# A Formal Model for the Automatic Configuration of Access Protection Units in MPSoC-Based Embedded Systems

Tobias Dörr, Timo Sandmann, and Jürgen Becker
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
Email: {tobias.doerr, sandmann, becker}@kit.edu

***Abstract*—Heterogeneous system-on-chip platforms with multiple processing cores are becoming increasingly common in safety- and security-critical embedded systems. To facilitate a logical isolation of physically connected on-chip components, internal communication links of such platforms are often equipped with dedicated access protection units. When performed manually, however, the configuration of these units can be both time-consuming and error-prone. To resolve this issue, we present a formal model and a corresponding design methodology that allows developers to specify access permissions and information flow requirements for embedded systems in a mostly platform-independent manner. As part of the methodology, the consistency between the permissions and the requirements is automatically verified and an extensible generation framework is used to transform the abstract permission declarations into configuration code for individual access protection units. We present a prototypical implementation of this approach and validate it by generating configuration code for the access protection unit of a commercially available multiprocessor system-on-chip.**

***Index Terms*—Multiprocessor system-on-chip, on-chip isolation, system-level isolation, access protection, model-based design, information flow tracking, code generation, safety, security.**

## I. Introduction

In order to fulfill their steadily increasing performance demands, modern embedded systems increasingly rely on multicore processors. A popular manifestation of such devices are multiprocessor system-on-chip (MPSoC) platforms. They integrate multiple processing cores along with memory hierarchies, input/output (I/O) controllers, and similar peripherals on a single chip. Especially heterogeneous MPSoCs, which comprise diverse processing cores, are promising platforms for the cost-, area-, and power-efficient implementation of applications with high performance requirements [1]. Due to the tight integration and the fact that their cores share many of the on-chip resources, however, their use in safety- and security-critical systems is a challenging endeavor.

From a dependability perspective, *safety* is the property that a system is free of catastrophic consequences on users or the environment, while *security* describes the property that unauthorized actions do not result in the disclosure or alteration of information [2]. In this paper, we refer to a system whose malfunction can result in it becoming unsafe as a *safety-critical* system and to one whose malfunction may result in it becoming insecure as a *security-critical* system.

Two specific issues associated with the use of MPSoCs in safety- and security-critical environments can be understood by considering the field of autonomous driving: Here, increasing communication bandwidth and performance requirements facilitate a trend towards centralized electrical/electronic (E/E) architectures, in which powerful components such as heterogeneous MPSoCs host a variety of functions, often with different real-time requirements and safety criticalities [3]. Despite the fact that these functions are connected to the same on-chip resources, they must be designed in such a way that they meet their respective real-time requirements and are unaffected by possible failures of less critical functions. Therefore, a sufficient degree of logic-level isolation as well as measures against timing interferences are mandatory [4]. Furthermore, vehicle-to-vehicle (V2V) and vehicle-to-infrastructure (V2I) communication are an essential part of an autonomous vehicle's functionality. This means that additional, often wireless, external interfaces are present and closely connected to the same components that execute the safety-critical functions described above. This can result in attack surfaces that must be counteracted by the application of suitable security mechanisms such as the isolation of individual subsystems [5].

Therefore, commercially available MPSoC platforms such as the Zynq UltraScale+ MPSoC (ZynqMP) from Xilinx [6] comprise dedicated hardware units that control how the individual bus masters, which are most importantly the processing cores, access shared resources. These units are configured at runtime with information on which transactions are legal and which are illegal. After their configuration, they monitor the global interconnect and prevent illegal transactions from reaching their respective destinations. In the specific case of the ZynqMP, there are two such units [7]: the Xilinx memory protection unit (XMPU) and the Xilinx peripheral protection unit (XPPU). They operate on addresses from the global address space and together cover the majority of the platform's memory-mapped resources. In their behavior, they are comparable to memory protection units (MPUs). However, their scope is not limited to transactions from a specific bus master. It extends to all transactions on the global interconnect.

For the purposes of this paper, we refer to a hardware unit that acts on the logical level of some communication link and enforces a fixed or configurable set of access protection rules on this link as an *access protection unit* (APU). During

the design of MPSoC-based embedded systems with safety or security requirements, the proper configuration of APUs is of vital importance. However, determining such a configuration is often complicated by the following aspects:

1) Each type of APU requires its configuration data in a particular form. For runtime-configurable units on commercially available MPSoCs, for instance, it is usually necessary to determine the values of a platform-specific set of configuration registers. Performing this process manually is both time-consuming and error-prone. Software tools to generate these values are provided by some semiconductor vendors, but they are still highly MPSoC-specific and often not well suited for integration into existing development toolchains.
2) The task that an APU performs is a low-level mechanism with the goal of enforcing certain information flow policies. These policies usually originate from some functional, safety-related, or security-related requirements. Therefore, considering the APU configuration in isolation makes it difficult to verify that it successfully leads to the desired information flow policies.
3) MPSoCs in embedded systems are often integrated into a network of different components. As a consequence, resource isolation at the level of an MPSoC can have a significant impact on the information flow at system level. These system-level implications of an APU configuration must therefore be considered.

To tackle these issues, we present a formal model to describe both information flow requirements and access protection settings of a system in an abstract manner, i.e., without the need to consider the characteristics of specific APUs. Furthermore, we describe an approach to check instances of this model for their consistency and present a framework to map them to concrete platform architectures with a particular set of APUs. Based on the model instance and this mapping, the framework performs a feasibility check and, if possible, derives a suitable configuration for each of the utilized APUs automatically.

## II. Related Work

There is a large volume of published approaches to achieve resource isolation on MPSoCs. Some of them isolate processing cores as much as possible from each other. The architecture proposed in [8], for instance, is based on "local systems" that comprise resources to which the corresponding processing cores have exclusive access. In this approach, physically shared memory is protected by a memory controller that arbitrates transactions from the local systems in a deterministic manner and forwards them to disjoint memory partitions.

Many processing platforms map shared resources into the global address space. It is then possible to employ MPU-like structures to protect the resources from specific masters. The approach presented in [9] targets system-on-chip (SoC) platforms with a single processor whose tasks need to be isolated from certain memory regions and I/O controllers. Similar approaches that are targeted at multi-master platforms are described in [10] and [11]. These approaches are comparable to the APU mechanism described above. Note that such an isolation operates mainly on a logical level and is less effective against timing interferences [4]. In practice, additional measures to ensure that the interferences do not cause a violation of real-time requirements might be necessary [12].

To isolate parts of a system from each other, the concept of *information flow tracking* (IFT) can be seen as an alternative to strict physical isolation or APU-based approaches [13]. Its underlying idea is to statically or dynamically determine where information flows to. An important advantage of these approaches over access protection schemes is that they capture not only how information is released from a source or introduced into a sink but also its propagation through the overall system [14]. Many such schemes, such as the one proposed in [15], focus on information flows within or between software programs. However, IFT has also been applied on the level of logic gates to analyze and prohibit unintended information flow over $I^2C$ and USB links [16]. Due to the fact that this approach captures interactions on the granularity of individual bits, it is able to detect implicit information flow, such as through timing behavior. The approach that we propose is similar to the IFT concept in the sense that it does not consider access protection units in isolation. Instead, it captures how they affect the propagation of information. However, our work is limited to the explicit flow of information. The consideration of implicit flows is beyond the scope of this paper.

In order to deal with the steadily increasing complexity of embedded systems, various model-based approaches with the goal to automatically analyze or validate system properties, particularly at early stages in the design process, have been proposed. The goal of the Architecture Analysis & Design Language (AADL), an SAE standard [17], is to capture the architecture of hardware/software systems in a single model, which then serves as the basis for automated analyses and transformations. Conceptually similar to our approach is the AADL-based design methodology presented in [18]. Like our work, this model-based approach enforces isolation policies between SoC partitions at run-time. However, the authors limit the scope of their work to safety considerations on a single SoC and explicitly consider real-time aspects and timing interferences. While they base their model on AADL and generate hardware wrappers for individual processors, we use a custom model and provide a flexible framework to generate configuration code for existing APUs.

## III. Concept and Methodology

At the core of the proposed concept is a model capturing both the information flow requirements of and the envisaged transactions within a considered system. It can be applied to any environment in which *execution units* (= *units*), which are hardware components to execute functions, are connected via so-called *communication links* (CLs). A communication link describes a transmission channel to which one or more units are attached. It is logically shared between all its units and realizes the read and write transactions that they exchange. A transaction is initiated by its *master unit* and targeted at an

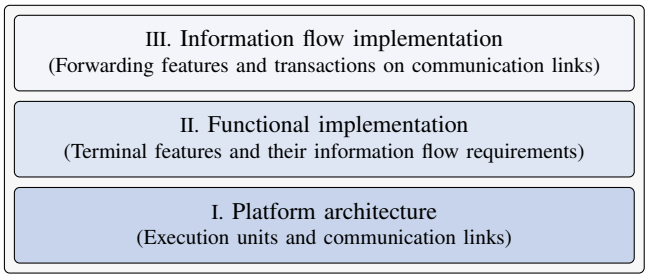


Fig. 1. Model layers and the information they capture

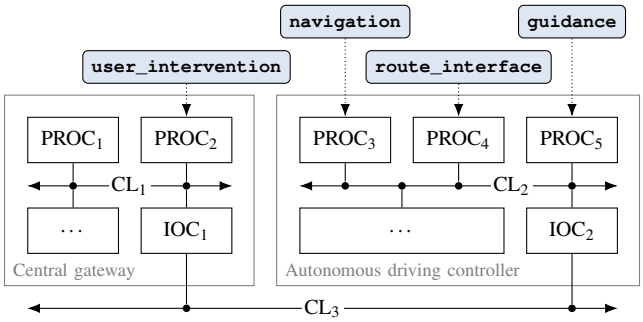


Fig. 2. Mapping of terminal features to a platform architecture

explicitly addressed *slave unit*. The model assumes that all units are equal in the sense that they can both initiate and be addressed by a transaction. A slave unit that receives a read transaction returns its response as part of this transaction, i.e., does not initiate another transaction to do so. The definition is deliberately broad to accommodate various types of transmission channels, including off-chip and system-level interconnects such as $I^2C$ or CAN buses. In the following, we assume that APUs base the decision of whether a specific transaction is legal or illegal on its master unit, its slave unit, and its type (read or write). An example of such an APU is one of the above-mentioned system isolation units of the ZynqMP. Note that in practice, a slave unit will usually dispatch a received transaction to a specific function. We assume that an APU is entirely unaware of such unit-internal aspects.

### A. Overview of the Model and its Layers

This section gives a fist overview of the model, while Section IV describes it in a formal manner. Fig. 1 shows the three layers of the proposed model. They capture the dependencies between model fragments in the sense that each layer depends on and extends the layers that are depicted below it.

The starting point of every model instance is an existing platform architecture and an existing functional specification. The *platform architecture* is a particular hardware setup, while the *functional specification* describes the functionalities that the corresponding platform architecture shall deliver.

The platform architecture forms layer I of the model. It consists of a set of units, a set of CLs, and their connections. If applicable, it also describes the execution environments (operating systems, ...) that the units provide. APU-specific aspects are not captured at this layer. Instead, certain portions of the platform architecture (a specific MPSoC, ...) can be linked to a *platform-specific generator*. Such a generator is invisible to the higher model layers and translates the model parts that are relevant to this portion to APU configurations.

Instead of the functional specification per se, layer II contains *terminal features* that describe how the developer intends to implement the required functionality on the platform architecture. Each of them is mapped to exactly one unit and represents a specific function. An example of such a terminal feature is a task of an operating system. They serve as sources and sinks of so-called *information flow*. Furthermore, layer II captures the information flow requirements that the developer defines between terminal features. They describe

1) which information flows are required for the set of terminal features to fulfill their functionalities and
2) which information flows can be accepted from an isolation perspective beyond the required ones.

Layer III of the model then captures how these information flows are planned to be implemented. In order to implement information flows over a chain of units, it is possible to introduce so-called *forwarding features* as part of this layer. Each forwarding feature is mapped to and executed by exactly one unit, but its sole purpose is to forward every piece of information that it receives. An example of such a forwarding feature is the functionality fulfilled by an $I^2C$ controller of an MPSoC that forwards data from an MPSoC-internal link to an $I^2C$ bus. To implement the flows, the developer defines write and read transactions that originate from a feature, propagate over a specific CL, and lead to another feature. Depending on the type of transaction, information will flow either from the master feature to the slave feature or the over way around.

Note that information flow requirements can only be specified between terminal features. Forwarding features are merely a mechanism to help fulfill these requirements.

As a concrete example, consider the guidance and navigation tasks that according to [19] are essential functions in the field of autonomous driving. Assume that an autonomous vehicle shall deliver a navigation functionality, which determines a route to travel, and a guidance functionality, whose goal is to derive suitable maneuvers from this route. Via an external interface, a user shall be able to influence the navigation. Furthermore, the route that the navigation generates needs to be validated and passed to the subsystem implementing the guidance functionality over a dependable, strictly controlled interface. For safety reasons, no other functionality shall have an impact on the guidance subsystem. Based on a specific platform architecture, the developer translates these requirements into the terminal features `navigation`, `route_interface`, `guidance`, and `user_intervention`. Furthermore, the following three required information flows are defined:

- `navigation` → `route_interface`,
- `route_interface` → `guidance`, and
- `user_intervention` → `navigation`.

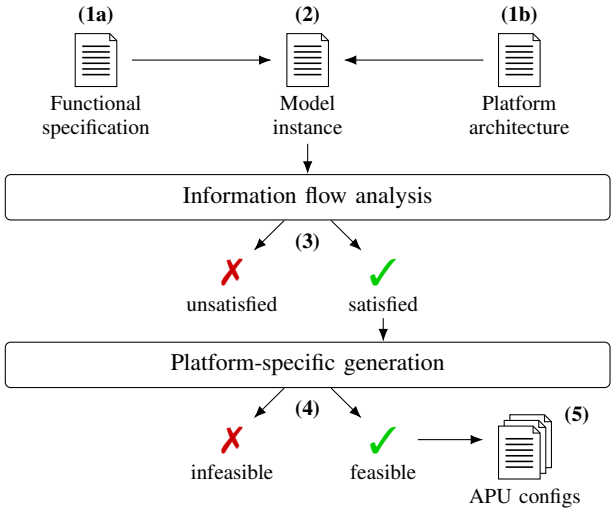


Fig. 3. Proposed design methodology

The layer-II mapping of these terminal features to a sample platform architecture is shown in Fig. 2. PROC blocks in the figure are processor subsystems with private memories that communicate directly over their communication links, while the two IOC blocks represent I/O controllers attached to $CL_3$. Assume that all shown CLs are protected by an APU.

Layer III of the model now requires the developer to specify how these flow requirements are fulfilled: The first two required flows are satisfied, for instance, if `navigation` uses $CL_2$ to write to `route_interface`, while `guidance` uses $CL_2$ to read from `route_interface`. To fulfill the third requirement, forwarding features on $IOC_1$ and $IOC_2$ are required. If `user_intervention` uses $CL_1$ to write to the feature on $IOC_1$, $IOC_1$ uses $CL_3$ to write to the feature on $IOC_2$, and $PROC_3$ uses $CL_2$ to read from the feature mapped to $IOC_2$, the required flows are satisfied. The transaction specifications can be used to derive APU configurations for $CL_1$, $CL_2$, and $CL_3$. During runtime, these configurations enforce that no other flows can occur over the shared CLs. Note that if `guidance` could read from the forwarding feature on $IOC_2$, for instance, this would enable an unaccepted flow from `user_intervention` to `guidance`.

### B. Model-Based Design Methodology

The overall design methodology that we propose as part of this work is shown in Fig. 3. It depends on a given functional specification (1a) and knowledge of the platform architecture (1b) on which this specification shall be realized. Based on these inputs, the developer derives a particular model instance (2). This derivation involves taking certain design decisions such as the definition of terminal features. The model instance is then used as the basis for an automated information flow analysis (3). In particular, this step verifies whether the transactions specified in layer III enable all required flows to occur and prevent any flow that is neither required nor accepted from taking place. If these two conditions are satisfied, model portions that are linked to a platform-specific generator are passed to this generator. It will check if it is able to translate all transactions on links for which an APU protection is requested into suitable APU configurations (4) and, if this is the case, derive them automatically (5).

It is possible to run through these process steps in an iterative manner, i.e., to start with a partial model instance and extend it incrementally. In such an approach, the developer is guided by the model and its verification capabilities through the process of finding an appropriate isolation setting.

Recall that the definition of a communication link is deliberately broad. In practice, a typical CL will therefore exhibit only a subset of the aspects that the model is able to capture. Units that are attached to a memory-mapped AXI4 interconnect, for instance, can act either as a master or as slave unit, i.e., are not equal. In such a case, the platform-specific generation step should additionally ensure that the envisaged transactions are feasible before generating an APU configuration.

### C. Dependability and Protocol Transactions

In safety- and security-critical systems, failure of system elements to deliver their intended functionalities must be anticipated. With respect to our model, we must therefore consider the possibility that a terminal feature unintentionally forwards information from its input to its output. Furthermore, one has to consider that any feature might misbehave and initiate a transaction that is targeted at the wrong destination.

Another aspect that a suitable model must be able to capture are systematic or random faults of unit: If a unit is affected by a fault, all the features that are mapped to it might behave incorrectly. A unit that normally isolates the features that it executes from each other, such as a processor running an operating system that isolates its processes, might fail to enforce this isolation if it is affected by a fault itself.

Therefore, the model that we propose considers all features and units to be undependable by default. In safety- and security-critical systems, however, one can assume that suitable mechanisms are applied in order to make individual features or units dependable. For these entities, the above assumption is overly pessimistic. In the model, both terminal and forwarding features as well as units can therefore be declared as *dependable*. During the information flow analysis, these entities are then treated accordingly.

Dependable entities are particularly important in combination with *protocol transactions*, the last fundamental aspect of the model. In practice, some transactions on CLs carry only protocol data. Consider again the architecture shown in Fig. 2, for instance. In order for $PROC_2$ to transfer payload data over $CL_3$, payload data must flow from $PROC_2$ to $IOC_1$, but protocol data such as ready or error flags will usually flow from $IOC_1$ to $PROC_2$ as well. In our model, such transactions are captured as protocol transactions. If all the features and units that are involved in such a transaction are dependable, it is often justified to assume that it will not allow for information flow in a strict sense. This aspect is again taken into account during the information flow analysis.

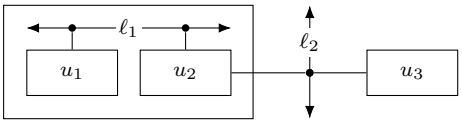


Fig. 4. Platform architecture comprising three units and two links

## IV. Formal Definition of the Model

In the following paragraphs, each of the three model layers is formally defined as a tuple of sets, relations, and functions.

*Definition 1:* A *platform architecture* is a tuple

$$M_P = (U,\, L,\, C,\, c_0,\, \varphi_C,\, \varphi_L,\, \delta_U),$$

where $U$, $L$, and $C$ are sets of *units*, *links*, and *containers*, respectively. $c_0 \in C$ is the *root container*, which shall serve as the direct or indirect parent of all units, links, and other containers. These model entities will be referred to as

$$E \coloneqq U \cup L \cup (C \setminus \{c_0\})$$

in the following. $\varphi_C \colon E \to C$ maps each such entity to its enclosing container, while $\varphi_L \colon L \to \mathcal{P}(U)$ maps each link to the units that it is connected to. $\delta_U \colon U \to \{0,1\}$ maps every unit to its dependability, where a value of 0 signifies an undependable and a value of 1 a dependable unit.

Containers represent hierarchy aspects of the platform architecture and can be arbitrarily nested. Their main purpose is to serve as the formal basis of the platform-specific generation. A particular MPSoC with several units and links can for example be modeled as a container. For convenience, we define the function $\hat{\varphi} \colon E \times C \to \{0,1\}$ such that $\hat{\varphi}(e,c) = 1$ if and only if $e$ is directly or indirectly contained in $c$.

*Definition 2:* A platform architecture $M_P$ is valid if and only if the following conditions are satisfied:

$$\begin{aligned} \forall e \in E &: \ \hat{\varphi}(e, c_0) = 1 \\ \forall c \in C \setminus \{c_0\} &: \ \hat{\varphi}(c, c) = 0 \\ \forall \ell \in L \ \forall u \in \varphi_L(\ell) &: \ \hat{\varphi}\left(u, \varphi_C(\ell)\right) = 1 \end{aligned}$$

In other words, a platform architecture is valid if and only if all units, links, and containers except for the root are contained in the root, no container is contained in itself, and no link is connected to a unit "outside" of its own container.

To model the platform architecture in Fig. 4, for instance, we define $U = \{u_1, u_2, u_3\}$, $L = \{\ell_1, \ell_2\}$, $C = \{c_0, c_1\}$, and choose $c_0$ as the root of the architecture. Setting

$$\begin{aligned} \varphi_C(u_1) = \varphi_C(u_2) = \varphi_C(\ell_1) = c_1, \\ \varphi_C(u_3) = \varphi_C(\ell_2) = \varphi_C(c_1) = c_0, \end{aligned}$$

as well as $\varphi_L(\ell_1) = \{u_1, u_2\}$ and $\varphi_L(\ell_2) = \{u_2, u_3\}$ leads to a valid $M_P$ that describes the given architecture accurately.

*Definition 3:* A *functional implementation* is a tuple

$$M_F = (T,\, I_R,\, I_A,\, \sigma_T, \delta_T)$$

that is based upon a specific $M_P$. $T$ is the set of terminal features, while $I_R, I_A \subseteq T \times T$ with $I_R \cap I_A = \varnothing$ are relations representing required information flows and flows that are not required but still accepted, respectively. $\sigma_T \colon T \to U$ maps every terminal feature to a unit of the platform architecture, while the function $\delta_T \colon T \to \{0,1\}$ is comparable to $\delta_U$ and maps every terminal feature to its dependability.

The first element of an $I_R$ or $I_A$ tuple represents the source and the second element the sink of a flow. Note that the purpose of $I_A$ is to capture information flows that are entirely optional. All required flows will be implicitly treated as accepted and are therefore not contained in $I_A$.

*Definition 4:* An *information flow implementation* is a tuple

$$M_I = (F,\, \sigma_F,\, \delta_F,\, X_W,\, X_R,\, X_L,\, \omega_P,\, \omega_L)$$

that is based on a specific $M_F$. $F$ is a set of forwarding features for which $F \cap T = \varnothing$ is satisfied. $\sigma_F \colon F \to U$ and $\delta_F \colon F \to \{0,1\}$ are defined analogously to $\sigma_T$ and $\delta_T$. The relations $X_W, X_R \subseteq (T \cup F) \times L \times (T \cup F)$ represent the write and read transactions on the links, respectively. Furthermore, the relation $X_L \subseteq (T \cup F) \times (T \cup F)$ captures all information flows that occur within of a particular unit. For every transaction in $X_W$ or $X_R$, $\omega_P \colon (X_W \cup X_R) \to \{0,1\}$ is 1 if and only if the transaction is a protocol-only one. Finally, the function $\omega_L \colon L \to \{0,1\}$ defines whether or not transactions on a specific link shall be protected by an APU, where a value of 1 represents the APU-protected case.

Note that for $X_W$ and $X_R$, the first value of a 3-tuple represents the feature initiating the transaction (*master feature*), while the third value is the feature that receives it or responds to it (*slave feature*). Which of these features is the source and which is the sink of information flow depends on the transaction type. If the transaction is a protocol transaction, it might even be possible that it will not be associated with any information flow. It is important to understand that the local information flows captured by $X_L$ need to be interpreted differently: Here, the first value of the tuple represents the flow source, while the second one represents the flow sink.

For brevity, we define the function $\sigma \colon F \cup T \to U$ such that $\sigma(x) = \sigma_T(x)$ if $x \in T$ and $\sigma(x) = \sigma_F(x)$ otherwise. Analogously, we define $\delta \colon F \cup T \to \{0,1\}$ in such a way that $\delta(x) = \delta_T(x)$ if $x \in T$ and $\delta(x) = \delta_F(x)$ otherwise.

*Definition 5:* An information flow implementation $M_I$ is valid if and only if the following conditions are satisfied:

$$\begin{aligned} \forall (m, \ell, s) \in (X_W \cup X_R) : \big(\{\sigma(m), \sigma(s)\} \subseteq \varphi_L(\ell) \\ \wedge\, \sigma(m) \neq \sigma(s)\big) \\ \forall (j,k) \in X_L : \sigma(j) = \sigma(k) \end{aligned}$$

In other words, it is valid if and only if every transaction from a master to a slave feature takes place over a link that is connected to the distinct units of these features and the endpoints of local flows are mapped to the same unit.

*Definition 6:* A *model instance* is a 3-tuple

$$M = (M_P, M_F, M_I)$$

in which $M_I$ is based on $M_F$ and $M_F$ is based on $M_P$. It is valid if and only if both $M_P$ and $M_I$ are valid.

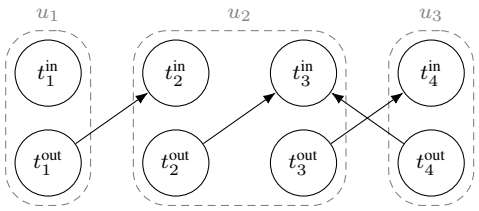


Fig. 5. $G_\alpha$ constructed from a sample model instance

## V. Information Flow Analysis

The starting point for an information flow analysis is a valid model instance $M$. In layer III of this instance, the developer has captured how the required information flows ($I_R$) from layer II are intended to be realized from a technical perspective. The analysis checks if this realization

1) allows all required information flows ($I_R$) to occur and
2) ensures that all the information flows that are potentially feasible are either required or accepted ($I_R$ or $I_A$).

Regarding the generation of APU configurations, it is important to understand that the transactions that are defined on links for which $\omega_L$ returns 1 can be seen as concrete requirements for the APU of this link. For the second analysis, we assume that all the involved APUs are able to fulfill their requirements exactly and do not fail to enforce them during runtime.

In order to check the first condition, we are interested in a function $\alpha\colon T \to \mathcal{P}(T)$ that maps each terminal feature to all the terminal features to which information that it outputs normally flows, i.e., under strict consideration of $F$, $X_W$, $X_R$, and $X_L$. For the second condition, we are interested in a function $\beta\colon T \to \mathcal{P}(T)$ that maps each terminal feature to all the terminal features to which information that it outputs may potentially flow, either nominally or if any undependable unit or feature of the system misbehaves.

*Definition 7:* A model instance $M$ satisfies its information flow requirements if and only if the following conditions hold:

$$\forall (j,k) \in I_R :\ k \in \alpha(j)$$
$$\forall j \in T\ \forall k \in \beta(j) :\ (j,k) \in I_R \cup I_A$$

To determine $\alpha$ for a given model instance, we transform this instance into a directed graph $G_\alpha$. In this graph, every feature $x \in T \cup F$ is represented by two nodes: an input node $x^{\text{in}}$ and an output node $x^{\text{out}}$. For every forwarding feature $f \in F$, a directed edge from $f^{\text{in}}$ to $f^{\text{out}}$ is added. The remaining edges are constructed from the $X_W$, $X_R$, and $X_L$ values. More specifically, edges are added

- from $m^{\text{out}}$ to $s^{\text{in}}$ for every $w \coloneqq (m,\ell,s) \in X_W$ that is not a protocol transaction, i.e., for which $\omega_P(w) = 0$,
- from $s^{\text{out}}$ to $m^{\text{in}}$ for every $r \coloneqq (m,\ell,s) \in X_R$ that is not a protocol transaction, i.e., for which $\omega_P(r) = 0$, and
- from $j^{\text{out}}$ to $k^{\text{in}}$ for every $(j,k) \in X_L$.

With this, $\alpha$ is constructed by considering every $t \in T$ and mapping it to all the $\tau \in T \setminus \{t\}$ for which $G_\alpha$ contains a directed path from $t^{\text{out}}$ to $\tau^{\text{in}}$. As an example, consider a system with $U = \{u_1, u_2, u_3\}$, $T = \{t_1, t_2, t_3, t_4\}$, and $F = \varnothing$, where $t_1$ is mapped to $u_1$, both $t_2$ and $t_3$ are mapped to $u_2$, and $t_4$ is mapped to $u_3$. Under the assumption that all these units are connected to $\ell \in L$ and that none of the specified transactions is declared as protocol-only, the specifications $X_W = \{(t_1,\ell,t_2),(t_4,\ell,t_3)\}$, $X_R = \{(t_4,\ell,t_3)\}$, as well as $X_L = \{(t_2,t_3)\}$ lead to the $G_\alpha$ shown in Fig. 5. From this graph, one can derive the values $\alpha(t_1) = \{t_2\}$, $\alpha(t_2) = \alpha(t_4) = \{t_3\}$, and $\alpha(t_3) = \{t_4\}$.

The derivation of $\beta$ is a more intricate problem. In order to determine the potentially feasible information flows, the possibility that every single entity might fail or misbehave must be taken into consideration. Recall, for instance, that a terminal feature is assumed to act as the sink for information

**Algorithm 1:** $G_\beta$ construction from a model instance $M$

$G_\beta \leftarrow (V,E)$, $V \leftarrow \varnothing$, $E \leftarrow \varnothing$
**for each** $\ell \in L : \omega_L(\ell) = 0$ **do**
  **if** $\exists\, u \in \varphi_L(\ell) : \delta_U(u) = 0$ **then**
    $V \leftarrow V \cup \{\pi_\ell\}$ ▷ Add link sharing node
**for each** $u \in U : \delta_U(u) = 1$ **do**
  **for each** $\ell \in \varphi_L(u)$ **do**
    $V \leftarrow V \cup \{p^{\text{in}}_{\ell,u}, p^{\text{out}}_{\ell,u}\}$ ▷ Add port nodes
    **if** $\pi_\ell \in V$ **then**
      $E \leftarrow E \cup \{(p^{\text{out}}_{\ell,u}, \pi_\ell), (\pi_\ell, p^{\text{in}}_{\ell,u})\}$
**for each** $u \in U : \delta_U(u) = 0$ **do**
  $V \leftarrow V \cup \{\pi_u\}$ ▷ Add unit sharing node
  **for each** $\ell \in \varphi_L(u) : \omega_L(\ell) = 0$ **do**
    $E \leftarrow E \cup \{(\pi_u, \pi_\ell), (\pi_\ell, \pi_u)\}$
**for each** $x \in F \cup T$ **do**
  $V \leftarrow V \cup \{x^{\text{in}}, x^{\text{out}}\}$ ▷ Add feature nodes
  **if** $x \in F \ \vee\ \delta(x) = 0 \vee \delta_U(\sigma(x)) = 0$ **then**
    $E \leftarrow E \cup \{(x^{\text{in}}, x^{\text{out}})\}$
  **if** $\delta_U(\sigma(x)) = 0$ **then**
    $E \leftarrow E \cup \{(\pi_{\sigma(x)}, x^{\text{in}}), (x^{\text{out}}, \pi_{\sigma(x)})\}$
**for each** $\nu \in X_W$ **do** HandleWrite($\nu$)
**for each** $\nu \in X_R$ **do** HandleRead($\nu$)
**for each** $(j,k) \in X_L$ **do** $E \leftarrow E \cup \{(j^{\text{out}}, k^{\text{in}})\}$

**Algorithm 2:** Procedure to handle write transactions

**procedure** HandleWrite($\nu$)
  $(m,\ell,s) \leftarrow \nu$ ▷ Extract values from the 3-tuple
  masterUnitDep $\leftarrow \delta_U(\sigma(m)) = 1$
  slaveUnitDep $\leftarrow \delta_U(\sigma(s)) = 1$
  **if** slaveUnitDep **then**
    $E \leftarrow E \cup \{(p^{\text{in}}_{\ell,\sigma(s)}, s^{\text{in}})\}$ ▷ Add the listening edge
  **if** masterUnitDep $\wedge$ slaveUnitDep **then**
    **if** $\omega_P(\nu) = 0 \vee \delta(m) = 0 \vee \delta(s) = 0$ **then**
      $E \leftarrow E \cup \{(m^{\text{out}}, s^{\text{in}})\}$
  **if** masterUnitDep $\wedge\ \neg$slaveUnitDep **then**
    $E \leftarrow E \cup \{(m^{\text{out}}, \pi_{\sigma(s)})\}$
  **if** $\neg$masterUnitDep $\wedge$ slaveUnitDep **then**
    $E \leftarrow E \cup \{(\pi_{\sigma(m)}, p^{\text{in}}_{\ell,\sigma(s)})\}$
  **if** $\neg$masterUnitDep $\wedge\ \neg$slaveUnitDep **then**
    $E \leftarrow E \cup \{(\pi_{\sigma(m)}, \pi_{\sigma(s)})\}$

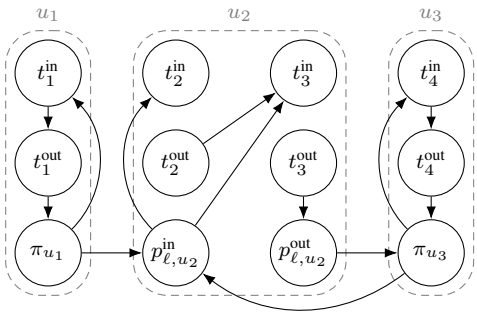


Fig. 6. $G_\beta$ constructed from a sample model instance

that it receives. However, it is possible that the unit executing the feature becomes affected by a random fault or that the implementation of the feature is affected by a systematic fault. In such a case, one must assume that this feature unintentionally forwards information that it receives.

In order to derive $\beta$, we once again transform the model instance into a graph $G_\beta$. During this transformation, we assume that a feature $x \in F \cup T$ unintentionally forwards information and fails to recognize protocol transactions as such if and only if $\delta(x) = 0$ or $\delta_U(\sigma(x)) = 0$ are satisfied. Furthermore, we assume that a unit $u \in U$ with $\delta_U(u) = 0$ does not isolate the features that are mapped to it from each other and has no control over the transactions that its features respond to or initiate. In contrast, a unit $u \in U$ with $\delta(u) = 1$ is assumed to enforce a strict isolation between its features, permit features to initiate only those transactions that are explicitly specified, and ensure that features receive or respond to only those transactions that originate from links to which they have to listen to according to the specification. Finally, we assume that a link $\ell \in L$ for which $\omega_L(\ell) = 0$ does not perform any access protection. A link $\ell \in L$ with $\omega_L(\ell) = 1$ is assumed to be protected by an APU that is configured in such a way that it permits only those transactions that are necessary to implement $X_W$ and $X_R$. The procedure to derive $G_\beta$ from a model instance $M$ is shown in Algorithm 1, while Algorithm 2 shows the subprogram to add the edges for a particular write transaction. The corresponding subprogram to handle a read transaction is defined in an analogous manner but not shown for brevity. The function $\beta$ is again constructed by considering every $t \in T$ and mapping it to all the $\tau \in T \setminus \{t\}$ for which the graph $G_\beta$ contains a directed path from $t^{\text{out}}$ to $\tau^{\text{in}}$.

Continuing the example from above, assume that the link $\ell$ is APU-protected, that all features as well as $u_2$ are dependable, while both $u_1$ and $u_3$ are undependable. The corresponding $G_\beta$ for this extended example is shown in Fig. 6. From this graph, we deduce $\beta(t_1) = \beta(t_4) = \{t_2, t_3\}$, $\beta(t_2) = \{t_3\}$, and $\beta(t_3) = \{t_2, t_4\}$. Note that the APU of $\ell$ must allow both $u_1$ and $u_3$ to write to $u_2$. Since both $t_2$ and $t_3$ must listen to write transactions from $\ell$ and $u_2$ is unable to determine the origin of such a transaction, information might in reality also flow from $t_1$ to $t_3$ and from $t_4$ to $t_2$. $\beta$ is able to capture this circumstance accurately. Another aspect that $\beta$ captures is that although an unreliable unit, $u_1$, is connected to $u_2$, on which the feature $t_3$ responds to any read transaction that originates from $\ell$, no information will flow from $t_3$ to $t_1$. This is due to the fact that $\ell$ is APU-protected and there is no specified transaction that requires read access from $u_1$ to $u_2$. Note that in the $\omega_L(\ell) = 0$ case, the algorithm would introduce the link sharing node $\pi_\ell$ to $G_\beta$. This node would then reflect among others the potential information flow from $t_3$ to $t_1$.

As part of our work, we have implemented the formal model using the Eclipse Modeling Framework (EMF) and created a domain-specific language using Xtext that allows developers to describe a model instance in a textual form. The validity of model instances and, in case they are valid, the fulfillment of the information flow requirements from Definition 7 are automatically checked by this implementation.

## VI. Generation Framework

During the creation of a model instance, several requirements towards the capabilities of system entities are formulated: Both features and units can be declared as dependable, but it remains the developer's responsibility to implement them in a way that they satisfy the assumptions that are associated with this dependability. Furthermore, links can be declared as APU-protected. Other than in the previous case, however, it is the goal of the proposed methodology to derive the corresponding APU configurations automatically.

More specifically, an APU of a protected link needs to be configured in such a way that the attached units are able to exchange transactions according to $X_W$ and $X_R$, while all other interactions are prevented. Recall that in order to deliver its access protection, an APU considers only the master unit and the slave unit of a transaction. It has no knowledge of the individual features that are involved in transactions.

As part of this work, we implemented a framework for the platform-specific generation of APU configurations. It is extensible in the sense that new platform-specific generators can be implemented in Java. Using the domain-specific language, every $c \in C$ of a model instance can then be mapped to such a generator. Every link for which an APU protection is desired, i.e., for which $\omega_L(\cdot) = 1$, needs to be directly or indirectly included in exactly one container with such a generator. For every such link in a valid model that fulfills its information flow requirements, the framework derives the respective APU configuration requirements from $X_W$ and $X_R$ and passes them to the corresponding generator. The task of this generator is then to decide whether or not the APUs that it is responsible for are able to fulfill all of these requirements and, if this is the case, generate appropriate configuration code.

In this manner, the APU-specific aspects of the isolation are delegated to pieces of software that are designed to deal with low-level details of a platform. Note that in some cases, a generator will need to be provided with additional inputs. Therefore, the domain-specific language allows developers to specify input parameters that are not considered by the model itself. Instead, they are forwarded to the generator.

As an example, we have implemented a generator for the i.MX 8M from NXP, which is comparable to the ZynqMP. It comprises a so-called Resource Domain Controller (RDC) to protect on-chip components from illegal accesses. The generator expects the MPSoC itself to be modeled as a container and every unit that is part of this container to be either an on-chip master or an on-chip resource that is mapped to a particular memory region. It uses its MPSoC-specific knowledge to generate APU configuration code. More specifically, the generator will output C code that—when being executed on a specific core of the i.MX 8M—will write suitable values to the RDC configuration registers. Note that in order to generate this configuration code, the generator must be aware of the so-called domain that the developer assigns every on-chip master to. This is achieved using the input parameter forwarding described above. Due to the strict separation of these platform-specific aspects, the model itself can be kept as generic as possible and does not have to deal with low-level details such as the domain of an on-chip master.

To validate the overall concept, we applied the implementation to a sample architecture based on the i.MX 8M, were able to generate suitable configuration code, and ensured that the resulting RDC configuration leads to a policy that meets all specified information flow requirements.

## VII. Conclusion and Future Work

As part of this work, we targeted access protection units that are available as part of many commercially available, heterogeneous MPSoCs. Our primary goal was to develop a method for the automatic generation of their configuration data from a platform-independent description of legal and illegal accesses. Therefore, we formulated a model-based design methodology that uses these platform-independent descriptions and associates them with system-level information flow requirements. In this methodology, the specified access descriptions are first verified with respect to the requirements and then forwarded to an extensible framework that handles platform-specific aspects of the configuration by delegating them to suitable generators. As a proof of concept, we implemented the formal model, the information flow analysis, and the generation framework using Java, the Eclipse Modeling Framework, and Xtext.

While our implementation demonstrates basic feasibility of the approach, it is limited to systems that the formal model can capture. An aspect that it cannot currently capture is, e.g., a unit that is dependable in the sense that it strictly isolates the features that it executes from each other and undependable in the sense that it does not protect a feature from failures caused by random faults. To extend the model in such a way that it can represent more complex scenarios is an area for future work. Another limitation of the current concept is that it requires developers to specify exactly one set of transactions and does not allow them to formulate a certain degree of flexibility that platform-specific generators are be able exploit during the search for feasible or optimal configurations. The removal of this limitation is another fruitful area for future work. For certain types of units, it is further conceivable to extend the platform-specific generation process in such a way that it produces unit-internal configuration code (such as for an MPU of a specific processor) in addition to the APU configurations that the current framework generates. This is another possible starting point for further research.

## Acknowledgment

This work was funded by the German Federal Ministry of Education and Research (BMBF) under grant number 16KIS0886 (DEFEnD). The responsibility for the content of this publication lies with the authors.

## References

[1] M. Hassan, "Heterogeneous MPSoCs for Mixed-Criticality Systems: Challenges and Opportunities," *IEEE Design & Test*, vol. 35, no. 4, pp. 47–55, 2018.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[3] S. Saidi, S. Steinhorst, A. Hamann, D. Ziegenbein, and M. Wolf, "Special Session: Future Automotive Systems Design: Research Challenges and Opportunities," in *2018 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Turin, 2018, pp. 1–7.

[4] R. Ernst, "Automated Driving: The Cyber-Physical Perspective," *Computer*, vol. 51, no. 9, pp. 76–79, 2018.

[5] SAE International, "J3061: Cybersecurity Guidebook for Cyber-Physical Vehicle Systems," Standard, 2016.

[6] V. Boppana, S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, "UltraScale+ MPSoC and FPGA families," in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Cupertino, California, 2015, pp. 1–37.

[7] S. McNeil, P. Schillinger, A. Kolarkar, E. Puillet, and U. Gertheinrich, "Isolation Methods in Zynq UltraScale+ MPSoCs," 2019, Xilinx Application Note, XAPP1320.

[8] D. Kliem and S.-O. Voigt, "Scalability evaluation of an FPGA-based multi-core architecture with hardware-enforced domain partitioning," *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 845–859, 2014.

[9] L. Lopriore, "Memory protection in embedded systems," *Journal of Systems Architecture*, vol. 63, pp. 61–69, 2016.

[10] T. Nojiri, Y. Kondo, N. Irie, M. Ito, H. Sasaki, and H. Maejima, "Domain Partitioning Technology for Embedded Multicore Processors," *IEEE Micro*, vol. 29, no. 6, pp. 7–17, 2009.

[11] B. Tan, M. Biglari-Abhari, and Z. Salcic, "Towards decentralized system-level security for MPSoC-based embedded applications," *Journal of Systems Architecture*, vol. 80, pp. 41–55, 2017.

[12] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*, Madrid, 2014, pp. 109–118.

[13] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical Fundamentals of Gate Level Information Flow Tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1128–1140, 2011.

[14] A. Sabelfeld and A. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[15] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," *SIGPLAN Not.*, vol. 39, no. 11, pp. 85–96, Oct. 2004.

[16] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Proceedings of the 48th Design Automation Conference*, San Diego, California, 2011, p. 254.

[17] SAE International, "AS5506C: Architecture Analysis & Design Language (AADL)," Standard, 2017.

[18] R. Pellizzoni, P. Meredith, M.-Y. Nam, M. Sun, M. Caccamo, and L. Sha, "Handling mixed-criticality in SoC-based real-time embedded systems," in *Proceedings of the seventh ACM international conference on Embedded software*, Grenoble, 2009, pp. 235–244.

[19] M. Maurer, J. C. Gerdes, B. Lenz, and H. Winner, Eds., *Autonomous Driving: Technical, Legal and Social Aspects*. Springer, 2016.