Components, Platforms and Possibilities: Towards Generic Automation for MDA

Ethan K. Jackson
Microsoft Research, USA
ejackson@microsoft.com

Eunsuk Kang
Massachusetts Institute of Technology, USA
eskang@csail.mit.edu

Markus Dahlweid
Microsoft Research, Germany
mdahlwei@microsoft.com

Dirk Seifert
Microsoft Research, Germany
dseifert@microsoft.com

Thomas Santen
Microsoft Research, Germany
tsanten@microsoft.com

ABSTRACT

Model-driven architecture (MDA) is a model-based approach for engineering complex software systems. MDA is particularly attractive for designing embedded systems because models can be easily evolved as hardware and software requirements evolve. However, efforts to apply MDA in industrial settings expose several open problems surrounding tooling: Engineers need automated techniques that are scalable, general, and extensible. In this paper we describe the FORMULA framework as a novel approach towards general automation for MDA. We develop a running example and benchmarks to compare our tools with other state-of-the-art approaches.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Software Architectures

General Terms

Algorithms, Design

1. INTRODUCTION

Model-driven architecture (MDA) is a model-based approach for engineering complex software systems [22]. It has the advantage of decoupling the software architecture from the computing platform, while allowing both to be co-designed. MDA is particularly attractive for designing embedded systems because models can be easily evolved as hardware and software requirements evolve. Consequently, there exist long-running industrial efforts to apply MDA to embedded systems, and there exist significant tooling efforts for MDA [6].

However, efforts to apply MDA in industrial settings expose several open problems surrounding tooling:

- *Scalability* - In MDA, design complexity appears in the form of constraints over the interfaces of software components and constraints on the legal deployments of components to computing elements [22]. Without tool support to aid in model construction, the architect must manually solve global and difficult constraint problems to arrive at a legal design. Thus, scalability depends on automated techniques, yet there exist few results from the tool community showing scalable techniques for MDA.

- *Generalized Synthesis* - Often, tools are crafted to solve a specific part of the modeling task. For example, there are tools that synthesize schedulable deployments given timing constraints on software components [3]. However, this synthesis is often uni-directional. If the modeler requires assistance writing timing constraints, then the same tools cannot offer assistance. General automation should be able to synthesize any part of the model.

- *Extensibility* - Industrial case studies have shown that the modeling process must be extensible: Software components will be extended with new abstractions, e.g. security abstractions [4], and the legal deployments will be impacted by such extensions. It is an important open problem to provide an effective tool architecture in the face open-ended constraints and abstractions [15].

In this paper we present our FORMULA framework [18] as a tool towards generic automation for MDA. FORMULA addresses the problems surrounding MDA tooling through a combination of innovations: FORMULA (1) provides a unique formal specification language for encoding general MDA abstractions, (2) exposes formal composition operators for extending specifications and reasoning over extensions, (3) utilizes *symbolic execution* of *logic programs* and reduction to a *satisfiability modulo theories* (SMT) solver to provide state-of-the-art formal methods for model synthesis.

We illustrate our approach with a running example derived from the *AUTOSAR* project [13], which is a large industrial effort to apply MDA to automotive embedded systems. Our example shows how tool support becomes problematic and how FORMULA handles these issues. Next, we propose several benchmarks to compare FORMULA to existing tools. Finally, we provide scalability comparisons be-

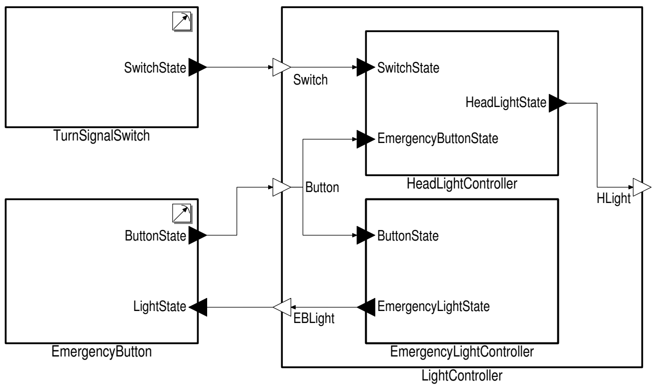

Figure 1: Example of componentized software architecture.

tween the *Alloy Analyzer*, which uses relational algebra as its foundations, and *SModels*, which computes *stable models* for *answer-set programming*.

This paper is divided into the following sections. Section 2 introduces MDA in the context of AUTOSAR and Section 3 formalizes a running example and benchmark based on AUTOSAR. Section 4 shows how FORMULA can be used to encode this running example. Section 5 provides tool comparisons and Section 6 discusses some related work. Finally, we conclude in Section 7.

2. AUTOSAR

The *AUTomotive Open System ARchitecture* (AUTOSAR) *Initiative* is an approach to engineering automotive systems that decouples the software architecture from the computing platform [13]. Its core members include *BMW*, *Toyota*, *General Motors*, *Ford*, *Volkswagen*, *Daimler*, and *Bosch*. AUTOSAR is described in a rich set of standards; these are now in their fourth revision. The AUTOSAR initiative is in its seventh year.

The goals of AUTOSAR align with MDA. As a result, the standards are phrased in the language of MDA: There is an AUTOSAR *metamodel* describing a schema for models of automotive systems. The metamodel is supplemented with guidelines for semantics of models. These supplements are intentionally partial to allow various tool vendors and software manufacturers to instantiate AUTOSAR in different ways. Opened-ended extensibility is a fundamental quality of the approach. Finally, the standards also describe a *runtime environment* (RTE) to support software components.

The tool ecosystem surrounding AUTOSAR is complex, but can be roughly broken into three parts: (1) Modeling tools for constructing models, (2) Code generators for producing implementation and platform configuration from system models, and (3) Implementations of the RTE. In this paper we focus on the modeling tools, as downstream tools assume or verify that good models have been constructed.

2.1 The Metamodel

At the core of AUTOSAR is a *metamodel* providing a common schema for models. The metamodel can be further subdivided into schemas for modeling software architectures, platforms, and assignments of components onto computing elements (called *electronic control unit*s or ECUs). Tool vendors and manufacturers must utilize this schema. However, the AUTOSAR metamodel is not closed; it can be extended to enrich models with new abstractions [15].

2.1.1 Components

Figure 1 shows a simplified model of an AUTOSAR software architecture (example adapted from [19]). It consists of a hierarchy of software components interconnected through ports. The software component is the basic unit of functionality, and AUTOSAR provides a rich vocabulary for components. An *atomic component* cannot be subdivided and encapsulates implementation in the form of *runnable entities*. For example, the HeadLightController and EmergencyLightController are atomic components. The former controls the four main lights on a vehicle and the latter controls the light on the emergency button, which is pressed to cause all head lights to flash. Sensors and actuators are modeled with *sensor\actuator components*: The TurnSignalSwitch and EmergencyButton components are examples, as indicated by the small gauge icon. Finally, components can be aggregated using *composition components*. The LightController aggregates the HeadLightController and EmergencyLightController components. Composition components are treated as purely structural and do not have implementation footprints.

2.1.2 Communication

Components communicate through *ports* connected by *connectors*. Again, there is a spectrum of port types to express different communication styles: *Sender-receiver*, *client-server*, and *event-triggered* ports are available. For the sake of discussion, the ports in Figure 1 can be viewed as sender-receiver ports. (Our port notation differs from AUTOSAR notation. We use unfilled ports to indicate interfaces on composition components.)

2.1.3 Platform and Mapping Constraints

A platform is comprised of *ECU instances* connected by *physical channels* as well as specifications of ECU message formats. The physical channels model the bus topology and the metamodel provides specialized constructs for common bus types including *CAN*, *FlexRay*, *LIN*, and *Ethernet*. A full discussion of these concepts is outside the scope of this paper.

A complete AUTOSAR model consists of the software architecture, platform model, and the *software component mapping* assigning atomic components onto ECUs. The mapping is not arbitrary and is subject to *mapping constraints*. Some constraints are implicitly present, e.g. if two software components communicate then they must be mapped to ECUs that communicate. The metamodel also predefines two types of constraints that can be manually added: *component clustering* and *component separation* constraints. A component clustering constraint requires components to be placed on the same ECU, while a component separation constraint requires two components to be placed on different ECUs. Extensions of AUTOSAR may further constrain the legal mappings.

2.2 Modeling is Hard

This brief introduction is enough to illustrate that constructing legal models is an NP-hard problem. Determining a legal mapping in the face of component separation constraints is a *graph coloring problem*, where the number of colors is equal to the number of available ECUs. Model construction becomes problematic when the ratio of components to ECUs is large and there are many constraints. In practice these separation constraints may be solved manually. However, as more realistic constraints come into play, e.g. scheduling constraints, then manual model construction also becomes infeasible in practice.

3. EXAMPLE AND BENCHMARK

In this section we develop a running example based on AUTOSAR. We use this example to show how abstractions are specified and extended with FORMULA. We also use this example as a benchmark to evaluate other tools that could offer generalized automation for MDA. With this in mind, our example is intentionally simple while retaining those qualities from AUTOSAR that should challenge existing formal methods. To our knowledge there are few published benchmarks for evaluating the effectiveness of generalized automation in MDA.

3.1 Basic Abstraction

DEFINITION 1. **Software -** *Let $S = \langle C, id_C, E_C \rangle$ be a structure where C is a finite set, $id_C : C \rightarrow \mathbb{N}$ is an injection, and $E_C \subseteq C \times C$ is a relation.*

A software model consists of a finite set of components C. Each $c \in C$ has a unique ID given by $id_C(c)$. The reason for explicit IDs shall be explained shortly. The relation E_C gives the communication topology between components; the details of ports are ignored. An element $(c_1, c_2) \in E_C$ indicates a directed communication path where c_1 sends data to c_2.

DEFINITION 2. **Platform -** *Let $P = \langle N, F, id_F, E_N \rangle$ be a structure where N, F are finite sets, $id_F : F \rightarrow \mathbb{N}$ is an injection, and $E_N \subseteq N \times N$ is a relation.*

A platform model consists of a finite set of computing nodes N (e.g. ECUs) and a finite set of functionalities F (e.g. runnable entities). The injection id_F assigns a unique ID to functionalities. Similarly, E_N gives the directed communication topology between nodes. Nodes can always communicate with themselves and all other bus details are ignored.

DEFINITION 3. **Mapping -** *Let $M = \langle S, P, map \rangle$ be a structure where S is a software model, P is a platform model, and $map : C \rightarrow N$ is a function from components to nodes.*

The mapping model completes the description by placing components onto nodes according to *map*. Mapping models are subject to the following constraint:

$$id_C(C) \subseteq id_F(F). \tag{1}$$

This constraint provides a simple mechanism to decouple the software architecture from the platform model: Each component c encapsulates the functionality f that shares the same ID. Other than these IDs, components and functionalities are kept separate. As in AUTOSAR, the mapping must respect communication paths:

$$\forall (c_1, c_2) \in E_C,\ (map(c_1), map(c_2)) \in E_N^* \tag{2}$$

where E_N^* is the transitive closure of the node topology. Let $Map(S, P)$ be the set of all legal map structures for software model S and platform model P.

Model construction under the basic abstraction is polynomial-time solvable. For example, mapping every component onto the same node is guaranteed to satisfy communication constraints. However, model synthesis tools may have trouble dealing with this constraint, because a transitive closure operator is required.

3.2 Extended Abstraction

We illustrate extensibility by adding new concepts to the basic abstraction. Software components will be annotated with timing information, the platform will contain information about worst-case execution times (WCETs), and the mapping will have more constraints. This extension is based on the *logical execution time* (LET) semantics for *time-triggered architectures* (TTAs). TTA-based tools / systems are actively being developed by AUTOSAR-compliant vendors [12]. The resulting abstraction encodes an NP-hard multi-processor scheduling problem [14].

DEFINITION 4. **Extended Software -** *Let $S^e = \langle T, S, period, owner \rangle$ be a structure where T is a finite set, S is a software model, $period : T \rightarrow \mathbb{N}$ is a function, and $owner : C \rightarrow T$ is a function.*

The extended software model contains a set T of *timed modules*. Each component is owned by a module according to $owner(c)$. Associated with each module is a unit of time, called its period. The components owned by a module t execute periodically, once every $period(t)$ units of time. A module also acts as a clustering constraint; all components in a module must execute on the same node.

DEFINITION 5. **Extended Platform -** *Let $P^e = \langle P, wcet \rangle$ be a structure where P is a platform model and $wcet : F \times N \rightarrow \mathbb{N}$.*

The platform model is extended by a worst-case execution time matrix $wcet$, i.e. $wcet(f, n)$ is the WCET of functionality f on node n.

DEFINITION 6. **Extended Mapping -** *Let $M^e = \langle S^e, P^e, map \rangle$ be a structure where S^e and P^e are extended software and platform models. As before, $map : C \rightarrow N$.*

The extended mapping must ensure that a component c on a node n can execute once every $period(owner(c))$ units of time by considering the WCETs of c and all other components placed on n. Assume n runs components from exactly one module t, then the following constraint is sufficient for schedulability:

$$period(t) \geq \sum_{\{c | map(c) = n\}} wcet(n, f_c) \tag{3}$$

where f_c is the functionality f with the same id as c, $id_F(f) = id_C(c)$.

The constraint is more complex when two or more modules with differing periods are placed on the same node. Essentially, the components execute at different rates, so the least common multiple (lcm) of module periods must be used to compare modules. Given a node n let:

$$lcm(n) = lcm\left\{ period(owner(c)) \middle| map(c) = n \right\}. \tag{4}$$

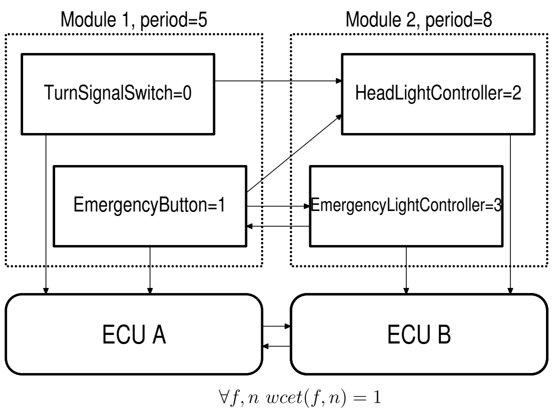

Figure 2: Example model under the extended abstraction benchmark.

The true sufficient condition for schedulability is:

$$lcm(n) \geq \sum_{\{c \mid map(c)=n\}} \frac{lcm(n)}{period(owner(c))} wcet(n, f_c). \qquad (5)$$

Components in the same module must be mapped to the same device:

$$\forall c, c' \in C,\; owner(c) = owner(c') \Rightarrow map(c) = map(c') \qquad (6)$$

and the mapping must be valid under the basic abstraction:

$$\langle S, P, map \rangle \in Map(S, P). \qquad (7)$$

3.3 Benchmark Instances

Figure 2 shows an example mapping model under the extended abstraction. This is based on the example in Figure 1, except that ports have been erased from components. The sensors/actuators are grouped into one module with period 5 time units, and the controllers are grouped into another module with a period of 8 time units. Assuming two ECUs with the topology shown and a WCET matrix where every functionality takes 1 time unit, then Figure 2 is a valid mapping model. (Functionalities are not shown.)

In our benchmark experiments a random model is generated, except that some parts of the model are unknown. Perhaps the *map* function is unknown or the timing specification is underspecified. A general synthesis algorithm must complete these missing details to form a total mapping model that satsifies all constraints. We encode this benchmark using the formalisms of various tools, so the definitions in Sections 3.1-3.2 may be rephrased, either out of necessity to apply a particular tool or to improve efficiency. Nonetheless, solving instances of this benchmark problem is expected to challenge existing tools, because synthesis requires: (1) reasoning about the finite transitive closure of a graph, (2) aggregating WCETs, (3) solving an NP-hard scheduling problem, and (4) computing LCMs. Each of these ingredients poses challenges to various approaches.

4. FORMULA

FORMULA is our specification language and analysis tool for model-based abstractions. It was first introduced in [18]. A formal pattern for composing and extending abstractions

```
 1. domain Software
 2. {
 3.    Component : (id: Integer).
 4.    [defined]
 5.    Connector : (from: Component,
 6.                   to: Component).
 7. }.
 8. model Ex1 : Software
 9. {
10.    c0 is Component(0), c1 is Component(1),
11.    c2 is Component(2), c3 is Component(3),
12.    Connector(c0, c2), Connector(c1, c2),
13.    Connector(c1, c3), Connector(c3, c1)
14. }.
```

Figure 3: Specifying the basic software model with FORMULA.

with FORMULA was first presented in [17]. FORMULA specifications consist of a set of *regular types* and a set of declarative rules in the form of *stratified logic programs*. In the interest of space, we informally present FORMULA using the running example.

4.1 Domains and Models

The basic specification unit for an abstraction is called a *domain*. A domain contains all the data types and constraints required to describe an abstraction. Lines 1-8 of Figure 3 encode the basic software model (Definition 1). Line 3 declares a new data type called Component for creating components. This declaration can be read like a *structure* in *C++*. Each instance of a component has a integer field called id. A component is instantiated by calling the Component constructor with an integral argument for its id, e.g. Component(0) creates a component with id of zero. As a matter of style, we use rich data structures to represent concepts, such as IDs, which might otherwise require additional functions (e.g. the id_C function in Definition 1). This is possible because two FORMULA instances are equal by *structural equality*, i.e. two instances are the same iff they were created by the same constructor with the same arguments.

The relation E_C is modeled by the Connector data type, which has two fields of type Component for indicating the source and destination of the component connection. A FORMULA *model* is just a finite set of data type instances satisfying all constraints of the associated domain. Lines 8-14 encode the software model (ignoring modules) of Figure 2. The notation c0 is Component(0) assigns the 0^{th} component to the identifier c0. This syntactic sugar allows instances to be used across a model without constructing them repeatedly. After expanding all identifiers, a model is just a set of data type instances.

A FORMULA model instantiates sets, relations, and functions using instances of data types. For example, to identify the set of components C in a FORMULA model M, one examines the Component instances. The function id_C is reconstructed by mapping each component to its id field: $id_C(Component(id)) \mapsto id$. The relation E_C is reconstructed according to:

$$(c_1, c_2) \in E_C \Leftrightarrow Connector \begin{pmatrix} Component(id_C(c_1)), \\ Component(id_C(c_2)) \end{pmatrix} \in M.$$

However, this last relationship depends on a technicality that every component appearing as an argument to a connector also appears directly in M. Consider the FORMULA model $Ex1'$ with a new instance added:

$$Ex1' = Ex1 \cup \left\{ Connector \left(\begin{array}{c} Component(100), \\ Component(100) \end{array} \right) \right\}. \quad (8)$$

The $Component(100)$ instance only exists as an argument to a connector, unlike all the other components that exist directly in $Ex1$. Such an connector contradicts our intent that connectors form a finite relation over the declared components. This situation is remedied by adding a constraint on the use of connectors. Line 4 attaches the *defined* constraint to the Connector data type, which disallows models where connectors contain components not defined in M. In FORMULA [defined] is called an *attribute*. Attributes attach themselves to the expression immediately following their introduction.

4.2 Logic Programs

The platform abstraction is slightly more complicated, because the transitive closure of the node topology must be computed. Figure 4 shows this specification. As in the previous example, the data types Node, Func, Channel encode the sets N, F, the injection id_F, and the relation E_N. Lines 8-10 compute the transitive closure using logic programming *rules*.

Given a set of data instances X, a rule examines this set and may create new instances as a result. Formally, rules are *inflationary functions* that grow the set of instances. Let $\mathcal{I}$ be the (infinite) set of all instances that can be formed according to the type system. A rule R defines a function $\Gamma_R : \mathcal{P}(\mathcal{I}) \rightarrow \mathcal{P}(\mathcal{I})$, called its *immediate consequence operator*. If $X \subseteq \mathcal{I}$, then R extends the set of instances according to $\Gamma_R(X) \supseteq X$. Given an initial set of instances X_0 a logic program applies rules until a fixpoint X_f is reached:

$$X_f = \Gamma_{k_n}(\ldots\Gamma_{k_2}(\Gamma_{k_1}(X_f))\ldots) \;\wedge\; X_0 \subseteq X_f. \quad (9)$$

Most logic programming languages restrict the form and order of application of rules so X_f exists and is the least fixpoint. In FORMULA the initial set X_0 is always a model (finite set of data instances) and the rules derive information about this model in the form of new data instances. FORMULA specifications correspond to *stratified logic programs*; stratification is a syntactic restriction on rules guaranteeing the existence of least fixpoints [8].

Returning to lines 8-10, these rules simultaneously introduce an auxiliary data type called nReach and use this data type to hold the transitive closure. Auxiliary data types must begin with a lower-case letter and can never appear in a FORMULA model. The first rule nReach(n,n) :- n is Node searches for Node instances. Every time an node is discovered, it is bound to the variable n. Next, the *head* of the rule extends the set of instances with the new data instance nReach(n,n), where n is substituted with its binding. Similarly, line 9 searches for channels and binds the channel end-points to the variables x and y. The recursive nature of transitive closure is captured by the final rule, which must be applied until no new nReach instances are created. FORMULA performs this forward computation until a fixpoint is reached to evaluate models against their domain specifications.

```
 1. domain Platform
 2. {
 3.   Node    : (id: Integer).
 4.   Func    : (id: Integer).
 5.   [defined]
 6.   Channel : (from: Node, to: Node).
 7.
 8.   nReach(n, n)  :- n is Node.
 9.   nReach(x, y)  :- Channel(x, y).
10.   nReach(x, z)  :- nReach(x, y), nReach(y, z).
11. }.
```

Figure 4: Specifying transitive closure of node topology.

```
 1. domain Mapping restricts Software * Platform
 2. {
 3.   [function]
 4.   Map : (c: Component, n: Node).
 5.
 6.   missingFunc :? c is Component,
 7.                  fail f is Func, c.id = f.id.
 8.
 9.   badTopology :? Connector(x, y),
10.                  Map(x, nx), Map(y, ny),
11.                  fail nReach(nx, ny).
12.
13.   conforms     :? !(missingFunc | badTopology).
14. }.
```

Figure 5: Defining legal maps through composition and constraints.

4.3 Domain Constraints

The previous examples imposed minimal constraints on their models. We now describe the mapping abstraction, which contains more complex constraints requiring additional features to encode. Figure 5 shows the specification of the mapping abstraction. Line 1 composes and imports the Software and Platform domains. (We describe this line in more detail shortly.) It suffices to know that the types, rules, and constraints of the other domains can be accessed within the scope of the Mapping domain. Line 4 introduces a data type Map for representing the assignment of components onto nodes. Map is annotated with the [function] attribute, so it must act like a function.

The first interesting constraint is that for every component there exists a functionality with the same ID (Equation 1). This constraint is expressed with a *query* called missingFunc. Like a rule, the body of a query is a pattern. Unlike a rule, a query sets a boolean variable to *true* if the pattern is matched at least once; otherwise it is set to *false*. The name of the boolean variable is given on the left side of the query operator (:?). The body of missingFunc tries to locate a component c for which there is no corresponding functionality f with the same id. The fail keyword is a logic programming construct called *negation-as-failure*. It allows a pattern to test for the absence of a data instance. The missingFunc query becomes *true* exactly when Equation 1 is not satisifed. The next constraint is that *map* must respect component communication (Equation 2). The badTopology query encodes this by trying to find a connector that is not mapped to nodes that (transitively) communicate. Again, the badTopology query is satisfied exactly when the constraint is violated.

These query definitions do not actually constrain the domain models. To turn them into constraints they must appear in a special query called conforms. Line 13 defines the special conforms query as a boolean combination of the previous queries. In this case, conforms is *true* when both missingFunc and badTopology are *false*. FORMULA tests if a model satisfies domain constraints by evaluating the logic program and checking if conforms evaluates to true. In fact, every domain has a conforms query, even if it is not explicitly defined. In the previous examples, the [defined] attributes simply added additional queries into the specification that were implicitly mentioned in conforms. In the Mapping domain, the [function] attribute also introduces additional queries an appends them to our explicit conforms query.

4.4 Composition and Extension

The specification of the Mapping domain is succinct because it can utilize the previous specifications in its context. However, uncontrolled importation of formal specifications can be dangerous. Constraints may contradict each other and logic programs may interact creating an unexpected net-effect. FORMULA provides several composition operators for controlled composition. We summarize a few of these.

Given two FORMULA domains D_X and D_Y, then the *pseudo-product operator* (*) forms a new domain $D_{X*Y} = D_X * D_Y$ whose models can be uniquely decomposed into a model of D_X and a model of D_Y. In other words, D_{X*Y} represents all the possible pairs of models. The pseudo-product operator is constructive. It forms the product domain by unioning both domains, creating a conforms query that conjuncts conformance of the original domains, and checking that the composite logic program is still well-behaved. Actually, the pseudo-product is slightly more flexible. It only guarantees unique decomposition if D_X and D_Y have disjoint type declarations. Disjointness can always be constructed using the *renaming operator* (as), which on-the-fly renames all declarations: $(D_X \text{ as } A) * (D_Y \text{ as } B)$. Now every type T_X in D_X is named $A.T_X$ and every type T_Y in D_Y is named $B.T_Y$. If the signatures overlap and renaming is not applied, then the models of D_{X*Y} will have an overlap in their decomposition.

Another useful operator is the *restriction* operator (restricts). The restriction operator imports the contents of one domain into another, but automatically conjuncts the conforms query of the imported domain onto the conforms query of the importer. It also performs static analysis to ensure that the composite logic program is well-behaved. The result is that the models of a restricted domain must use the imported abstractions correctly. Combining these operators give a powerful mechanism to modularize and compose abstractions. Returning to line 1 of Figure 5, the Mapping domain restricts the pseudo-product of Software and Platform. The pseudo-product sets the stage for describing maps from components to nodes. The restriction operator ensures that the new abstraction only adds constraints.

These composition operators can be used to describe a formal pattern for extending MDA abstractions. First, restricted domains are formed from the software and platform abstractions. These may contain new data types and constraints. Lines 1-11 in Figure 6 show the ExtSoftware and ExtPlatform domains, which define Modules, Owners and Wcets. Second, a pseudo-product is formed from the restricted software/platform domains and the original mapping domain (Line 13). Finally, this pseudo-product is further restricted to exclude those maps that are not valid according to the extended software/platform abstractions.

```
 1. domain ExtSoftware restricts Software
 2. {
 3.   Module  : (id: Integer, per: Natural).
 4.   [function]
 5.   Owner   : (c: Component, m: Module).
 6. }.
 7. domain ExtPlatform restricts Platform
 8. {
 9.   [function(f, n -> t)]
10.   Wcet  : (f: Func, n: Node, t: Natural).
11. }.
12. domain ExtMapping restricts
13.        Mapping * ExtSoftware * ExtPlatform
14. {
15.   badCluster     :? Owner(c1, m), Owner(c2, m),
16.                     Map(c1, n1), Map(c2, n2),
17.                     n1 != n2.
18.
19.   unschedulable :? nLCM(n, t), t <
20.                     sum(nCost(n, _),1).
21.
22.   conforms    :? !(badCluster | unschedulable).
23.
24.   onNode(n, c, p)  :- m is Module(_, p),
25.                        Owner(c, m), Map(c, n).
26.
27.   nLCM(n, t)  :- n is Node, t =
28.                lcm(onNode(n, _, _),2).
29.
30.   nCost(n, t)  :- onNode(n, c, p),
31.                f is Func(id), Component(id) =
32.                c, Wcet(f, n, w), nLCM(n, l),
33.                t = l / p * w.
34. }.
```

Figure 6: Composition pattern for adding new abstractions.

According to our benchmark, the ExtMapping domain must enforce a clustering constraint: Components in the same module go to the same device. This is easily specified in lines 15-17 with the badCluster query. The schedulability constraint is more difficult to specify, and is done using several auxilliary rules. The first rule (lines 24-25) records which tasks are mapped to which nodes with which periods. This information is stored using the auxiliary onNode data type. The next rule (lines 27-28) applies the *lcm aggregation operator* to all periods associated with a particular node. Aggregation operators find all instances of a particular form and then aggregate over their fields. In this case the lcm operator is aggregating over the second (zero-indexed) field of onNode instances. The final LCM of a node is stored

```
1. partial model Ex1Partial : ExtMapping
2. {
3.    c0 is Component(0), c1 is Component(1),
4.    ///More data instances from software model
5.    n0 is Node(0), n1 is Node(1),
6.    ///More data instances from platform model
7.    Map(c0, _), Map(c1, _), Map(c2,_),Map(c3, _)
8. }.
```

Figure 7: A partial model for defining a synthesis problem.

as an instance of the nLCM data type. The last rule records the cost of placing a component c on node n according to the wcet matrix and the lcm of n. The unschedulable query checks if the LCM of a node is too small to support all of its components. This completes the specification of the basic and extended abstractions using FORMULA.

4.5 Synthesis

In order to synthesize models, FORMULA specifications are translated into quantifier-free first-order formulas. This is accomplished by *symbolically executing* a logic program over a set of instances that may contain variables. We call this input set a *partial model* and it determines the parts of the model that must be solved. Unknowns can be placed anywhere and our framework will attempt to complete them. Figure 7 shows a partial model where the completion solves the mapping problem for the instance of Figure 2. Notice that the Map instances in line 7 receive underscores for their arguments. These underscores stand for fresh variables that must be instantiated. Partial models allow unknowns to be combined with parts of the model that are already fixed.

Let q be a query defined in domain D, then symbolic execution over a partial model generates a quantifier-free first-order formula $\varphi[\overrightarrow{x}]$ encoding all the ways that variables could be assigned values so the query q becomes *true*. Generating this formula requires unrolling all the paths through the logic program. Let $\overrightarrow{x}$ denote the vector of variables appearing in φ, then a *satisfying assignment* is an assignment of variables to values $\{x_1 \mapsto v1_, \ldots, x_n \mapsto v_n\}$ such that φ is *true*. Given a satisfying assignment for $\varphi[\overrightarrow{x}]$, a reverse translation converts this instance a into FORMULA model.

The expressiveness of FORMULA specifications may result in a formula φ containing elements from many different mathematical theories. For this reason we utilize the state-of-the-art *satisfiability modulo theories* (SMT) solver *Z3* [9] to construct satisfying instances. SMT solvers represent a significant step in automated theorem by soundly combining decision procedures for various theories while using efficient SAT techniques to drive the solving process. For example, symbolic execution over the unschedulable query (Figure 6) might impart the following fragment into φ:

$$\begin{gathered} test_{nLCM}(x) \;\wedge\; test_{nCost}(y_1) \;\wedge\; test_{nCost}(y_2) \;\wedge\; y_1 \neq y_2 \\ \wedge \\ 2Int(sel_{nLCM,1}(x)) < \begin{pmatrix} 2Int(sel_{nCost,1}(y_1)) + \\ 2Int(sel_{nCost,1}(y_2)) \end{pmatrix} \ldots \end{gathered}$$

This fragment tests if x is of the nLCM data type and if y_1, y_2 are of the nCost data type. If so, fields are extracted from instances using *selector functions*, e.g. $sel_{nLCM,1}(x)$ returns the first (zero-indexed) field of x. If $y_1 \neq y_2$, then the sum of these periods must be greater than the LCM of x. The function $2Int$ coerces fields into integers. SAT techniques provide a strategy for satisfying subformulas, and specific decision procedures actually solve the subformulas. In this example, three different decision procedures are required: (1) Term algebras (TA) for inductive data types encoded with testers and selectors, (2) linear arithmetic for summing WCETs, (3) uninterpreted function symbols for axiomatizing data type coercions ($2Int$).

5. EVALUATION

In order to compare our techniques with other approaches, we encoded the extended abstraction (Section 3.2) in two other tools: *Alloy* [16] and *SModels* [23]. In these experiments we do not evaluate the ease of specifying constraints, but focus entirely on the ability of tools the synthesize models. Our experiments are purely quantitative.

All the tools we test find models by converting specifications into quantifier-free SAT-like problems where search is directed by the DPLL algorithm. At first glance it might seem that tool evaluation is only testing the relative efficiency of the underlying SAT/SMT solvers. However, it turns out this is not the case because all tools accept specifications that are more expressive than quantifier-free first-order logic. As a result, the translation process into the underlying solver may be a source of exponential blow-up. Depending on the approach, this blow-up may or may not be avoidable.

5.1 Setup

We derive two parameterized benchmarks from the extended abstraction example. Note that in the following benchmarks we fix all module periods to 10 to avoid computing the LCM. FORMULA implements a careful encoding of LCM by unrolling a binary GCD algorithm. However, other tools do not have native encodings of LCM, so we decided to remove this rather specific operator from the evaluation.

The first benchmark, $Deploy(c, m, n)$ requires a tool to find a deployment for a random partial model containing c components owned by m modules onto n nodes. The component and node topologies, the WCET matrix, and the module ownership are randomly generated and may represent an unsatisfiable mapping problem. There are potentially n^m possible mappings to consider. The second benchmark, $Timing(c, m, n)$ requires a tool to find a WCET matrix that makes the system schedulable for a random mapping of c components owned by m modules onto n nodes. A random instance fixes the component and node topologies, the module ownership, and the mapping. Again, the random instance may be unsatisfiable. In principle, the size of this search space is unbounded as any WCET might be feasible. However, since modules have a fixed period of 10 only small WCETs are reasonable choices.

5.2 Background on Alloy

Alloy [16] is a modeling language based on first-order relational logic. Its flagship tool, the Alloy Analyzer, is equipped with a SAT-based engine that can be used to generate valid system configurations or counterexamples to a property. Due to the lightweight nature of the language and its powerful analysis, the Alloy Analyzer is gaining popularity among the model-based development community, with applications in a variety of domains [1, 5, 24].

At its core, Alloy is essentially first-order logic with built-in support for transitive closure, relation composition, and aggregation. It can be used to model components, platforms, and complex mapping constraints, as well as composition and extension of domain abstractions. Like FORMULA, Alloy supports arithmetic operations, which are crucial in specifying interesting properties of embedded systems. Both FORMULA and the Alloy Analyzer provide automated support for model synthesis. The combination of these three aspects makes the Alloy Analyzer a unique and suitable tool for comparison against FORMULA.

Although both of the tools provide the same kind of analysis, they differ in the underlying engines; FORMULA is based on symbolic execution of logic programs into an SMT solver, and the Alloy Analyzer encodes n-ary relations into a SAT solver. The SMT solver employs powerful decision procedures for theories, including linear arithmetic, uninterpreted functions, term algebras, and bit vectors. In comparison, the Alloy Analyzer compiles the entire specification into a propositional conjunctive-normal form (CNF), so only bit-level reasoning can be applied. This has implications on the translation overhead for Alloy.

The Alloy Analyzer, using its backend compiler called Kodkod [25], translates an input specification directly into an equisatisfiable CNF before presenting it to an off-the-shelf SAT solver. The underlying represention in Alloy is relations. For the purpose of translation into a boolean formula, Kodkod encodes every relation as a matrix, where each entry corresponds to a boolean variable that indiciates whether or not a particular tuple belongs to the relation. The size of the matrix is exponential in the arity of the relation. For example, consider a relation Map, which binds each component to some node. Let us assume that the upper bound on the number of Component and Node instances is 6. Then, the size of the resulting matrix is $6 \cdot 6 = 32$.

This underlying representation has significant implications on the performance of the Alloy Analyzer. Kodkod applies a number of optimizations in order to reduce the size of the resulting CNF, but in general, given k number of variables in the specification, the resulting CNF is exponential in size. Thus, when the degrees of freedom in the synthesis problem is large, Kodkod quickly runs out of memory capacity during the translation phase, even before reaching the SAT solver. In our experiments the Alloy engine preemptively rejects some synthesis problems by detecting that the resulting CNF will be too large.

5.3 Background on SModels

SModels [23] is a tool that implements the *stable model* semantics for *answer set programming* (ASP). The goal of ASP is to declaratively encode and solve problems using logic programming. For example, classic NP-hard problems like graph coloring and Hamiltonian cycle problems have very concise formulations as logic programs. In this sense, ASP is similar in spirit to FORMULA: An *answer set* is a set of facts (like a FORMULA model) under which some query evaluates to *true* in the logic program. ASP attempts to construct these answer sets.

However, the techniques to construct answer sets are different from the symbolic execution approach used by FORMULA. First, ASP uses a more general semantics for logic programs called the *stable model semantics* [10]. This semantics addresses problems when using negation-as-failure (NAF) recursively. Logic programs that do not use NAF in this way are called *stratified logic programs.* FORMULA only accepts stratified programs. Fortunately, the stable model semantics for stratified programs coincides with FORMULA's interpretation of stratified programs, so the tools can be compared directly.

The first phase in constructing an answer set is called *grounding*, which expands the logic program into one that contains no variables. Grounding is implemented by a related tool called *LParse.* SModels reads the results of LParse to construct an answer set. In our experiments the grounding process is a major source of exponential blow-up. If the program can be grounded before memory resources are consumed, then SModels can find solutions quickly. Once all variables are removed by grounding, SModels uses SAT techniques to derive answer sets (stable models) from the expanded program.

In order to ground the logic program LParse must have a bound on the range of values that can match an expression. For example, the following rule from the ExtMapping domain is problematic:

```
onNode(n, c, p) :- m is Module(_, p), Owner(c, m), Map(c, n).
```

The period p of a module m is an unbounded integer value. In order to ground this rule, an explicit finitization of the periods must be given in the form of a range. The declaration $period(0..10)$ creates a period relation that accepts the integers one through ten. Next, p must be guarded by this finite relation:

```
onNode(n, c, p) :- m is Module(_, p), Owner(c, m),
                   Map(c, n), period(p).
```

The grounding process creates instances of this rule for the possible substitutions of the variables. The larger the finitization the larger the grounded program.

5.4 Constructing Deployments

We studied two subproblems of the $Deploy(c, m, n)$ benchmark. First, we constructed random instances of the form $Deploy(5, 1, n)$ with $1 \leq n \leq 20$. This subproblem is polynomial-time solvable, because there is only one module involved. In theory, SAT/SMT solvers should be efficient at solving these instances, so this subproblem reveals the cost of translating into SAT. For SModels we finitize WCET times to be in the interval $[1, 20]$. However, aggregating WCETs may overflow this interval, so we define a related interval $max_{time} = [0, 20 \times n]$. Aggregation computations are ranged over this bigger interval. Alloy also requires a finitization of integers in the form a bit-width. We chose the bit-width to be: $bw = 1 + \lceil log_2(20 \times n) \rceil$. The addition of an extra bit is required because Alloy assumes integers are signed. Figure 8 shows the results. In SModels we observe the exponential cost of grounding the logic program as the finitization and number of nodes increases. Alloy continues to scale, though FORMULA exhibits the best results for this computationally easy subproblem.

The next benchmark consisted of instances of the form $Deploy(n, n, n)$ for $1 \leq n \leq 20$ (Figure 9). These problem instances are NP-hard to solve, and each instance is quadratically bigger than the previous. SModels does reasonably well for $n < 6$, after which all memory resources

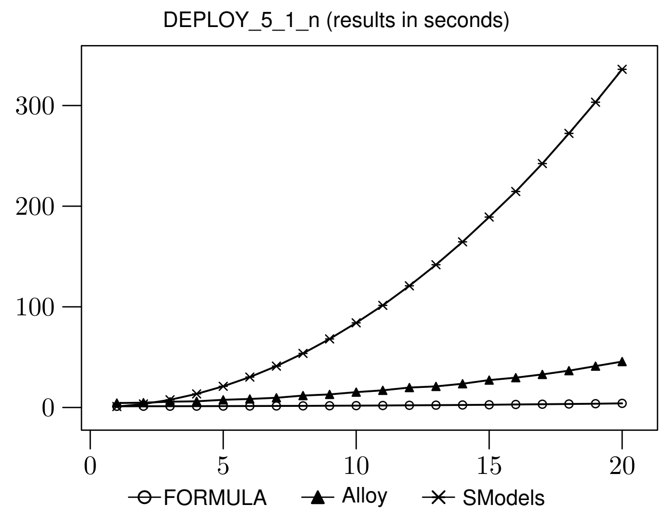

Figure 8: Generate deployments for random instances of five components in one module onto n nodes.

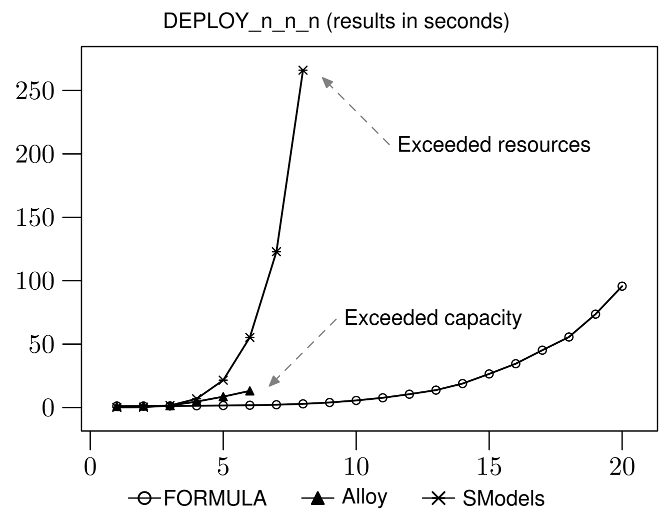

Figure 9: Generate deployments for random instances of n components, n modules, and n nodes.

are depleted. Alloy performs competitively for $n < 7$. After $n = 6$ Alloy preemptively rejects the problems due to the unacceptably large estimated cost of translation. FORMULA continues to scale reasonably well, with some hint of exponential growth for $n > 15$. This is the hardest benchmark for FORMULA.

5.5 Constructing Timing Specifications

We studied the same subproblems for the $Timing(c, m, n)$ benchmark. This benchmark should reveal the affects of unknowns that range over many possible values. Since the WCET matrix is unknown, there are $c \times n$ of these unknowns. Surprisingly, the $Timing(5, 1, n)$ benchmark yields similar results for all tools (Figure 10). In hindsight, the eager grounding of logic programs by LParse incurs the same translation cost regardless of whether the WCET matrix is given or not. A similar argument can be made for Alloy's representation of relations. FORMULA has a higher translation cost for this problem, but it is not enough to observably impact performance.

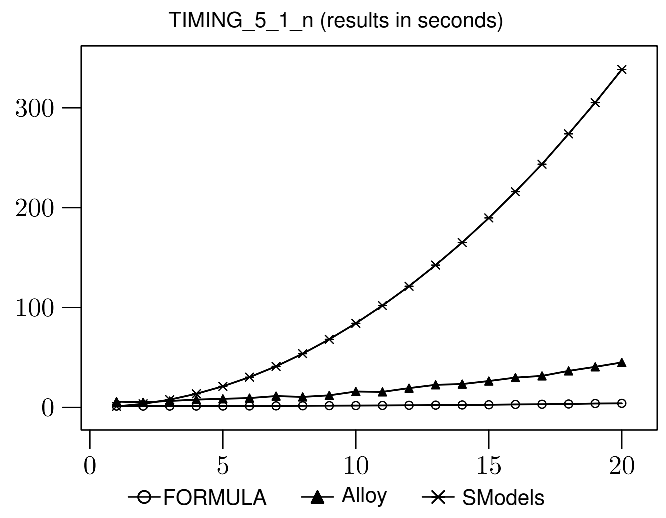

Figure 10: Generate a WCET matrix for random instances of five components in one module and n nodes.

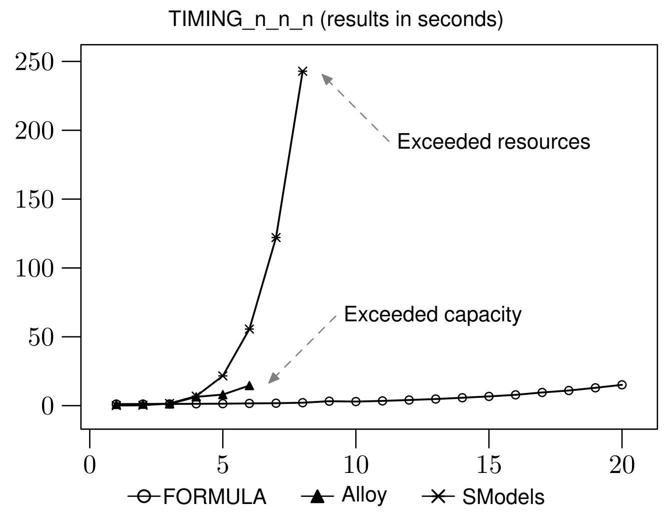

Figure 11: Generate a WCET matrix for random instances of n components, n modules, and n nodes.

The final benchmark $Timing(n, n, n)$ also exhibits similar results (Figure 11). Again SModels depletes resources for $n > 7$ and Alloy rejects problems for $n > 6$. Interestingly, FORMULA behaves better on this instance even though there are more degrees of freedom with larger ranges than the $Deploy(n, n, n)$ benchmark. We attribute this to the SMT solver's ability to efficiently reason about linear arithmetic. Since the mapping of components to nodes is already fixed, most of the entries in the WCET are actually irrelevant.

6. RELATED WORK

In addition to Alloy and SModels we describe some other related work. DESERT [21] is a framework for exploring design alternatives at the architectural level. It requires a representation of architectural variants as a tree of design alternatives. Boolean constraints can be added to this tree to model the cross-cutting impact of design choices. DESERT encodes design spaces using BDDs [7] and can instantiate legal models once the BDD is formed.

CoBaSa [20] is a tool for automating the assembly of commercial off-the-shelf (COTS) components. It compiles system requirements and constraints among components into a pseudo-Boolean satisfiability (PBSAT) problem, which is tackled by a constraint solver. Although CoBaSa uses constraint solving for model synthesis, it does not provide language level constructs for composing abstractions.

There are many synthesis tools for embedded systems that produce optimized implementations from specifications. Gries provides a survey of these tools [11]. However, most of these are tailored for a particular domain, and provide only a fixed set of abstractions and constraints. Our work focuses on providing a generic, flexible framework to support extensible specifications such as AUTOSAR. Also, we currently do not address the problem of optimization.

In software product lines, researchers have studied techniques to explore and analyze the space of product configurations based on feature diagrams [2]. These efforts focus on modeling feature interactions in the form of constraints, and do not typically deal with schedulability constraints.

7. CONCLUSION

In conclusion we have demonstrated that the FORMULA approach can be used to construct modular and extensible specifications for general MDA abstractions. We have provided MDA benchmarks derived from industrial case studies and compared our approach with others on these benchmarks. The results show that the FORMULA framework is a step towards general automation for MDA, and out-performs other state-of-the-art approaches. These results are not entirely due to the underlying SMT solver technology, but also rely on a careful translation of specifications into SMT formulas. For this task, we use an optimized symbolic execution engine to translate logic programs into SMT formulas.

8. REFERENCES

[1] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *MoDELS*, pages 436–450, 2007.

[2] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20, 2005.

[3] A. Benveniste, L. P. Carloni, P. Caspi, and A. L. Sangiovanni-Vincentelli. Heterogeneous Reactive Systems Modeling and Correct-by-Construction Deployment. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2003)*, pages 35–50, 2003.

[4] B. Best, J. Jürjens, and B. Nuseibeh. Model-Based Security Engineering of Distributed Information Systems Using UMLsec. In *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, pages 581–590, 2007.

[5] B. Bordbar and K. Anastasakis. MDA and Analysis of Web Applications. In *TEAA*, pages 44–55, 2005.

[6] M. Broy. Challenges in automotive software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE 2006)*, pages 33–42, 2006.

[7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[8] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.

[9] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, pages 337–340, 2008.

[10] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP/SLP*, pages 1070–1080, 1988.

[11] M. Gries. Methods for evaluating and covering the design space during early design development. *Integration*, 38(2):131–183, 2004.

[12] C. Hammerschmidt. Vector, TTTech win Audi Autosar software contract. *EE Times Europe*, (222900943), July 2010.

[13] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. L. Sangiovanni-Vincentelli, and M. D. Natale. Software Components for Reliable Automotive Systems. In *DATE 2008*, pages 549–554, 2008.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[15] M. Herrmannsdoerfer, S. Benz, and E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In *ECOOP 2009*, pages 52–76, 2009.

[16] D. Jackson. *Software Abstractions: Logic, language, and Analysis.* MIT Press, 2006.

[17] E. K. Jackson, D. Seifert, M. Dahlweid, T. Santen, N. Bjørner, and W. Schulte. Specifying and Composing Non-functional Requirements in Model-Based Development. In *Proceedings of the International Conference on Software Composition (SC 2009)*, pages 72–89, 2009.

[18] E. K. Jackson and J. Sztipanovits. Towards a formal foundation for domain specific modeling languages. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2006)*, pages 53–62, 2006.

[19] O. Kindel and M. Friedrich. *Software Engineering with AUTOSAR.* dpunkt, Heidelberg, 2009.

[20] P. Manolios, D. Vroon, and G. Subramanian. Automating component-based system assembly. In *ISSTA*, pages 61–72, 2007.

[21] E. Neema, J. Sztipanovits, and G. Karsai. Constraint-based design-space exploration and model synthesis. In *EMSOFT*, pages 290–305, 2003.

[22] Object Management Group. *MDA Guide Version 1.0.1*, 2003.

[23] T. Syrjänen and I. Niemelä. The Smodels System. In *LPNMR 2001*, pages 434–438, 2001.

[24] A. Tiberghien, P. Merle, and L. Seinturier. Specifying Self-configurable Component-Based Systems with FracToy. In *ABZ*, pages 91–104, 2010.

[25] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *TACAS*, pages 632–647, 2007.