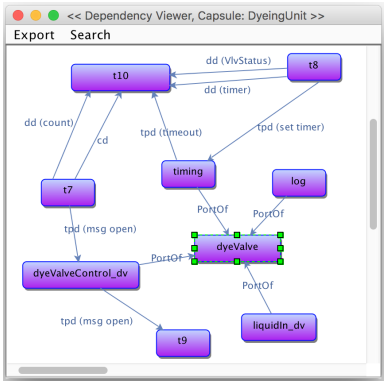

Figure 4: Parts of the dependencies of the *DyeingUnit*

For the testing application, we integrated our slicer with a Symbolic Execution (SE) tool presented in [48]. SE is an analysis technique, which is used for automatically generating test cases with high coverage [14]. For each component of a model, SE creates a tree that represents all possible execution paths of that component. SE is known to be an effective technique in finding bugs in programs and models by deeply exploring the state space of a model or program, but usually suffers from path explosion [14]. We show how the efficiency and scalability of symbolic execution of some models can be improved by integrating our slicer with the SE tool.

For the debugging application, we integrated our slicer with a model debugger presented in [10]. The debugger works at model level by instrumenting the state machine of a component to make it debuggable. The code generated from the instrumented component is executed and debugged through a set of commands provided by the tool. Since the tool works at model level, it is not dependent on any specific code-level debugger such as GDB [45]. However, the instrumentation and code generation processes can be heavy tasks for large models. We will show the integration of our slicer with such tool can make model-level debugging a more efficient task.

We performed our experiments using a computer equipped with a 3.0 GHz CPU and 8GB of RAM, with identical hardware, software,

configurations, and system workload throughout the experiment. We first briefly introduce the UML-RT models that we used in this experiment, and then we elaborate our results.

5.1 Sample Models

Autonomous Rover (AR) [8] is a system that models a vehicle equipped with four wheels driven by two engines and sensors to collect information from the environment as well as sensors to detect obstacles. *Fabric Dyeing Machine (FDM)* is a system that models a fabric dyeing machine (more information in Section 2.2). This system was originally designed in ROOM [41] and we manually converted it to a behaviorally equivalent system in UML-RT. *Adaptive Cruise Control (ACC)* is a system that adjusts the vehicle speed and distance to that of a target vehicle. ACC decelerates or accelerates the vehicle's speed according to the speed and distance settings established by the driver. This system was originally designed in AutoFOCUS [2, 18] and we manually converted it to a behaviorally equivalent system in UML-RT. *Parcel Router (PR)* [30] is a system that models an automatic system where tagged parcels are routed through successive chutes and switchers to a corresponding bin. The model checks whenever parcel jam occurs and prevents a parcel from being transfered from one chute to another one until it is empty. *Traffic Light Controller (TLC)* is a timed system that controls traffic lights using a set of timers and based on the number of cars that it detects by a camera. *Car Lock (CL)* is a system that models a control system for locking and unlocking vehicle doors. *PBX (Private Branch eXchange)* is a system that models the interaction of a user with a private telephone network used within an organization. The user can communicate with the PBX system from within the organization or from the outside world through various communication channels, including Voice over IP or ISDN [29].

In our case studies, CL is a relatively simpler model compared to the other systems, e.g., FDM and AR that are relatively larger. FDM and AR each have multiple nested capsule parts, and use UML-RT advanced features, such as optional parts, and port/part replication. PBX is our largest model, and approaches industrial size. The model includes 29 capsules, 1,716 lines of C++ action code, and includes state machines with states that are nested in up to six levels. More details about the models are specified in Table 1.

5.2 Results

Table 1 shows the complexity of each model, which is presented in terms of the total number of capsules in the model, followed by the average number of parts, ports, transitions and lines of action code in each capsule, computed before and after the slicing. We conducted slice computations several times for each capsule in each model using randomly selected slicing criteria, and hence the complexity of capsules were reduced in various ways. Since the results are based on multiple computations, the numbers shown in the table are the cumulative average values computed for each model. In the best case, slicing reduced the complexity of some capsules by 40 percent, whereas in the worst case slicing could only remove some action code and transitions. However, as shown in the table, on average, slicing could make the capsules considerably smaller (e.g., reducing the number of action code in larger systems by around 50% in FDM, TLC, and PBX).

The last six columns of the table summarize the average size of the code generated from each capsule and its slice, the average time required for each capsule to generate test cases, and the average time required to prepare a capsule for debugging by instrumentation, respectively. The table compares the required times for the original capsules as opposed to their slices. As shown in the table, in almost all cases slicing made testing and debugging tasks considerably more efficient. For instance, for the FDM system, slicing makes test generation around 3 times faster (41 seconds for the original capsules as opposed to 12 + 3.75 seconds for the sliced capsules). For the same system, slicing made instrumentation more than 45 percent faster (25.5 seconds as opposed to 9.5 + 3.75 seconds). However, based on the table, the impact of slicing on efficiency of testing and debugging is negligible for small systems (e.g., CL system).

Fig. 5 presents the measured memory and CPU time required by our tool for slicing on the Papyrus-RT tool. The box plots present results of multiple slice computations for each capsule in each model. For instance, for the PBX system, the minimum slicing time for each capsule is 9 seconds, the maximum is 19 seconds, the median is 16.8 seconds, and the mean is 15.4 seconds. As shown in the figure, the run-time and memory requirements for the PBX and the PR systems are the largest on average. Table 1 also shows that both these systems have the most complex capsules in terms of the number of parts in each capsule. In addition, PBX has the most number of action code in each capsule (60 lines of action code on each transition). Fig. 5 shows some outliers (the polygon spots) on the plots, which represent numbers that are distant from the rest of the numbers in our computations. In FDM system, for example, an outlier on the box plot on the right shows slicing time of around 10 seconds and another outlier on the box plot on the left shows the memory consumption of around 150MB for the same system. These two numbers represent the slice computation and memory consumption for a composite capsule in FDM system that encompasses 7 capsule parts. Therefore, constructing dependency graphs and computing slices seem to be heavier for capsules with nested parts as well as for capsules with more action code.

6 RELATED WORK AND DISCUSSION

Various program slicing techniques have been used for testing, debugging and maintenance [20, 22, 32, 46, 47] as well as tools that allow analyzing architectural dependencies in a program by identifying interactions between components [44]. With respect to slicing models and state-based systems, most of the work reported in literature addresses development of techniques based on slicing the behavioral aspects of a model or only architectural diagrams. Work on slicing the whole model that captures structural as well as behavioral aspects has scarcely been reported in the literature. For instance, the approach and tool proposed by Korel et al. [26] slices state machines by taking a transition and variable as the slicing criterion. There are multiple other techniques capable of slicing based on different criteria and for various applications, including, testing, debugging, and model checking [7, 15, 21, 23], where all consider slicing state machines only. In another work, slicing is conducted using model transformation [28] on class diagrams and state machines, and hence the output slice may structurally differ

Table 1: The effect of slicing on optimizing model-level testing and debugging

Model	Model Complexity (Original/Slice)					Avg. Slicing Time (sec.)	Optimization (Avg. per capsule)					
							Generated Code (KB)		Test Generation Time (sec.)		Instrumentation Time (sec.)	
	Total# Capsules	Avg.# Parts	Avg.# Ports	Avg.# Transitions	Avg.# Action Code		Original	Slice	Original	Slice	Original	Slice
AR	5	0.8/0.6	2.5/2.5	4.5/3	16/10	5.12	28	20	25	9	23	12
FDM	9	0.9/0.6	4/3	4.5/3	11/6	3.75	44	25	41	12	25.5	9.5
ACC	7	1/0.6	4/3	7/5	15/10	3.35	28	17	66.5	16	13	7
PR	7	1.7/1	2/2	4/3	15/9	10	32	26	65	23	18.8	5
TLC	2	0.5/0.5	4.5/3	7/4	14/7	6.92	41	32	52	24	8	3
CL	4	1.5/1.3	1.5/1.5	4.5/4	7/6	1.59	11	10	10	8	9.4	8
PBX	29	2/1.5	7/4	10/6	60/37	15.4	135	69	130	59	58	21

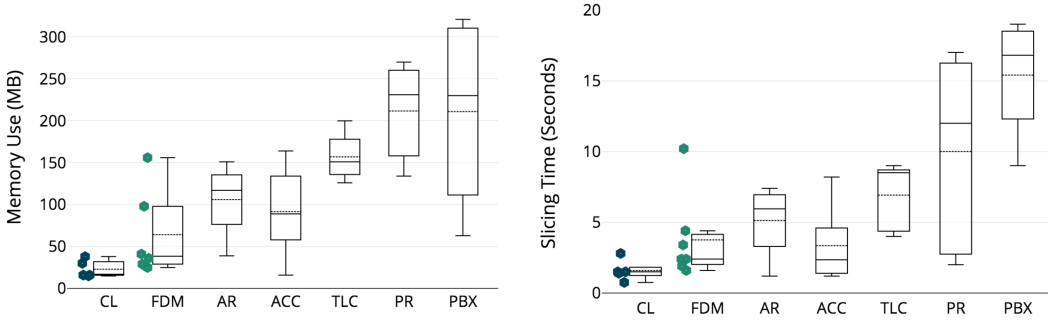

Figure 5: CPU time and memory consumption of the slicer tool

from the input model, but the semantics are preserved. In our approach, however, the structure of the slice is preserved and hence model comprehension is easier for model developers.

Other researchers proposed techniques that consider architectural aspects of models only [24, 44]. For instance, Kagdi et al. [24] propose a technique and tool to analyze and slice UML hierarchical class diagrams that captures relationships between classes such as associations or aggregations. Similarly, the technique proposed by Lallchandani et al. [27] slices UML architectural models that finds dependencies between class diagrams and sequence diagrams. In addition, the technique proposed by Falessi et al. [16] computes slices for analyzing safety requirements in SysML models, where they consider slicing blocks and activity diagrams only. Slicing has been also used for verifying UML class diagrams annotated with OCL [42], where slicing constructs a set of sub-model from the original model and verification is applied on the sub-models.

There are multiple work on modeling model slicers in order to be able to construct slicers for any domain specific language [11, 12]. In these approaches, a model, its corresponding metamodel, and a criterion are inputs to a tool. The tool, then, generates an slicing function, which can produce a slice that is compatible with the input metamodel. These approaches have other interesting applications, as well, e.g., by analyzing an operation such as *state machine flattening* on a UML model, the tool finds out that state machine and sequence diagrams are the effective metamodel in this particular operation rather than the whole UML metamodel [11]. However, in these approaches, the authors did not explain how they treat state machines and the action code that affects the interactions between the components of a model.

There are multiple other slicing tools and techniques with various applications. For instance, Pietsch et al. [34] proposed a tool to incrementally extract sub-models from a larger model, where the sub-models are guaranteed to be editable. Salay et al. [37] proposed a slicing tool for slicing heterogeneous models for model change management. In this approach, the authors consider slicing a collection of interrelated models, but an explanation of how to capture the dependencies of two concurrent state machines in a heterogeneous model is missing. In our approach, however, we capture such interactions between two concurrent state machines and include them in our dependency graph. Slicing has been also used for feature models to compute various projections of feature models for more straightforward analysis of such models [6]. In addition, there are techniques for slicing MATLAB Simulink models [36, 43], where slices are computed by capturing control and data dependencies between blocks. In these techniques, data dependency is calculated using flow of data between the blocks, and control dependency is calculated by capturing predicate blocks in the model that control the execution of other blocks.

7 CONCLUSION AND FUTURE WORK

In this paper, we have described and implemented an approach for slicing UML-RT models using various criteria. We showed the

applications of our approach and tool for optimizing model-level testing and debugging by conducting an empirical evaluation on a set of UML-RT models. Based on our results, slicing could make the tasks mentioned considerably more efficient. Nevertheless, there are some limitations. In our current evaluations, the slicing criteria is randomly selected. We think that the experts ability to effectively specify the slicing criteria has an impact on the usefulness of the generated slices. Therefore, as a future work, we intent to extend our work by adding the mentioned evaluations to our study. In addition, we intend to support more options for slice presentation to help with model comprehension and analysis. For instance, instead of preserving the slice and removing the irrelevant model elements, we preserve all model elements, but highlight the relevant ones within the complete model.

REFERENCES

[1] A piece of cake. http://www0.cs.ucl.ac.uk/staff/mharman/exe1.html. Accessed: 2018-02-02.

[2] Autofocus. https://af3.fortiss.org/en/, 2016. Accessed: 2017-12-15.

[3] Rational Software Architect Realtime Edition. http://www-01.ibm.com/support/docview.wss?uid=swg24034299, 2016. Accessed: 2016-8-10.

[4] Modeling real-time applications in RSARTE. https://www.ibm.com/developerworks/community/blogs/a8b06f94-c701-42e5-a15f-e86cf8a8f62e/resource/Documents/RSARTEConcepts.pdf, 2017. Accessed: 2018-03-06.

[5] mxGraph. https://jgraph.github.io/mxgraph/, 2017. Accessed: 2018-03-02.

[6] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B France. Slicing feature models. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 424–427. IEEE Computer Society, 2011.

[7] Reza Ahmadi, Nicolas Hili, and Juergen Dingel. Property-aware unit testing of uml-rt models in the context of mde. In *European Conference on Modelling Foundations and Applications*, pages 147–163. Springer, 2018.

[8] Reza Ahmadi, Nicolas Hili, Leo Jweda, Nondini Das, Suchita Ganesan, and Juergen Dingel. Run-time monitoring of a rover: MDE research with open source software and low-cost hardware. In *Open Source for Model-Driven Engineering (OSS4MDE'16)*.

[9] Kelly Androutsopoulos, David Clark, Mark Harman, Robert M Hierons, Zheng Li, and Laurence Tratt. Amorphous slicing of extended finite state machines. *IEEE Transactions on Software Engineering*, 39(7):892–909, 2013.

[10] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Model-level, platform-independent debugging in the context of the model-driven development of real-time systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 419–430. ACM, 2017.

[11] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Modeling model slicers. In *International Conference on Model Driven Engineering Languages and Systems*, pages 62–76. Springer, 2011.

[12] Arnaud Blouin, Benoît Combemale, Benoit Baudry, and Olivier Beaudoux. Kompren: modeling and generating model slicers. *Software & Systems Modeling*, 14(1):321–337, 2015.

[13] Francis Bordeleau and Edgard Fiallos. Model-based engineering: A new era based on Papyrus and open source tooling. In *OSS4MDE@ MoDELS*, pages 2–8, 2014.

[14] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[15] Valentin Chimisliu and Franz Wotawa. Improving test case generation from UML Statecharts by using control, data and communication dependencies. In *13th International Conference on Quality Software (QSIC)*, pages 125–134. IEEE, 2013.

[16] Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, Lionel Briand, and Antonio Messina. SafeSlice: A Model Slicing and Design Safety Inspection Tool for SysML. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of Software Engineering*, pages 460–463. ACM, 2011.

[17] Madeleine Faugere, Thimothee Bourbeau, Robert De Simone, and Sebastien Gerard. Marte: Also an uml profile for modeling aadl applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359–364. IEEE, 2007.

[18] Martin Feilkas, Andreas Fleischmann, C Pfaller, M Spichkova, D Trachtenherz, et al. A top-down methodology for the development of automotive software. 2009.

[19] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language.* Morgan Kaufmann, 2014.

[20] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[21] Mats PE Heimdahl and Michael W Whalen. Reduction and slicing of hierarchical state machines. In *Software Engineering–ESEC/FSE'97*, pages 450–467. Springer, 1997.

[22] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[23] Wang Ji, Dong Wei, and Qi Zhi-Chang. Slicing Hierarchical Automata for Model Checking UML Statecharts. *Formal Methods and Software Engineering*, 2495:435–446, 2002.

[24] Huzefa Kagdi, Jonathan I Maletic, and Andrew Sutton. Context-free slicing of UML class models. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.*, pages 635–638. IEEE, 2005.

[25] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information processing letters*, 29(3):155–163, 1988.

[26] Bogdan Korel, Inderdeep Singh, Luay Tahat, and Boris Vaysburg. Slicing of state-based models. In *Proceedings of International Conference on Software Maintenance, 2003.*, pages 34–43. IEEE, 2003.

[27] Jaiprakash T Lallchandani and Rajib Mall. Slicing uml architectural models. *ACM SIGSOFT Software Engineering Notes*, 33(3):4, 2008.

[28] Kevin Lano and Shekoufeh Kolahdouz-Rahimi. Slicing of uml models using model transformations. In *International Conference on Model Driven Engineering Languages and Systems*, pages 228–242. Springer, 2010.

[29] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.

[30] Jeff Magee and Jeff Kramer. *State models and Java programs.* wiley, 1999.

[31] George H Mealy. A method for synthesizing sequential circuits. *Bell Labs Technical Journal*, 34(5):1045–1079, 1955.

[32] Lynette I Millett and Tim Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, pages 75–83, 1998.

[33] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.

[34] Christopher Pietsch, Manuel Ohrndorf, Udo Kelter, and Timo Kehrer. Incrementally slicing editable submodels. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 913–918. IEEE Press, 2017.

[35] Ernesto Posse and Juergen Dingel. An executable formal semantics for UML-RT. *Software & Systems Modeling*, 15(1):179–217, 2016.

[36] Robert Reicherdt and Sabine Glesner. Slicing matlab simulink models. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 551–561. IEEE, 2012.

[37] Rick Salay, Sahar Kokaly, Marsha Chechik, and Tom Maibaum. Heterogeneous megamodel slicing for model evolution. In *ME@ MODELS*, pages 50–59, 2016.

[38] N Sasirekha, A Edwin Robert, and Dr M Hemalatha. Program slicing techniques and its applications. *arXiv preprint arXiv:1108.1352*, 2011.

[39] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[40] Bran Selic. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*, pages 250–260. Springer, 1998.

[41] Bran Selic, Garth Gullekson, and Paul T Ward. *Real-time Object-Oriented Modeling*, volume 2. John Wiley & Sons New York, 1994.

[42] Asadullah Shaikh, Robert Clarisó, Uffe Kock Wiil, and Nasrullah Memon. Verification-driven slicing of uml/ocl models. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 185–194. ACM, 2010.

[43] Adepu Sridhar and D Srinivasulu. Slicing matlab simulink/stateflow models. In *Intelligent Computing, Networking, and Informatics*, pages 737–743. Springer, 2014.

[44] Judith A Stafford, Alexander L Wolf, and Mauro Caporuscio. The application of dependence analysis to software architecture descriptions. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 52–62. Springer, 2003.

[45] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with GDB. *Free Software Foundation*, 675, 1988.

[46] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[47] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.

[48] Karolina Zurowska and Juergen Dingel. Symbolic execution of UML-RT state machines. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1292–1299. ACM, 2012.