llhsc: A DeviceTree Syntax and Semantic Checker

Vítor Rodrigues
VORTEX-CoLab
Vila Nova de Gaia, Portugal
vitor.rodrigues@vortex-colab.com

André Matos Pedro
VORTEX-CoLab
Vila Nova de Gaia, Portugal
andre.pedro@vortex-colab.com

***Abstract*—Feature models are a commonly used technique for modeling software variability and representing possible configurations of a system. This paper presents the llhsc tool, which is designed to generate and check device trees for custom hardware based on the constraints specified in a feature model. The form of these constraints is a set of logical formulas, which enables product line validation using off-the-shelf satisfiability solvers. For checking purposes, a new set of constraints is defined, which includes the specification of both syntax and semantic correctness of a delta-oriented software product line for DeviceTree bindings. This approach provides a more general and flexible solution for configuring static-partitioning hypervisors, but can also be used in systems without virtualization support, enabling the tool to be used in various contexts without sacrificing its generality. Through an empirical running example, we demonstrate the effectiveness of our approach, providing evidence that supports the construction of customized configurations for systems running static-partitioning hypervisors.**

***Index Terms*—Devicetree, hypervisors, embedded systems, software product lines, satisfiability modulo theories.**

I. Introduction

Embedded systems are used in a diverse range of applications, from small-scale consumer devices like IoT devices to complex industrial automation and control systems. As technology continues to evolve, the demand for more powerful embedded systems that can handle increasingly complex tasks is only going to grow. In addition to the technical challenges, there are also important considerations around the safety and security of embedded systems. These systems are often deployed in critical applications where failures can have serious consequences, such as medical devices or transportation systems. As a result, there will be an increasing focus on developing robust, secure, and reliable embedded systems that can withstand the demands of the future.

Traditionally, embedded systems have been built using a variety of custom hardware and software components that can be difficult to integrate and maintain. Nonetheless, the use of commercial off-the-shelf components, such as single-board computers (SBCs) with open instruction set architectures, paired with the adoption of open-source software, has made these systems easier to develop and deploy, while improving also their reliability and security.

One key technology that has helped to enable this shift is the DeviceTree (DT) [11], [13]. This device description language provides a standardized way of describing the hardware components of laptops, desktops, servers, and SBCs, which can make it easier to develop software that runs across different hardware platforms. By providing a uniform structure for describing the hardware components, the DT can help to simplify the development process and improve the flexibility and maintainability of the resulting software system.

When using custom hardware, managing a DT can be a difficult task because each device needs to be described separately. This is done using a data structure commonly referred to as DT binding. In practice, each platform-specific DT is likely to be written from scratch, where changes made to one binding may have an impact on other bindings, leading to compatibility issues and potential system failures.

To address these challenges, it is important to use powerful syntax and semantic checkers and other tools to effectively manage the DT and ensure that all nodes are compatible and functioning correctly. Proper management of the DT can help to ensure the reliability and performance of SBCs, particularly in safety-critical applications where failures can have serious consequences. The syntax and semantic checkers available today may not be able to detect all potential issues and ensure the compatibility of all nodes in the tree. Therefore, it is important to develop more powerful checkers to ensure the reliability and safety of embedded systems.

Without loss of generality, the llhsc tool is designed to generate and check configurations for open-source static-partitioning hypervisors, including Bao [14], [18], which are commonly used in the field of real-time and embedded systems. Others like Jailhouse [17] can also be supported. The tool helps developers and system integrators who are working with embedded systems and need to ensure that the DT is well-structured, correctly defined, and compatible with the hypervisor. By using a static-partitioning hypervisor like Bao, developers can ensure that each virtual machine (VM) or operating system (OS) has the necessary resources to perform its tasks without interfering with other parts of the system.

A. Running Example

Consider a simple SBC with a memory, a processor cluster and serial I/O ports. A DT describing this hardware is shown in Listing. 1. This hierarchical data structure is then stored in the DeviceTree source (DTS) file format. There are three top-level nodes, `memory`, `cpu` and `uart`, suffixed by a unit address (`@`). The description of the cluster is stored on the file `"cpus.dtsi"`, which is included in the main DTS.

Lets us choose `reg` as the property of interest. In the case of the `memory` node, `reg` specifies a memory consisting of two 64-bit memory banks, each one defined by four 32-bit addresses. The `reg` property specifies the address of the device

Listing 1: Example of a DT (.DTS file)

```
/dts-v1/;
/ {
    model = "Custom SBC";
    compatible = "sbc";
    #address-cells = <2>;
    #size-cells = <2>;

    /include/ "cpus.dtsi"
    /include/ "uarts.dtsi"

    memory@40000000 {
        reg = <0x0 0x40000000 0x0 0x20000000
               0x0 0x60000000 0x0 0x20000000>;
        device_type = "memory";
    };
};
```

within the address space defined by its parent bus. In this example, both `#address-cells` and `#size-cells` have the same value of 2, which means that each address range is described by a sequence of four (2+2) addresses.

Now consider that, by mistake, the user enters an address for the serial port that clashes with the address of one of the memory banks. Assume that this mistake happened when the base memory address of the second memory bank was added. In this situation, we say that the value of `reg` is semantically invalid although it is perfectly valid syntactically. The reason is that the second memory bank and the serial port both have the same base address, which leads to a runtime failure. Without careful examination, this kind of error can be detected at runtime only, and it is very difficult to trace it back to a consistent state. A purely syntactic tool, such as the DT Compiler (dtc) itself, is unable to detect this kind of error.

Currently, to the best of our knowledge, the only tool capable to extend the functionality of the compiler is the dt-schema [12] validation tool. This tool defines schema files that impose constraints on what data can be put into a DT. Essentially, for each node inside the tree, the tool determines which schema(s) are applicable and checks if the properties of the device node match the schema constraints. These checks are performed statically before compilation with the help of a parser internal to dt-schema.

An example of a semantic check performed by dt-schema is concerned with the size of the `reg` array. Depending on the values of `#address-cells` and `#size-cells`, accepted values for the array size are expressed in the form of an assertion. For instance, the device node `memory` in Listing. 1 is semantically correct for this rule, because there are 2 sub-arrays of size 4 inside `reg` (corresponding to two memory banks), and the semantic rule specifies that each sub-array must have size 4.

However, regardless of this structural check, the tool dt-schema is unable to detect the address clash between the `memory` and the `uart` device nodes, because the schema constraints cannot express relations between addresses. In concrete, it cannot define some rule that would verify that `0x60000000` (base address of `uart`) is lower than `0x80000000` (the ending address of `memory`), that is, it cannot detect the overlap between these two addresses. In this paper, we propose a constraint-based checking solution that extends the rules of dt-schema using the Z3 Theorem Prover, a satisfiability modulo theories (SMT) solver.

Another kind of error that cannot be detected by the dtc compiler or the dt-schema validation tool is that of dependency between nodes. For the hardware considered in our running example, it is natural to assume that some processing unit is a mandatory definition inside the DT. However, dt-schema does not enforce this rule for required device nodes, only for required properties. Furthermore, we may want to specify that if a processor is to be used, then the memory device *must* be defined. This kind of requirement can be expressed in terms of *features* and a set of valid combinations of features [10]. If every DT device node is a feature, then feature combinations encode the variability of the DT. Throughout this paper, the common variability of a DT is described by a feature model [2], [3].

Dependencies between device nodes assume greater importance in the context of virtualization. Let us consider that, on top of the hardware layout described in Listing. 1, where there exists a 2-CPU cluster that requires two DRAM memory banks to provide 64 bits of information at a time, runs the Bao hypervisor. As a safety requirement, we may impose that this hardware is partitioned such that one processor is exclusively assigned to a single VM, while the main memory is partitioned between the two VMs. This kind of specification is of much importance because the generation of Bao's configuration files is automatically obtained from a DT. Similarly to node dependency checks, the check against this kind of resource specification is done incrementally by solving constraints extracted from a feature model.

B. Contributions

While llhsc contributes as a practical tool to the state-of-the-art in the generation and the semantic checking of DTS files, this paper presents an approach towards both syntactic and semantic checking. The feasibility analysis of the approach is studied throughout this paper using the Bao static-partitioning hypervisor as the target platform.

The contributions of the paper are enumerated as follows:

1) The automatic generation of constraints that extend the syntax checks of dt-schema with a set of predefined semantic checks of memory addresses.
2) These constraints are defined as axioms in first-order predicate logic. Syntax and semantic correctness is defined as an SMT problem.
3) Configuration files for the hypervisor are automatically generated from a feature-oriented product-line of DTS files, which are syntactically and semantically correct.

C. Structure of the paper

The paper is organized as follows: Section II introduces the preliminary definitions and tools to support the remaining sections. Section III describes the approach to the DT resource

allocation problem in the context of virtualization and the generation of customized DTSs through the definition of a product line. Section IV and Section V show how the checking process is implemented, and its artifact evaluation, respectively, while Section VI presents the related work. Finally, Section VII draws the conclusions.

II. Preliminaries

A. Devicetree Specification

When compared to an everyday embedded system, an SBC has significantly more computational power, but typically it requires an OS for a complete function. As a result, the OS must have programmatic access to configurable hardware information, such as the bus clock-speed, during the boot process. In the early versions of Linux, the kernel included all of the hardware features, which resulted in increased complexity and reduced portability. By utilizing a DT, embedded software development can benefit from greater flexibility. Instead of compiling the Linux kernel for every custom hardware, the kernel can be compiled once and retrieve hardware information at runtime from a DT binary blob using the kernel's application binary interface (ABI). This approach enables the same kernel to run on different hardware without requiring recompilation, simplifying the development process.

The DT is essentially a hardware description language, specifically an OS-agnostic data structure (a tree), made up of device nodes. It defines the configuration of hardware components, such as processors, power units, memory, storage units, clock signals, etc., as well as how these components should operate. It also acts as a compatibility layer, guiding the OS on which device drivers to load. The use of DT bindings to describe hardware devices is not always necessary. However, in the case of an SBC with numerous hardware components, promoting driver integration becomes crucial.

One argument against using DT bindings is related to compatibility issues and the potential for additional errors resulting from adopting the DT language. These issues can include not only syntactic errors that are not detected by the DT compiler but also semantically incorrect information that is challenging to trace. For example, in the presence of an error, it is difficult to find which DT binding is actually causing the error. This aspect is precisely the focus of this paper, where we present an extensible verification mechanism based on a system of constraints.

The challenge of employing a DT structure is that certain properties, such as `#address-cells` and `#size-cells`, lack static semantics, meaning that their values have various interpretations based on their context within the tree. To illustrate this issue, let us examine the DT binding for the processor feature in Listing 2. In this example, the node is named *cpus* and has a `#address-cells` property of `0x1` and a `#size-cells` property of `0x0`. This means that the value of the `reg` property is interpreted as the processor's volume name – its physical number in decimal notation. Consequently, the two ARM Cortex-A53 processor cores are represented by two sub-nodes named `cpu@0` and `cpu@1`, respectively. Note

Listing 2: Example of a processor cluster DT binding

```
cpus {
    #address-cells = <0x1>;
    #size-cells = <0x0>;

    cpu@0 {
        compatible = "arm,cortex-a53";
        device_type = "cpu";
        enable-method = "psci";
        reg = <0x0>;
    };

    cpu@1 {
        compatible = "arm,cortex-a53";
        device_type = "cpu";
        enable-method = "psci";
        reg = <0x1>;
    };
};
```

that this differs from the interpretation of `reg` discussed in Section I-A, where `#address-cells` and `#size-cells` are used to parse an array of memory addresses. The dynamic semantics of a property like `reg` shows the need for semantic checks to ensure sound configurations.

An additional limitation of DT is its syntax, which is not expressive enough to represent relationships between device nodes. Therefore, the process of combining multiple DT bindings into a single DT is challenging and time-consuming, particularly as it needs to be repeated for every distinct SBC architecture. The lack of means to specify the correctness of a DT is crucial at two levels: first, at the top level, where it is essential to define the minimum set of device nodes required for a complete description of a particular hardware; and second, at the binding level, where certain properties, such as clock frequency, interrupt ports, and register area, must be configured based on the values provided by the manufacturer.

We propose using a software product line (SPL) [4], [16] to address these limitations. By separating the hierarchical organization of device nodes and their internal attributes, we can create generalized specifications of custom hardware where each hardware feature serves as a variability point (device node) that can be refined through a set of available feature realizations (bindings). In this way, each DT binding is a correct implementation of its respective variability point. We introduce the concept of a feature model as a compact representation of all products in an SPL for DTs.

B. Feature Model

A feature model [10], [19], [21] consists of a tree structure where each node represents a feature or a group of features, and the edges between the nodes represent the dependencies and relationships between them. Features are abstract aspects of a software system, described with domain vocabulary. Edge decorations define a partitioning of the child features of a given feature through a *consists of* relation that is designed to represent decomposition: one parent feature can have several sub-features grouped either by an AND, OR or XOR decomposition semantics. Cross–hierarchy constraints are composition rules that describe how features relate to one

another: one feature can require another one, or two features can be mutually exclusive. Features can also be identified as abstract, mandatory or optional.

The main advantage of feature models is the possibility to perform automated analysis over the set of features. The input to this process is a feature model translated into a particular representation, such as propositional logic, constraint programming, description logic or ad–hoc data structures, and the output is produced by using Boolean satisfiability problem (SAT) solvers or other specific algorithms. Examples of analysis are: detection if a feature model is *void* or not; checking if a given product is valid or not; generation of all valid products; detection of *dead* features, etc.

Feature models are commonly used in product line engineering, where they are used to manage the variability of a family of related software products. While individual products are specified using features, software product lines (SPLs) are specified using feature models. In this way, it is possible to identify the commonalities and variabilities across the different products, and to develop a systematic approach to product development reuse and customization. By design, an SPL starts from a base feature realization artifact, called a feature module, which contains common parts of all products. To build a product variant, feature modules are composed incrementally. In this paper, we implement a delta-oriented programming (DOP) [6] solution for DTS files.

The motivation behind the use of DOP to a product line of DTSs is because of the nature of DT itself: once the DT binding is defined and used by the Linux Kernel, it should no longer be changed anymore. It can only be extended if different types of peripheral integrated circuits and their definitions are not supported in the DT standard. The concept of *delta-module* is appropriate because each single DTS can be generated by applying a series of changes to the *core-module* of the product line. Each delta-module is then used to implement further products by adding, modifying and removing fragments of a DTS.

C. Hypervisor Configuration

Bao [14] is a static partitioning hypervisor designed for real-time and embedded systems. The hypervisor or VM monitor allows for the partitioning of resources, such as CPU, memory, and I/O devices, among multiple VMs or OSes. This can help to ensure that each VM is isolated and has access to the necessary resources to perform its designated tasks. The use of the Bao hypervisor improves the reliability and security of embedded systems allowing, at the same time, greater flexibility and control over the system's resources. Combined with a powerful syntax and semantic checker like llhsc, it may be possible to effectively manage the DT and ensure the compatibility of all nodes in the tree for a diversity of VMs.

An example of a platform configuration file for Bao is shown in Listing 3. In conformity with the running example given in Listing 1, two memory banks are defined at line 6, and one CPU cluster with two processor cores is defined at line 15. To achieve a complete hypervisor configuration, it is crucial to generate C files for every VM that is being controlled by the hypervisor. Later, in Section III-B, the automatic generation of the referred C files from DT source files is described.

Listing 3: Example of a Bao platform configuration C file

```
#include <platform.h>

struct platform_desc platform = {
  .cpu_num = 2,
  .region_num = 1,
  .regions = (struct mem_region[]) {
    { .base = 0x40000000, .size = 0x20000000 },
    { .base = 0x60000000, .size = 0x20000000 },
  },

  .console = { .base = 0x20000000 },

  .arch = {
    .clusters = {
      .num = 1, .core_num = (uint8_t[]) {2}
    },
  }
};
```

III. DTS Generation Process

A. Infer a feature model

The first step to define the DTS product line is the construction of the feature model. We can automatically extract the set of features from the DTS presented in Section I-A to define the product line, as shown in Fig. 1a. This feature model serves to limit the set of valid hardware configurations, where relationships between features come into play. In this feature model there are 12 valid products, and the root feature (`CustomSBC`) is present in all products within the SPL. The `cpus` feature is mandatory and, due to its exclusive-or (XOR) semantics, only one of its children can be selected.

We added a new type of device represented by the `vEthernet` feature, which is essential for VM communication. This feature is not present in the DTS described in Section I-A, because it represents virtual devices. The `uarts` and `vEthernet` features are both abstract and optional. The UART device node features can coexist in a product (OR), while the Ethernet device node features are mutually exclusive. We have also specified that if `veth0` is selected, then `cpu@0` must be selected (the same applies to `veth1` and `cpu@1`).

Fig.1 shows two examples of valid products from the feature model. In Fig.1b, the processor `cpu@0` is selected, which means that `cpu@1` cannot be selected in the same product. In Fig. 1c, the processor `cpu@1` is selected, which means that `cpu@0` cannot be selected in the same product. Both UART and Ethernet devices are used, but this time, `veth1` is used instead of `veth0`. Cross-constraints are also present in both products, specifying that the selected CPU device must use the corresponding Ethernet device. The cross-constraints, in this case, differ from those commonly found in SPLs, as the VMs need to coexist on the same platform, while typical SPLs generate products for a single machine/platform.

As mentioned in Section II-C, the complete configuration of the Bao hypervisor requires one DTS for the platform (acts as a platform configuration) and one DTS for each VM. We can

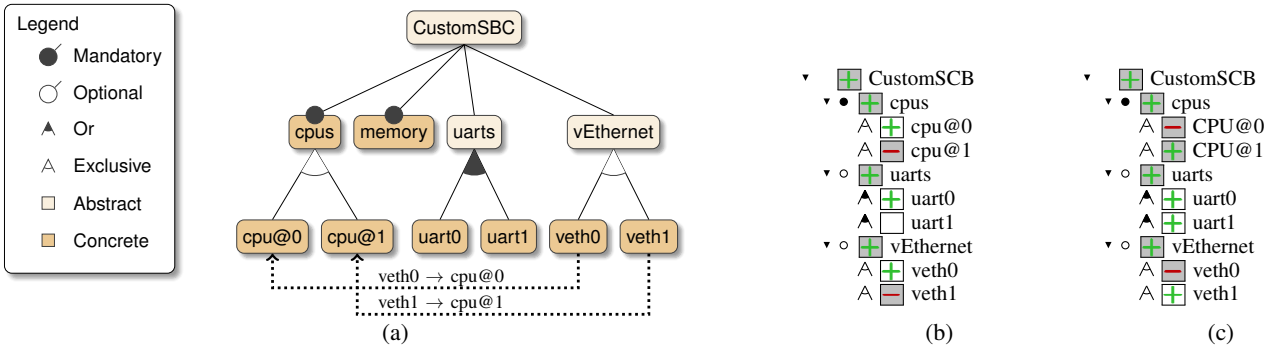

Fig. 1: The feature model for the running example includes two CPUs, one memory device, two serial devices, and two virtual Ethernet devices (a). Valid products of this feature model are illustrated in (b) and (c).

generate valid DTSs for each VM by taking the two products presented in Fig. 1, while the platform DTS is the union of selected features in both products.

B. Define the DTS product line

The goal of defining the product line for DT is to generate a DTS for both the hypervisor platform and VMs. Once the DTS files are created, the Listing 3 (discussed in Section II-C) can be automatically generated using a source-to-source transformation. To define the DOP product line, the core-module of the product is the DTS presented in Section I-A. However, since virtual devices cannot be included in the core-module, a set of delta-modules must be defined to modify the initial DTS with bindings for these virtual devices. Listing. 4 presents the set of deltas required to define two separate VMs.

To modify the core-module, four delta-modules are required, namely `d1`, `d2`, `d3`, and `d4`. The order in which these deltas are applied is determined by the propositional formula next to the **when** clause and the values specified next to the **after** clause, as per the semantics of DOP. Initially, the delta-modules are activated based on the feature selection. Subsequently, the application order is determined using the subset of active deltas.

Let us consider that in our running example, the hypervisor manages two VMs. The DTSs for the first and second VMs are generated from the feature configurations in Fig. 1b and Fig. 1c, respectively. The induced strict partial order between deltas for the first VM is `d3` < `d4` < `d2` while the second VM is `d3` < `d4` < `d1` .

The first delta, `d3`, *modifies* the root DT node (`/ `) in the core DTS by stating that the hypervisor uses 32-bit addresses (`#address-cells = <1>` and `#size-cells = <1>`) and introduces a new DT node called `vEthernet`. The second delta, `d4`, then *modifies* the `memory` DT node and defines two banks of 32-bit addressed memory. Finally, the third delta, `d2`, *adds* a DT node called `veth0@80000000` to the `vEthernet` node.

Although the partial order in the **after** clauses of delta-modules can establish dependencies between modifications of the same DT node, these dependencies only capture essential semantic requirements, name the well-formedness of the

Listing 4: The set of deltas that define the product line

```
delta d1 after d3 when veth0 {
    adds binding vEthernet {
       veth0@80000000 {
            compatible = "veth";
            reg = <0x80000000 0x10000000>;
            id = <0>;
        };
    }
}

delta d2 after d3 when veth1 {
    adds binding vEthernet {
       veth0@70000000 {
            compatible = "veth";
            reg = <0x70000000 0x10000000>;
            id = <1>;
        };
    }
}

delta d3 when (veth0 || veth1) {
    modifies / {
        #address-cells = <1>;
        #size-cells = <1>;
        vEthernet { };
    }
}

delta d4 after d3 when memory {
    modifies memory@40000000 {
       reg = <0x40000000 0x20000000
              0x60000000 0x20000000>;
    }
}
```

generated DTS. In fact, during delta-module application, it is not guaranteed that the resulting DTS is correct. As previously mentioned in Section II-A, syntax errors are only raised during the compilation of the generated DTS by the dtc compiler, but semantic errors are detected only at boot-time.

Thus, to achieve safety in the product line, it is necessary to extend the constraints expressing dependencies between delta names with additional constraints expressing syntax and semantic properties. In this way, if an error is detected by the checker, it can easily be traced back to the delta-module causing it. In the next Section, we present the checker component of lhsc, a constraint-based automatic mechanism that enables the generation of correct DTSs and, consequently,

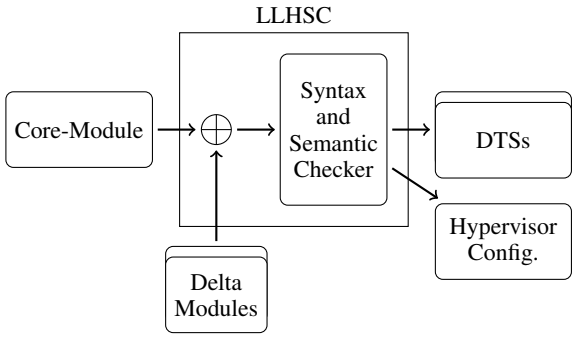

Fig. 2: The workflow of the lhsc tool.

correct configuration files for the Bao hypervisor.

Fig. 2 depicts the process of artifact generation, which implies adhering to a particular sequence involving the core-module, which is the DTS presented in Listing. 1, and the delta modules. Upon completion, the output consists of DTSs and a hypervisor configuration file where certain correctness properties are proven to hold by the checker.

IV. Constraint Checking

In this Section, we present the three kinds of constraints that the lhsc uses to ensure correctness. All these constraints are then solved using the Z3 API for Python. First, in Section IV-A, are presented the constraints used to build valid configurations in the context of static-partitioning employing feature models. Next, in Section IV-B, the syntax of DT bindings is checked against a set of constraints derived from the dt-schema specifications. Finally, in Section IV-C, a set of semantic constraints statically ensure that a given sort of VM configurations are valid and will run well in execution.

A. Resource Allocation Checker

This checker verifies that hardware configurations are correct by construction concerning their variability point through a refinement process controlled by the semantics of feature models. For one hypervisor configuration with k VMs, are required $k+1$ feature models, where every VM shares the same feature model with other VMs. The feature model for the hypervisor platform derives a particular type of feature model that is, in fact, a multi-product feature model, where static-partitioning is defined by an XOR relation over features that range across VM models.

For example, the sub-features of `cpus`, which are `cpu@0` and `cpu@1`, can be freely selected when configuring one VM, but in static-partitioning it is unreasonable to allocate the same CPU to different VMs. Therefore, the semantics of XOR will take into consideration the product, i.e. the VM, from where the feature was selected.

Consider a set of n sub-features of a feature model f, and let f_i^k be the i^{th} sub-feature of f, $1 \leq i \leq n$, selected in the product k, where k, $1 \leq k \leq m$, is the index of the VM. The constraint for exclusive resource usage across VMs is given by the following Boolean formula:

$$(f_1^1 \vee \cdots \vee f_1^m \vee f_2^1 \ldots f_2^m \vee f_n^1 \ldots f_n^m \Leftrightarrow f) \wedge \bigwedge_{i<j}^{k<l} \neg(f_i^k \wedge f_j^k) \wedge \neg(f_i^k \wedge f_i^l)$$

For the feature model in Fig. 1a, this constraint specifies that the 2 CPUs, `cpu@0` and `cpu@1`, are alternative sub-features in the same VM and, additionally, that `cpu@0` is exclusive to one VM. Since `cpus` is also a mandatory feature, this means that the maximum number of VMs is two ($m = 2$), and the assignment of CPUs is automatic (in Fig. 1 CPU features are grayed-out and cannot be selected by the user). The remaining feature relations, i.e. mandatory, optional and OR, correspond to the original semantics of feature models [10].

This checker guarantees that a set of features that violates the constraints is never selected by the user. In this sense, relations between hardware features, which are mapped into relation between DT nodes, are correct by construction.

B. Syntactic Checker

The verification of syntactic correctness is a fully automatic process, where constraints are extracted from dt-schema [12], and proof obligations are extracted from DT bindings. In this section, we present both kinds of constraints.

A fragment of the dt-schema specification of the memory DT node is shown in Listing 5. It states that there are two properties, `device_type`, which must have the constant value "memory", and `reg`, which is an array. Furthermore, it states that both these properties are *required*.

Listing 5: Fragment of the dt-schema for the memory DT node

```
properties:
  device_type:
    const: memory
  reg:
    minItems: 1
    maxItems: 1024

required:
  - device_type
  - reg
```

The constraints related to this dt-schema specification are given by the constraints (1) (2) and (3). The names of nodes and properties are encoded into strings using the hybrid theory in Z3, and the first-order formula $\forall x.R(x)$ specifies the presence condition of a given property x by means of the R predicate. Let $memory$ be the Boolean variable denoting the validity of the memory DT node, and $device\_type$ and reg be the string variables denoting the corresponding properties in the schema. The set of constraints derived from the schema include the following:

$$R(device\_type) \rightarrow (const \leftrightarrow \text{"memory"}) \quad (1)$$

$$memory \rightarrow R(device\_type) \wedge (device\_type \leftrightarrow \text{"device\_type"}) \quad (2)$$

$$memory \rightarrow R(reg) \wedge (reg \leftrightarrow \text{"reg"}) \quad (3)$$

Constraints (2) and (3) specify that `device_type` and `reg` are required properties, as shown in Listing 5.

Proof obligations are extracted directly from the instance of the DT binding. Constraint (4) states that the `memory` node was found and that the value of the `device_type` is "memory". Since both properties `device_type` and `reg` were found, we define the constraint 5 to state the Boolean condition (OR) for the presence of these properties.

$$const \leftrightarrow \text{"memory"} \tag{4}$$

$$\forall x.C(x) \leftrightarrow (x \leftrightarrow \text{"reg"} \vee x \leftrightarrow \text{"device\_type"}) \tag{5}$$

$$\forall x.(C(x) \rightarrow R(x)) \wedge (\neg C(x) \rightarrow (\neg R(x))) \tag{6}$$

Constraint (6) specifies that predicate $R(x)$ will be satisfied if condition $C(x)$ is met. If the condition is not satisfied, then the default value for all properties not captured in constraint (5) is $\neg R(x)$. Unlike the method described in Section IV-A, where the user must manually select or remove all hardware features, the syntactic checker automatically generates a complete closure over the properties.

C. Semantic Checker

Purely syntactic methods do not ensure that our product line process is *sound*. Here, soundness essentially means that the process does not introduce collisions between mutually exclusive memory addresses. This property is of major importance because the addresses inside the DTSs of the VMs must be translated into their machine counterparts internally to the hypervisor, which are then mapped into the physical layer [7] (2-stage translation). If there is an error inside a DTSs, the entire virtual environment is compromised.

Consider the case where the user forgets to update the `memory` node inside core-module by omitting the delta `d4` in Listing 4. Because dt-schema assumes that any multiple of the sum obtained from `#address-cells` and `#size-cells` is valid, it fails to capture the truncation from 64-bit to 32-bit addresses that were introduced by `d3`. This example shows that static-partitioning hypervisors require more granularity in the semantic checks.

The memory consistency between an arbitrary number of memory banks can be defined in first-order logic by the following formula:

$$\neg \bigvee_{i<j} \begin{pmatrix} \exists x_1, x_2.\ (b_i \leq x_1 \wedge (b_i + s_i) > x_1) \wedge \\ (b_j \leq x_2 \wedge (b_j + s_j) > x_2) \wedge \\ (x_1 < x_2) \end{pmatrix} \tag{7}$$

where (b_i, b_j) is any ordered pair of base addresses, and s_i and s_j are the corresponding sizes. The technique of bit-blasting is used by the Z3 theorem prover to encode memory addresses inside bit-vectors which are then translated into a SAT problem. In case of unsatisfiability of the negation of (7), a counter example of consistency is produced by Z3.

The running example (Section I-A and later extended in Section III-B), shows a simple case where the state-of-the-art tool dt-schema fails – the `reg` properties can be syntactically correct but semantically invalid. For the sake of simplicity, we assume that the virtual and physical addresses of both UART are the same. While dt-schema fails to detect the error caused by the missing conversion from 64 to 32-bit, our checker can find an actual collision on the address `0x0`, because four banks of memory are found, instead of the original two.

Listing 6: Example of a Bao VM Configuration for all resources of Listing 1 (without partitioning)

```
#include <config.h>
VM_IMAGE(vm, vmimage.bin);

struct config config = {
  CONFIG_HEADER
  .vmlist_size = 2,
  .vmlist = {
    { .image = {
        .base_addr = 0x40000000,
        .load_addr = VM_IMAGE_OFFSET(vm),
        .size = VM_IMAGE_SIZE(vm)
      }
    },
    .entry = 0x40000000,
    .cpu_affinity = 0b11,
    .platform = { .cpu_num = 2, dev_num = 2,
    .region_num = 2,
    .regions = (struct mem_region[]){
      { .base = 0x40000000 ,  .size = 0x20000000 },
      { .base = 0x60000000 ,  .size = 0x20000000 }
      },
      .devs = (struct dev_region[]){
      /* from uarts.dtsi */
      { .pa = 0x20000000,
        .va = 0x20000000,  .size = 0x1000 },
      { .pa = 0x30000000,
        .va = 0x30000000,  .size = 0x1000 },
      },
    },
    .ipc_num = 1,
    .ipcs = (struct ipc[]) {
      { /* veth0 device */
        .base = 0x70000000, .size = 0x00010000,
        .shmem_id = 0,
      }
    },
  },
  .shmemlist_size=1,
  .shmemlist = (struct shmem[]) {
    [0] = { .size = 0x00010000 }
  },
};
```

V. Empirical Evaluation

This section details the empirical assessment of our running example. Our llhsc checker was initially designed as a tool but has since evolved into a cloud service that we invite readers to explore at https://llhsc.apps.vortex-colab.com. The reader can access the paper's running example, which includes the full generation of the two DTS files and the Bao configuration file for the `CustomSBC`. Listing 6 is also available for viewing at the aforementioned artifact location. Listing 6 exemplifies one VM configuration (or product) using all hardware resources listed in Listing 1 but without partitioning. The configurations automatically generated from the SPL can be utilized not only in Bao hypervisor but also in other virtualization solutions such as QEMU. Additionally, these configurations are compatible with SBCs that use aarch64 or RV64 architecture.

VI. Related Work

Approaches to the automatic generation of DT configurations from embedded system designs can be found in [1] and [9]. The DSML4DT [1] tool accomplishes automatic generation of DTSs using a domain-specific modeling language inside a model-driven environment, where the Acceleo [8] model query language is used to express validation rules, such as resource allocation, static semantics and user-defined constraint checks. The GRIP [9] framework aims at the automation of IP-integration into system-on-chip (SoC) designs, where constraints are used to restrict the design space when generating all candidate SoCs, before DTS code generation.

In contrast to these model-based approaches, our approach follows the DOP-SPL [6] methodology. Artifact reuse is accomplished using delta-modules instead of models, thus avoiding the model-to-text transformations. Because DTS are usually shipped unfinished and upgraded later, the SPL of DTSs is more suited to such unstable codebases. Whereas model query languages are used in [1] to verify only properties of the models, our approach encodes all properties of interest, namely the variability of feature models and the syntactic and semantic correctness of DT bindings in a single constraint satisfaction problem.

Different uses of constraints are proposed in [20] to address the structure of hardware configurations. The configuration algorithm is written in Prolog through constraint logic programming. The algorithm for resource allocation is automatic and tends to explode in complexity. In our setting, the allocation problem involves assigning resources to VMs within a hypervisor environment using a feature model. Because of the structural complexity characteristics of feature models, this problem is efficiently handled by the SAT-solver [15].

Apart from embedded system designs, low-level validation of DTS code is a subject of debate in the embedded Linux community. In the Request for Comments (RFC) [5], it was suggested that contemporary standards, such as XML Schema, should be used to represent device trees. Several attempts were made to define a DTS-like schema language, culminating with the adoption of dt-schema as the de facto tool for DT validation. The validation for particular bindings implemented directly in C code was also a topic of discussion. In our approach, syntax and semantic checks are both expressed as formulas in first-order logic. The incremental nature of the Z3 solver allows for easy extensibility, as constraints can be added incrementally to the same solver instance.

VII. Conclusions

The approach we have presented offers a constructive way to safely configure embedded systems through the use of a feature model and a series of deltas applied to a master DTS. The llhsc tool ensures the correctness of the DOP product line through a set of constraints defining the syntactic and semantic aspects that can be proven correct by the Z3 SMT solver. The generation of proof obligations for syntactic correctness is done by composing the schemas of individual DT bindings, while semantic validation of memory addresses and interrupts is performed using bit-vector constraints. The extensibility of the llhsc tool allows for the incremental addition of constraints to the same Z3 instance, making it a versatile tool for future work. Overall, the effectiveness of this approach has been demonstrated in the configuration of a hypervisor, providing a safe and efficient method for resource allocation.

Acknowledgments

This work is supported by the European Union/Next Generation EU, through Programa de Recuperação e Resiliência (PRR) [Project Route 25 with Nr. C645463824-00000063].

References

[1] S. Arslan and G. Kardas, "DSML4DT: A domain-specific modeling language for device tree software," *Comput. Ind.*, vol. 115, p. 103179, 2020.

[2] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*, ser. Lecture Notes in Computer Science, vol. 3714. Springer, 2005, pp. 7–20.

[3] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[4] P. Clements and L. M. Northrop, *Software product lines - practices and patterns*, ser. SEI series in software engineering. Addison-Wesley, 2002.

[5] B. Cousson and F. Parent. Device tree schemas and validation. (accessed on 2022-03-17). [Online]. Available: http://lists.infradead.org/pipermail/linux-arm-kernel/2013-October/201449.html

[6] F. Damiani and I. Schaefer, "Dynamic delta-oriented programming," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, 2011.

[7] C. Devigne, J. Bréjon, Q. L. Meunier, and F. Wajsbürt, "Executing secured virtual machines within a manycore architecture," *Microprocess. Microsystems*, vol. 48, pp. 21–35, 2017.

[8] Eclipse Foundation. Obeo. Acceleo. (accessed on 2022-03-17). [Online]. Available: http://www.eclipse.org/acceleo/

[9] M. Jassi, Y. Hu, D. Mueller-Gritschneder, and U. Schlichtmann, "Graph-grammar-based IP-integration (GRIP) - an EDA tool for software-defined SoCs," *ACM Trans. Design Autom. Electr. Syst.*, vol. 23, no. 3, pp. 40:1–40:26, 2018.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., Nov. 1990.

[11] G. Likely and J. Boyer, "A symphony of flavours : Using the device tree to describe embedded hardware," in *ELC*, 2008.

[12] ——, "Device tree schemas and validation," in *[RFC 00/15]*, 2013.

[13] J. Madieu, *Linux Device Drivers Development: Develop Customized Drivers for Embedded Linux*. Packt Publishing Ltd, 2017.

[14] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *NG-RES@HiPEAC*, ser. OASIcs, vol. 77. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 3:1–3:14.

[15] M. Mendonca, A. Wąsowski, and K. Czarnecki, "SAT-based analysis of feature models is easy," in *Proceedings of the 13th International Software Product Line Conference*, 2009, p. 231–240.

[16] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[17] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits! (almost)," *CoRR*, vol. abs/1705.06932, 2017.

[18] B. Sá, J. Martins, and S. Pinto, "A first look at RISC-V virtualization from an embedded systems perspective," *IEEE Trans. Computers*, vol. 71, no. 9, pp. 2177–2190, 2022.

[19] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Networks*, vol. 51, no. 2, pp. 456–479, 2007.

[20] A. Schüpbach, A. Baumann, T. Roscoe, and S. Peter, "A declarative language approach to device configuration," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 5:1–5:35, 2012.

[21] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund, "Abstract features in feature modeling," in *SPLC*. IEEE Computer Society, 2011, pp. 191–200.