EasyChair Preprint

№ 686

Making Configurable and Unified Platform, Ready for Broader Future Devices

Myungjoo Ham and Geunsik Lim

December 19, 2018

Making Configurable and Unified Platform, Ready for Broader Future Devices

MyungJoo Ham, Geunsik Lim
Samsung Research, Samsung Electronics
Seoul, Republic of Korea
{myungjoo.ham, geunsik.lim}@samsung.com

***Abstract*—The wide spread of IoT and edge devices has introduced new challenges for software platforms of consumer electronics, of which traditional targets had been smart phones, wearable devices, and smart TVs. In general, such traditional devices share well-defined common features and requirements, which emerging IoT devices lack of. Besides, IoT and edge devices have much longer tails, which makes it further intractable to define common features and requirements, composing a customized software platform. Such high diversities prohibits having individual build configurations per device type, which multiplies burdens for developers as well as infrastructures. Besides, IoT developers need to create customized platforms for new devices on-the-fly, which traditional platform tools are not capable of; such tasks usually require the rare release and build experts along with significant time and effort. In this paper, We have successfully addressed the issues: unifying software platform (Tizen) and its infrastructure to increase the developmental productivity for varying device types and making Tizen highly configurable so that even third party developers may create their own variations on-the-fly easily and quickly. This has enabled a public web service, Craftroom.tizen.org, where IoT developers may acquire their own customized Tizen on-the-fly. This project is integrated into Tizen since 4.0, which is released to the public, and has enabled Tizen team to start IoT platforms and a small team to prepare software platforms for autonomous driving systems and on-device AI systems with minimal time and effort.**

***Index Terms*—software engineering, build system, release and deployment, software platform**

I. Introduction

Traditionally, software platforms for consumer electronics devices (e.g., Tizen, Android, and iOS) have clear target device profiles: phones, wearables, tablets, or TVs. Such profiles define mostly (if not completely) common API sets, features, and requirements across different hardware sets. However, the wide spread of IoT and edge devices has introduced the new challenge: the long tail of the IoT [1], which incurs the following significant challenges for software platforms:

- Each IoT device and application might require their own customized software platform: *configurability*.
- Varying software platforms are to be built from the same source codes with different configurations. It may require extremely high build workloads: *build explosion*.

Both challenges have already induced significant developmental costs with traditional smart devices. *Build explosion* enforces developers and infrastructure to build and test software packages repeatedly for each profile. Besides, *build explosion* attributes to the lack of *configurability* by incurring high developmental costs to platform developers, which is further deteriorated by the long tail of the IoT. Even if we have only 10 different IoT devices, not 100, developers need to build 10 times for each commit and the build infrastructure gets additional x 10 workloads. This is especially unacceptable with the long tail-ness; each IoT device type is going to have little revenue although the whole IoT device portfolio may have huge revenue. We cannot afford linearly increasing platform development costs with the increasing number of IoT device types. Tizen platform developers have been already overloaded with repeated build and test workloads per profile and suffering from the lengthy build task queues of the overloaded build infrastructure. Thousands of build tasks have been waiting several hours in task queues during busy hours despite of a lot of powerful build servers.

Configurability is deteriorated not only by *build explosion*, but also by the difficulties in composing a customized software platform with software packages for a specific product. There are well-established package management systems (e.g., Advanced Package Tool (APT) for DEB packages [2] and Zypper/DNF for RPM packages [3]), which allow to choose software packages without understanding the dependencies of packages. However, developers still need to specify packages required for their software platform among the thousands of candidates, which require deep understandings on the software platform from device drivers and system software to applications. Because such developers are rare, if we need to prototype a lot of varying devices and their more varying tailored software platforms, the traditional well-established tools cannot support the required configurability. Thus, we need new tools that allow novice developers (or developers without full-ranges of knowledge across software platforms) to compose properly tailored software platform.

We have addressed *build explosion* first in order to enable *configurability* because we should be able to build software packages for various devices with affordable workloads. We address *build explosion* by unifying the Tizen build procedures and binary repositories, enforcing every profile to share the same set of binary packages, not only the source codes: Tizen:Unified project. In order to unify Tizen, we have introduced new rules for platform developers and plugin architectures with new dependency rules, enforced by build systems and Linux packaging systems, which are upstreamed to the open source communities as well so that the new rules

comply with the standard. Thanks to the cooperation from platform developers and the communities, we could have fully unified Tizen projects and repositories since Tizen 4.0 in 2017.

Unifying binary repositories has mitigated the build workload not only for new IoT devices, but also for the traditional smart devices. Tizen project (Tizen:Unified in https://build.tizen.org) no longer builds five times for five profiles, but builds only once for all device types, which is expected to reduce the workload of infrastructure roughly to one fifth. After the completion of Tizen:Unified, peak build task waiting queue size has dropped from thousands to tens [4].

We have applied new infrastructure designed to promote *configurability* along with Building Blocks of Tizen after the completion of Tizen:Unified. Building Block definitions [5], [6] provide easy-to-configure components to build a custom software platform, which are maintained by product managers. Building Blocks are meta packages with hierarchical structures that provide easy-to-understand descriptions. A Building Block does not have any file contents but have relational information of software packages and other Building Blocks; thus, it is a meta package. Tizen Image Creator (TIC) [4] offers a web user interface to configure a custom software platform with Building Blocks and individual packages. Tizen offers an instant software platform creation service for novice IoT developers, Craftroom [7], a simplified and beautified variation of TIC. TIC and Craftroom provide customized software platforms for IoT developers within minutes, not hours or days.

We have designed and implemented both projects, Tizen:Unified and Tizen:Configurability, refactored software packages, restructured build procedures and rules, which enabled on-the-fly platform customization. Build and packaging rules introduced in this work are either within the Linux standard (RedHat/OpenSUSE) or upstreamed to the community. We have reduced build and test workloads significantly for conventional devices and enabled Tizen for wider ranges of device types with significantly reduced workloads for both the developers and the infrastructure. This work is applied to Tizen 4.0 (2017) and 5.0 (2018), enabling various new products and prototypes, ranging from home appliances and IoT prototypes to autonomous driving systems and on-device AI platforms.

II. Related Work

A. Linux package management

The modern software platforms have extremely large number of packages with complex inter-package dependencies, which helps avoid duplicated source code and functions with shared libraries and daemon services complicating dependencies, which requires dividing software into smaller packages. As a result, modern Linux package management system is required to manage complex inter-package dependencies consistently and efficiently.

Mancinelli et al. [8] have shown a method to handle dependencies between packages of a software platform, which have previously been only addressed by manual labor. Most of the prior approaches have focused on declaring forward dependencies (i.e., `Requires` or `BuildRequires` in RPM systems) and following them. The method of [8] tries to automatically discover undeclared dependencies by inspecting build scripts so that we may reduce manual labor. Caixa Magica and Mandriva Linux distributions use this method.

Software Build Systems [9] describes requirements for future software build systems including the choices, benefits, and challenges of a well-designed build process. It surveys the tools and techniques for building software products and how things may go wrong. They conjecture that inadequate build systems can dramatically deteriorate the productivity; a subpar build system may incur bad dependencies, false compile errors, failed software images, slow compilation, and excessive manual processes. It explains how to optimize the performance and scalability of a given build system.

Galindo et al. [10] address the issues such as the lack of realistic variability models to evaluate dependency identification techniques, which is recognized as a major problem by the community. They suggest that Debian packages dependency language can be considered as a variability language. Also, they provide a mapping from this language to propositional formulas enabling their analysis by means of SAT solvers. They focus on other variability dependency language existing in open source community increasing even more the availability of realistic variability models up to 20,000 packages to the Software Product Line (SPL) community. However, they do not detect anomalies in Debian models such as conditionally dead packages or redundancies.

Cosmo et al. [11] point out why the upgrade problems faced by Free and Open Source Software (FOSS) distributions have characteristics not easily found elsewhere. They provide periodic snapshots of a whole software platform, which can mitigate OS upgrade problems along with disk partitioning. They survey current countermeasures to such upgrade failures, argue that they are not satisfactory, and sketch alternative solutions. They focus only on applying fingerprinting techniques to cluster maintainer scripts of Debian distribution to get a clear view of all their use cases.

Adoption of Academic Tools [12] presents an overview of 10 years of research in this field and the process leading to the adoption of our tools in a FOSS community. They presented the check tools such as *distcheck* and *buildcheck*, which scan all the packages in a Debian distribution to identify installability issues. They focused on the Debian distribution and in particular they looked at the issues arising during the distribution lifecycle: ensuring buildability of source packages, detecting packages that cannot be installed and bootstrapping the distribution on a new architecture.

Schroeder [13] depicts the package dependency solver library that is called SAT solver. This project has been started in May 2007 when the ZYpp community has decided to use a database to speed up installation. It offers an efficient file and memory representation for complex dependency relations and package repositories. The SAT solver of Libzypper is a port from the red carpet solver, which is to update packages of a running system. In addition to SAT solver, they provide adhoc mechanisms to discover some of undeclared dependencies

and an audit function for weak dependencies.

Zypper [14] is a package manager that installs, updates, and removes packages and manages repositories. It is especially useful for managing software packages remotely or with shell scripts. With Zypper, we can easily update the distribution. Besides, we can update the software platform in run-time. Note that Tizen is fully compatible with Zypper although it is not included in most traditional Tizen profiles.

B. Build systems of Tizen

We describe build and release systems of Tizen, a software platform for IoT/edge devices, general consumer electronics, mobile phones, wearable devices, or even autonomous driving vehicles. The standard packaging for Tizen is RPM [3], which is also standard for OpenSUSE, RedHat, and Fedora. We do not discuss Debian packages [2] although it is one of the two major standards along with RPM in Linux communities. Note that the expression power of inter-package dependencies of Debian is not higher than that of RPM; Debian package dependencies may be expressed with RPM dependencies.

Git Build System (GBS) [15] is a build and packaging tool for Tizen platform development. It generates tarballs, builds sources, and packages binaries from Git repositories. GBS also does local unit tests, provides build and test sandboxes, submits code to the build infrastructure, OBS. Package maintainers may use GBS to maintain their upstream branches or forks, or to prototype packages not included in Tizen mainline.

MIC Image Creator (MIC, originated from Meego Image Creator) [16] builds software platform images for Tizen. MIC creates images of different types, including live CD images, live USB images, raw images for KVM [17], loop images for IVI platforms, and filesystem images for chrooting. Users can enter into the generated images with MIC. MIC changes the apparent root directory with a *chroot* mechanism for the current running processes. Note that MIC uses Zypper [14] to resolve package dependencies during image creation.

Open Build Service (OBS) [18] is a general build, release, and distribution system for various target platforms in an automatic, consistent, distributed, and reproducible way. OBS releases software for a wide range of operating systems and hardware architectures with extensible web interfaces and APIs. It is an open and complete distribution development platform that provides a transparent infrastructure for the development of Linux distributions, used by openSUSE and Tizen. OBS supports Fedora, Debian, Ubuntu, RedHat, and many distributions. Like GBS, OBS builds binaries in a sandbox to ensure the consistency. OBS may create binary packages in varying formats including RPM [19], DEB, and many others. The created packages can be released and deployed via repositories compatible with package managers.

III. Issues of Tizen before 4.0

Tizen before 4.0 has a build project for each profile: i.e., Mobile, Wearable, TV, IVI, and Common, which has its own binary repository. Such binary repositories are not disjoint; there are packages with duplicated names across profiles. Thus, we cannot compose a software platform with packages of different profiles because the inter-package dependency chain of a profile is not compatible with that of others. For example, with packages `A`, `B`, and `C`, present in all profiles, where `A` and `B` depends on `C`, we cannot install `A` from Mobile and `B` from TV because we cannot install two instances of `C` from both profiles. Note that Common profile is simply yet another profile and does not represent common part of others.

Creating a new device type, which happens very frequently with IoT and edge devices, has required to define a new profile and its dedicated build project. This is not a unique issue of Tizen; Android, dominant for mobile phones, and Yocto [20], popular for IoT devices, also require an independent build project for each device prototype. For each build project, the infrastructure is required to build the whole packages and developers are required to build and test the packages. This incurs redundant workloads proportional to the number of device types. Even if the same source codes are shared across profiles, the whole packages are built repeatedly, which is already too expensive with only five profiles.

Frequently prototyping new device types significantly multiplies such workloads, making it intractable. Administrators do not allow creating new profiles in the already overloaded infrastructure, where developers and managers keep complaining about the latency. Besides, creating a new profile has been too difficult for non-build-expert developers. It requires to fully understand low-level system software packages and the underlying mechanisms of build systems.

IV. Design of Tizen:Unified

We have achieved the unification in April, 2017. We have removed per-profile build projects and provided a single build project and binary repository for all. The objectives of the unification includes: a) shorter build latency by eliminating duplicated builds; thus, making affordable to support more profiles, b) allowing to create arbitrary profiles from a single shared binary repository, and as a results of a) and b), c) allowing to create new profiles on-the-fly without rebuilding packages, enabling the next big step, configurable Tizen.

There are two major rules to achieve the unification:

- In build-time, build processes (build scripts, compilers, and build systems including OBS [18] and GBS [15]) should be agnostic to the profile except for the architecture: e.g., *armv7l* and *x86_64*. However, processes may identify the profile at install-time, boot-time, or run-time.
- Every single binary package (i.e., RPM for Tizen) should be able to co-exist in a single repository with other packages as long as the packages have the same architecture.

With the completion of the unification, these rules have been applied for all Tizen packages. The build system is configured to enforce the rules automatically so that any violating changes cannot affect the system since Tizen 4.0 [4].

For the unification, the huge number of software packages itself has been a significant challenge; e.g., Tizen 3.0 Common profile has 822 source packages [4]. When we have started planning Tizen:Unified in July, 2016, we have discovered

134 source packages disobeying the rules. When we have started refactoring with a small team of four developers in November, 2016, the number of such packages has increased to 171. Another challenge is the regression, where a complying package becomes not complying. We have observed multiple cases of such regressions, both regressions of the refactored packages and natively complying packages.

Ideally, refactoring is better executed by main contributors of the package. However, it requires the main contributors to fully understand the concept of Tizen:Unified, the mechanisms of build systems under their own workloads and tight schedules. During the early phase of the project, as a pilot program, we have tried the ideal method with few small groups. As the result, we have learned the following points:

- We should not expect developers to fully understand the build system and inter-package dependencies. They are users of the build system and a good build system should not require users to understand the system itself.
- It is extremely difficult to ensure that the rules are kept during active development where new commits are applied daily if not hourly.
- Applying yet another coding rule requires additional burden to developers. Even if the need is justified to team leaders and main contributors, it is often not enough for them to put additional efforts to prevent others from breaking the rules or to review more carefully. Moreover, we cannot enforce the rules with the infrastructure until every package follows the rule perfectly because any disobeying package will break the build.

As a conclusion, we have decided to do all the refactoring with Tizen:Unified members except for a few packages (Chromium, EFL, and input systems) with much complicated issues that require to rely on their own main contributors. During the progress of Tizen:Unified project, to detect any attempts of regression, we have developed a gerrit [21] monitoring service that reviews incoming commits and finds any possible regressions.

We categorize packages to refactor into the following types:

- **Type-A.** The build script is aware of profiles. It may use different source files or code blocks statically (usually with `#ifdef` or `#if`) per the profile.
- **Type-B.** Multiple git repositories generate packages with a common name. For example, both `mobile/efl-config.git` and `wearable/efl-config.git` had generated efl-config.
- **Type-C.** At build-time, the package depends on packages that generates different build environments per profile.

We have observed a lot of useless dependencies on the profiles or device types and removed them immediately. We also have had cases where the dependencies are easily removed by using configuration files in `/etc`.

A. Runtime profile identification

If we cannot avoid per-profile behaviors, applying the runtime profile identification is recommended. This is recommended because it allows using the same binary across different profiles avoiding installing different binaries per profile. The only more recommendable method is to behave exactly same regardless of the profile definition itself, which is often impossible. Many Type-A cases are resolved by this.

```
#ifdef PROFILE_MOBILE          ||  if (get_profile() == MOBILE)
    Do_mobile_action();        ||      Do_mobile_action();
#else                          ||  else
    Do_common_action();        ||      Do_common_action();
#endif                         ||
          (a)                  ||            (b)
```

Code I. (a) Type-A code with `#ifdef` and (b) refactored code

In order to allow runtime identification, we have used a Tizen API to detect the profile and removed all preprocessor conditionals related. Code I shows a simple example where such refactoring is applicable.

B. Inter-package dependency management

Sometimes, we have created different binary packages for each profile or device types along with meta packages or virtual packages (RPM capabilities) to make per-profile differences transparent to other packages.

Let us assume that we have a package *X*, which needs to have different binaries for mobile profile. Then, we can write the build scripts to create a subpackage, *X-profile_mobile.rpm*. The first variation of the mechanism is to create *X.rpm* file for other profiles and to create *X-profile_mobile.rpm* that may act as *X.rpm* by adding the reverse-dependency, *Provides: X*. With the first variation, *X.rpm* needs to declare that it `Provides` virtual subpackages such as *X-profile_common* in order to be explicit for per-profile configurations. The second variation is to create a meta package, *X.rpm* without any files, but with meta data stating its dependencies on a virtual subpackage, *X-compat*, and to create subpackages of profiles that do `Provides: X-compat`, which enforces to install one of such subpackages to install *X.rpm*. Both variations require subpackages to declare *Conflict* statements to prevent installing plugins of different profiles for one main package.

A critical side effect of these mechanisms is that build systems (both OBS [18] and GBS [4]) are confused by multiple candidates (the subpackages) for inter-package dependency resolutions. In other words, if *X.rpm* is required by *Y.rpm* while we build *Y.rpm*, the build systems cannot determine whether it should install *X-profile_mobile.rpm* or the other; they do not accept such ambiguity. We have resolved such ambiguity with weak dependencies, `Recommends`, to declare default selection for build systems, which the corresponding open source community (BSSolve and OBS) have accepted. We have chosen the weak dependency, `Recommends`, because we can override it with strong dependency, `Requires`, for other plugins or profile supports while we may provide hints for build systems. There is a weaker dependency in RPM standards, `Suggests`; however, we have not used it for this purpose because only human is supposed to use `Suggests`; thus, we use `Suggests` for constructing package list GUI only. Note that having the default package with `Recommends` at built

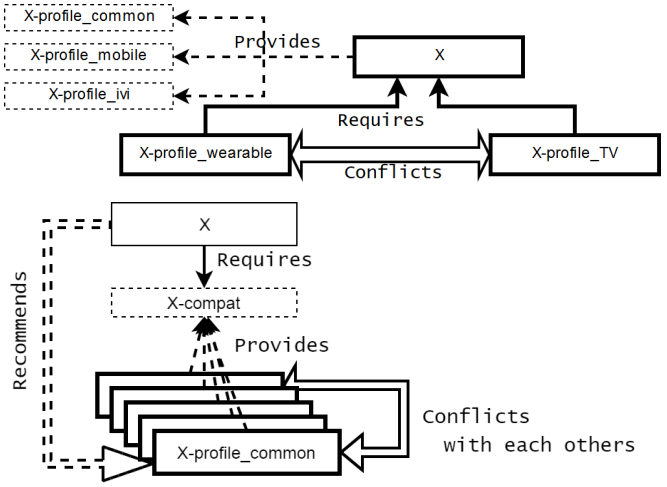

Fig. 1: The two subpackaging examples supporting different binaries per profile transparently to other packages.

time does not affect software platforms with different plugin packages because they are shared objects; only their forms (header files) matter, which holds for Tizen native platform binaries. However, we had exceptions to this, mentioned in the next section, where the contents of external headers (APIs) differ per profile.

Fig. 1 elaborates inter-subpackage dependency management mechanism; the top shows the first variation and the bottom shows the second variation. Note that any packages external to *X* should not explicitly depend on profile plugin subpackages (**-profile_*.rpm*). Such packages should refer to the main package (*X* in Fig. 1) only and let the package management system handle the rest.

Type-B and Type-C are mostly resolved by declaring inter-package dependencies explicitly after properly refactoring the build scripts and source codes so that per-profile parts are separated into different binary packages. There are many Type-A cases resolved by this mechanism as well when it is too difficult to apply the mechanism of the Section IV-A. It is usually when a profile has completely different source code files, which incurs too large modifications to apply runtime profile identifications. In such cases of Type-A, we generated different executables for each profile and packaged them into different binary packages, which is often referred as sub packages. However, for the package management mechanism, there are no differences between packages and sub packages.

C. Inappropriate definitions of APIs

Unfortunately, a few developers have written external header files with different function declarations per profile or even provided different header files for different profiles. There has been even a case where Tizen public API header has different C enum definitions per profile. In this case, the C enum definitions of mobile have been `PLAYER_DISPLAY_TYPE_EVAS = 1` and `PLAYER_DISPLAY_TYPE_NONE = 2` while those of wearable have been `PLAYER_DISPLAY_TYPE_EVAS = 2` and `PLAYER_DISPLAY_TYPE_NONE = 1`. This extreme case has required Tizen public API changes that may make previous applications incompatible with new Tizen versions. Thus, we have redefined C enum values with unused and unified values and marked old values (1 and 2) as deprecated along with compatibility resolving code that behaves differently for each profile detected in run-time.

For the first case, where profiles have had different function declarations using compiler preprocessor conditionals, we have manually refactored header files removing all preprocessor conditionals. We cannot apply the mechanism of Section IV-B for headers exposed externally because the depending package (`X.rpm` in the examples of Section IV-B) cannot be identical for different profiles; profile dependency is no more transparent to external packages! For the other cases, where different files have been used per profile, we have manually inspected each declaration and merged them into a superset containing all declarations from all profiles. Then, we have added dummy function definitions for profiles that do not or cannot execute it, determined either at run-time with profile probing APIs or at install-time with different binaries per profile.

D. Workarounds

There are a few cases where we cannot completely remove the dependencies on profiles or device types from the binary packages except for the plugin subpackages, which do not incur external dependencies.

- Case 1: Device drivers or kernel binaries require dependencies on device types and often, such dependencies are hardcoded.
- Case 2: We are not allowed to generalize product or business division specific routines. A business division has required keeping their product specific conditional codes embedded in Tizen while such codes cannot be generally used for other profiles or devices. Although, in principle, this is undesirable as it fragments the source code and damages the readability, this has been something we could not alter.

Case 1 includes Linux kernel, which is built and installed for all devices, but their source code repositories or build configuration might differ. We have applied an altered method of Section IV-B as shown in Fig. 2.

As shown in Fig. 2, different Linux kernel binaries from various git source repositories provide the capability of Linux-kernel so that any package or software platform manifests requiring Linux-kernel can be satisfied with one of Linux-kernel binary packages. However, as mentioned before, we cannot have multiple candidates for a single dependency (or capability in RPM documents) in build time. Thus, for external kernel module repositories, a representative kernel repository has been chosen to provide the Linux-kernel-modulebuild capability, which is the emulator kernel repository. Note that we can have a single representative for each Linux kernel version. Therefore, if there are multiple Linux kernel versions required for a specific Tizen

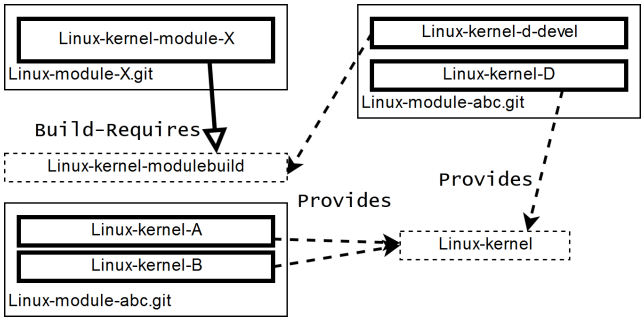

Fig. 2: The altered method of Fig. 1 for Linux kernel

version, we can choose multiple representatives, one for each kernel version. We may choose multiple representatives without introducing the ambiguity for build systems by specifying kernel versions in kernel modulebuild dependencies. That is `BuildRequires: linux-kernel-modulebuild = 4.4.0` and `Provides: linux-kernel-modulebuild = 4.4.0` instead of `BuildRequires: linux-kernel-moduilebuild` and `Provides: linux-kernel-modulebuild`.

For case 2, we have configured the Tizen main public build systems to ignore any macros related with products or business divisions. Although we have allowed preprocessor macro conditionals for products and business divisions, as long as the main build system ignores such macros, we can guarantee that the resulting binaries are complying Tizen:Unified. Simultaneously, with the same source code repositories, a business division may keep their own special source code activated in their own build projects in their own private build systems. However, codes creating different binaries under the same package names should be discouraged. Such codes are usually the results of laziness (or overloaded workloads) of developers; they should have used configuration files (.ini files in /etc directory) or accessed Tizen device APIs, probing device type, name, and profiles. We hope, in some day, such lame codes are totally eliminated.

E. How we have progressed

Regressions by incoming commits breaking the rules mentioned in the first paragraphs of Section IV have been headaches for the Tizen:Unified project. Before April, 2017, during the active development of Tizen:Unified project, the build system could not be configured to ignore per-profile build configurations. Thus, we have constantly monitored incoming Tizen commits with a monitoring service implemented to find any regressions in the rule enforcement. For the reported regressions, we have intervened the related activities, where we could have successfully dropped or corrected violating commits before being merged.

Once every single package in Tizen has obeyed the given rules of Tizen:Unified, we have removed all per-profile build projects from the infrastructure and unified all Tizen packages into a single build project named `Tizen:Unified`, which does not allow any dependencies on profile or device types in built-time. Note that build projects for older versions (e.g., Tizen 2.x or 3.0) are untouched and kept with per-profile basis; however, they are no more actively developed and do not require heavy build workloads as work-in-progress versions. Another build project, `Tizen:Base` exists independently in order to reduce the performance impact from cyclic build dependencies of toolchains. Cyclic build dependencies incur heavy redundant build workloads and are prohibited in Tizen:Unified because it is expected to be rebuilt frequently; most developers contribute to Tizen:Unified daily. Note that as long as multiple build projects are disjoint (not sharing source repositories) and inter-project dependency is acyclic, there is no build performance problem and on-the-fly configurability. The traditional per-profile build projects are not disjoint, sharing hundreds of source repositories with each other.

Once Tizen:Unified is completed, we have observed another unexpected advantage that we could have exploited for other projects. We can add a disjoint build project that does not have cyclic inter-project dependencies without any deterioration on the performance or the configurability. With this characteristics, we could have created aggressive prototypes quickly without affecting conventional Tizen projects. An autonomous driving project and an on-device AI platform project require a lot of additional source repositories, well over 100. Besides, although these projects are based on Tizen, they are not officially Tizen projects and the related packages could have not be included in official Tizen repositories. By adding a disjoint build project, depending on Tizen:Unified, but outside of Tizen official build infrastructure, Tizen:TAOS (representing Tizen AI OS Support, Tizen Autonomous-driving OS, and Tensor-Aware OS simultaneously), we could have been aggressively prototyping new software platforms. When there is a new need for yet another variation, a single developer can generate and deploy a new software platform and its binary images to corresponding developers and devices within few hours, which is frequently occurring. The inter-project dependencies are shown in Fig. 3. In Fig. 3, shaded blocks represent Building Blocks and white blocks represent individual software packages. Blocks with thick outlines represent mandatory items–subblocks or individual packages of a block–, which means the items are chosen unconditionally if their parent is chosen. For example, if a parent, `Graphics`, is chosen, its mandatory item (child), `2D`, is always chosen. On the other hand, optional item, `3D`, is may be omitted even if `Graphics` is chosen. Note that unless explicitly declared, a child may be chosen without choosing its parent. For example, `3D` or `2D` may be chosen without choosing `Graphics`. If the whole platform has not been unified as in pre-4.0 Tizen, this would have been challenging task because we cannot pick a few packages from a profile and some other packages from another profile simultaneously for a single new project. Note that this is different from the configurability because the configurability is to configure a system within the official Tizen project.

V. Design of Tizen:Configurability

With the completion of Tizen:Unified project, Tizen has achieved the basic ability to configure a software platform on-

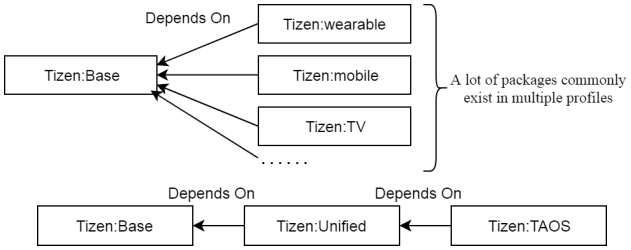

Fig. 3: Inter-project dependencies of Tizen build projects. The top shows pre-4.0 and the bottom shows 4.0 or later.

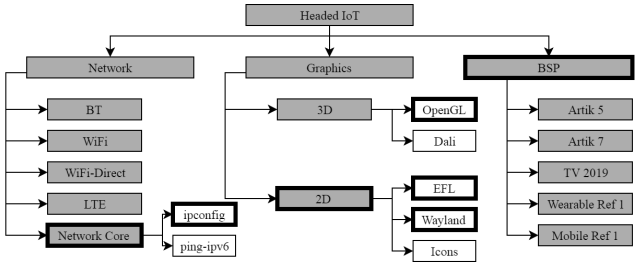

Fig. 4: Example of building block hierarchy

the-fly. Because every single binary package can be located in a single package repository and there is no more ambiguities between package names any more, we can now use package management systems without glitches anymore: zypper, DNF, or yum, which are equivalent to apt in Ubuntu/Debian systems.

For most workstations, PCs, or servers, this degree of configurability may be enough. However, it is not enough for embedded systems and IoT/edge systems consisting of both devices and software platforms. For such systems, we need to continuously release and deploy OS images that can be flashed to devices easily without manual labor. Traditionally, in Tizen, we have been using a tool, MIC, previously named MeeGo Image Creator. Issues with traditional tools include:

- Issue 1: It is too difficult to create a new software platform configuration (.ks file) for most users; you need build system experts who understand Tizen and its infrastructure deeply. Thus, it can be practically impossible for most third party developers to create Tizen prototypes.
- Issue 2: You need your own dedicated Linux workstation to create Tizen images. It would be more appropriate if IoT application developers with an access to web browsers and Tizen Studio [22] can write their own IoT applications and generate proper Tizen OS images for their own IoT devices and applications on the fly.

In order to address Issue 1, we have introduced the concept of building blocks and implemented the first draft in https://git.tizen.org/cgit/tools/building-blocks/ in May, 2017. The concept of building blocks is now the core of Tizen profiles and device type definitions. It is also the main tool for project managers. With building blocks, users can create prototypes without the knowledge of thousands of individual Tizen packages or to choose hundreds from them, but with the fewer (about one or two dozens) abstract and easy-to-understand building block names: e.g., Bluetooth, Haptic, and Default-App-Setting.

In order to address Issue 2, we have introduced Tizen Image Creator (TIC), which has the web frontend with node.js. Analyzing individual packages and generating deployable OS images for the configured prototype is executed in a web server. Tizen team has opened web service that uses TIC and building blocks as its backend at https://craftroom.tizen.org: Craftroom. Craftroom service offers simplified and easier interfaces focused on IoT developers.

A. Building blocks

A building block is a meta packages that designates mandatory packages and optional packages. As a meta package, it does not have its own files, but has dependency relation information consisting the list of mandatory or optional packages. A mandatory package is an individual Tizen package or another build block that is installed if the corresponding building block is chosen. An optional package is an individual Tizen package or another building block to be shown in the user interface if the corresponding building block is chosen. Unlike mandatory packages, optional packages are not automatically chosen with the corresponding building block.

Building blocks have the hierarchy between the blocks that can be expressed as a tree with a root node and a leaf node is an individual Tizen package of a empty building block. Fig. 4 shows an illustrative example of building blocks as a tree. However, please note that the actual definitions of build blocks are far more complex with a lot of building blocks and individual packages.

In Fig. 4, boxes filled with gray represent building blocks. Boxes without gray filling represent individual Tizen packages. Although most of building blocks are supposed to contain individual Tizen packages, we omitted to simplify the figure. Boxes with solid thick outlines represent mandatory packages. Boxes with dashed outlines represent optional packages. The root node, `Headed IoT` does not belong to any other building blocks; thus, it is neither mandatory nor optional.

Note that non-hierarchical dependency relations between building blocks are not shown in the figure; e.g., `TV 2019 Requires 3D`. The hierarchy of building blocks is defined in order to visualize building blocks for users. The left side of Fig. 5, shows how the hierarchy generates building blocks lists for users with TIC, which is to be elaborated in the next section.

In order to make the definitions of building blocks highly readable to project managers and developers and to make the resulting relations consistent, there are a list rules in writing building blocks along with a rule checker that breaks generating building block packages if there is a rule break. The rule is enforced by a rule checker that is implemented to generate build breaks if there is any violations. Because the list of rules is too lengthy to be described in this paper, please refer to [23] for the full list.

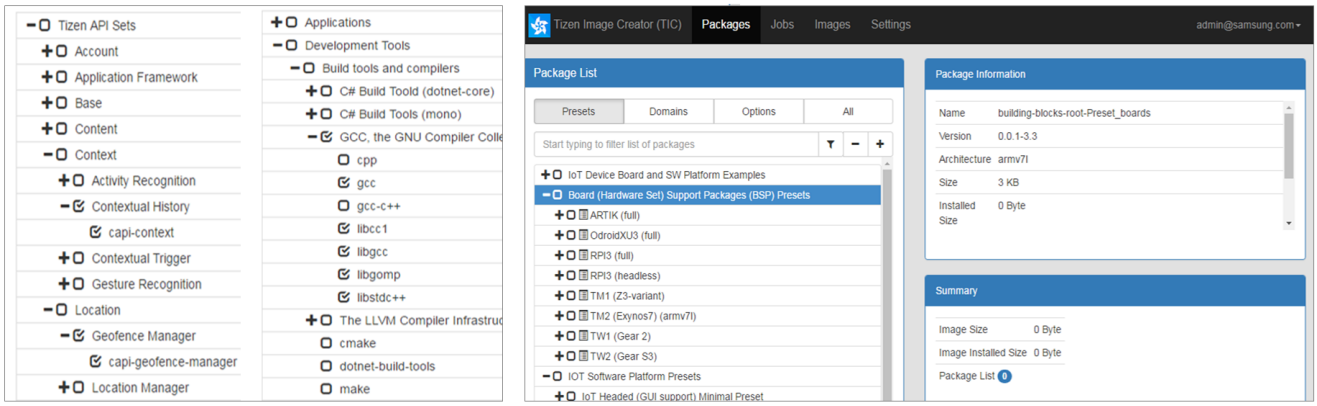

Fig. 5: Building Block hierarchy of May 2017 draft (left) and TIC screenshot where users may choose presets (right)

B. Tizen Image Creator (TIC)

Tizen experts can prototype software platforms easily and quickly with building blocks. However, third party developers or casual developers require approaches more user-friendly than writing the KickStarter [24] `.ks` file. With web application experts of another team, we have introduced TIC, which visualizes building blocks and individual packages with their relations and capabilities to configure in detail and in abstract simultaneously.

The right side of Fig. 5 shows the screenshot of TIC on a web browser. There are three different categories of building blocks: `Presets`, `Domains`, and `Options`. With `Presets`, users can select predefined sets of building blocks and individual packages before actually choosing blocks or packages in Domains or Options. For example, if a user wants to create a prototype based on Tizen-TV, he/she may choose TV preset and proceed. On the top-right panel, users can review the currently selected package or block. On the bottom-right panel, users can preview the currently configured software platform.

With `Domains`, users may choose building blocks listed based on the Tizen API sets. With `Options`, we provide additional blocks that may help prototyping although they are not supposed to be used for actual commercial products: e.g., debugging tools, text editors, and profiling tools. Recently, with development of Tizen 5.0, such building blocks and packages are suggested to be moved to Tizen:Tools however, related tasks are still in progress.

When users have completed configuring the software platform, they may order TIC to create the corresponding OS image file in the server so that they can download the file for deployment a few minutes later. Users may define additional binary package repositories so that they may add their own custom binaries. Please refer to the presentations and demonstrations of TDC 2017 for more details of TIC [4].

C. Craftroom

Based on TIC, the Tizen team has opened a public web service, Craftroom [7], which provides much simplified services of TIC. With Craftroom, users may generate Tizen IoT software platforms based on the hardware specifications and their own IoT applications developed with Tizen Studio [22] within minutes.

D. Side effects on commercialization

There has been a major concern from release engineers. In a commercialization build project of a business division, they build the corresponding packages only. They do not include packages not intended for their profiles in their own build system, which is a fork of Tizen public. Thus, if we release Tizen as in the form of Tizen:Unified, they may suffer from longer build latencies due to the inclusion of packages not required by their profiles.

In order to resolve this issue, we have implemented an inter-package dependency analyzer to generate the optimal list of source repositories for a specific software platform configuration. In order to have a minimal fork for a specific commercialization project, the release engineers may use the tool to minimally choose repositories to be forked.

VI. Evaluation

Both Tizen:Unified and Tizen:Configurability are applied and integrated to Tizen 4.0. We can no longer worry about regressions because any violations would not affect the system because the infrastructure is configured to be agnostic to profiles. The currently developed version, 5.0, has also adopted the proposed work in this paper and is working as expected and releasing and deploying binaries for various device types with the configurability intact. Note that Craftroom [7] provides pretty and easy-to-use UI; however, it is not exposing the full capabilities of configuring a platform. Craftroom simply provides customized software platform based on the user application provided.

Fig. 6 shows how the overall Tizen build, release, and deployment infrastructures before and after this work; the top shows the infrastructure before this work and the bottom shows that after this work. Before this work, adding another device type has required to multiply the build workload while it does not after this work. In other words, adding another device type now only requires to define a new recipe with the given building blocks, which does not increase the workload visibly

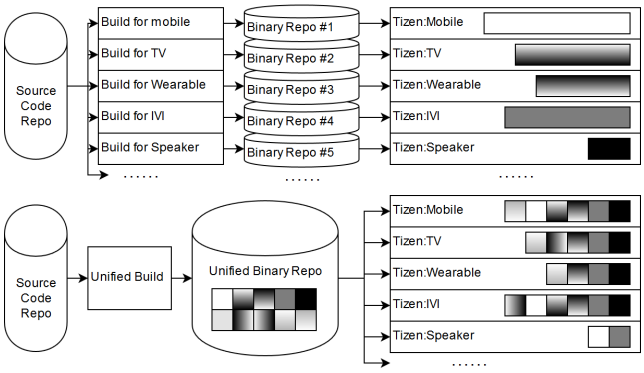

Fig. 6: The infrastructure before and after the work.

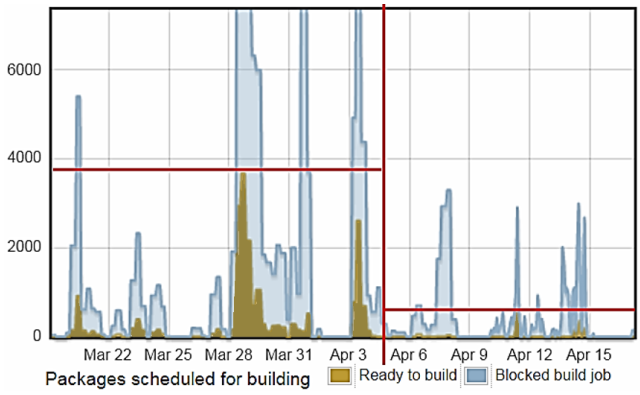

Fig. 7: Build server task queue status in Mar-Apr, 2017

to the infrastructure or the developers, which requires just a few man-hour for each additional device type. Note that with the new Tizen prototypes, autonomous driving systems and on-device AI devices, Tizen platform developers have even not been required to know the existence.

Fig. 7 shows the build task workload by describing the waiting task queue lengths. The dark khaki lines, "Ready to build", show the number of packages ready to be built and waiting for resource allocations. The light blue lines, "Blocked build job", show the number of packages to be built, but not ready yet; i.e., even if there are resources available, we cannot build them. We have migrated to Tizen:Unified from per-profile builds on Apr 4, which is denoted by vertical line in the middle of the figure. Note that at this stage, Tizen 4.0 had been still in progress and Tizen 3.0 projects (per-profile basis) had been being built as well as Tizen 4.0 projects.

As we can see in Fig. 7, the peak queue length has been decreased dramatically from several thousands to less than a thousand. Normally, the queue length has been reduced to less than dozens or zero from hundreds in typical business hours.

Fig. 8 shows the number of busy build servers. We can again see the dramatic reduces of workloads after applying Tizen:Unified to the build system. After Tizen:Unified is applied, developers usually no longer experience any delays in work queues; in most cases, there have been available build servers waiting for developers! This increases the productivity of developers greatly by allowing developers to get the build and integration results ready for test deployment in shorter time (within an hour, not a day).

With Tizen:Configurability, enabled by Tizen:Unified, Tizen team has started IoT projects (https://wiki.tizen.org/Tizen_IoT) along with CraftRoom [7]. Further utilizing Tizen:Unified and Tizen:Configurability, a few developers provide continuous integration and deployment services along with software platforms, one for an autonomous driving system and another for on-device AI systems as well, which is named as TAOS. TAOS is continuously releasing its software platform binaries for both projects with frequent changes in its configurations or creations with new hardware sets and software requirements.

Without the results of Tizen:Configurability, the required build task workload and the complexity of choosing individual packages would have required far more man-month to support the two projects. We have two members related with TAOS support and most of their workload is due to provide guides to other developers, to implement supplementary developmental tools (profilers and emulators), or to port external software packages for other developers, not on configuring and test-building software platforms. Note that in the old days, we had needed several experts for such tasks.

VII. Lessons Learned

In this study, we have observed critical to-dos and not-to-dos. Most of they may seem to be simple rediscoveries of software engineering principles; however, they have been largely ignored in the development process.

- Do automate coding rule checks and prevent any mishaps from merging to the source repository. Otherwise, we are destined to lose in the battle against regressions, especially if we have a lot of developers with different backgrounds.
- Do not allow code divergences–normally, due to `#if` and `#ifdef` in C, that results in different binaries per device type. It is a bad technical debt for a software platform.
- Do not allow code divergences especially in header files or having different header files per device types. This is even worse; it is contagious to other software packages.
- Do promote run-time, boot-time, or install-time device-type discovery; use `if`, not `#if`. Using configuration files (.ini files) parsed in run-time or boot-time is also recommendable.
- Do not use any hints of device types or profiles in build configurations or build scripts, except for device drivers, firmware, and kernels. In build-time, the code and its build system should be agnostic to device types or profiles.

Fortunately, Tizen is now configured to mandate many of these, relieving us from such concerns. Commercialization projects usually have forked Tizen itself along with their own build infrastructure in their own company or business divisions, which makes it vulnerable to these concerns. However, as long as they remain as forked projects, not the mainline projects, any related issues are expected to be cleaned up

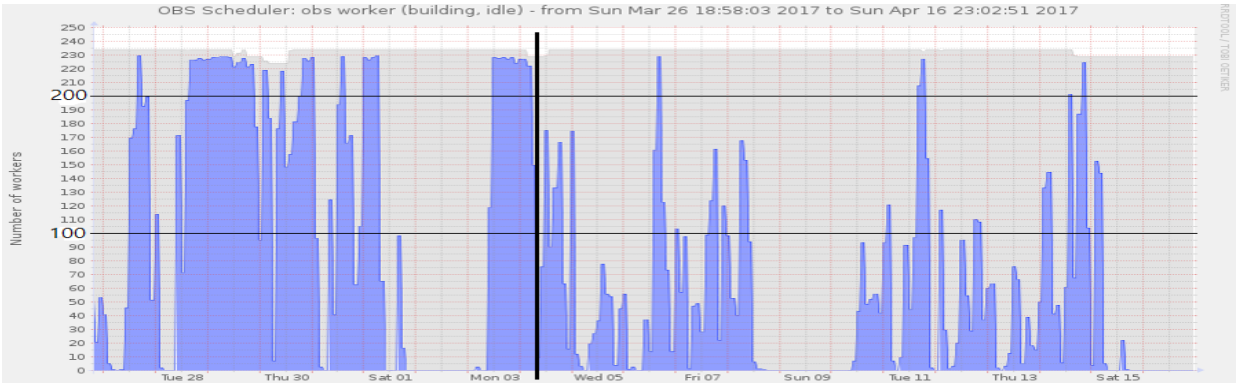

Fig. 8: Number of busy build servers in Mar-Apr, 2017. Tizen:Unified is applied at the vertical line.

at every new version releases Tizen; thus, they are ignorable threats for platform developers.

VIII. Conclusion

We have unified build projects and binary repositories of the software platform for various device types, improving the developmental efficiency, and proposed the concepts of Building Blocks and the highly configurable platform. In the due course, we have restructured the Tizen by refactoring hundreds of packages and implementing key infrastructures to support Configurability with Building Blocks. This work is successfully released via Tizen 4.0 and succeeded to Tizen 5.0 without any developmental overheads to keep the proposed mechanisms intact.

The productivity of both platform developers and infrastructures has improved significantly and Tizen has become capable of providing software platforms for various IoT and edge devices on-the-fly. The overall productivity of platform developers is improved by reducing turnaround time from code writing to integration and deployment and by reducing the number of binary packages to be created and tested for each source repository. According to the analysis in May, 2017, by Tizen team [4], the number of non-base packages built for a full build has been drastically reduced to 968 from 3,483. Moreover, after the full migration of build systems to Amazon Web Services (AWS), this work saves the cost of running AWS by reducing the number of build tasks. The improved configurability has allowed creating software platforms for various IoT devices, enabling IoT projects for Tizen and prototypes including autonomous driving systems and on-device artificial intelligence embedded systems.

References

[1] R. Want, B. N. Schilit, and S. Jenson, "Enabling the internet of things," *Computer*, vol. 48, pp. 28–35, 2015.

[2] D. Blackman, "Debian package management, part 1: A user's guide," *Linux Journal*, vol. 2000, no. 80es, p. 12, 2000.

[3] E. Foster-Johnson, *Red Hat RPM Guide*. Wiley New York, 2003.

[4] M. Ham, "Tizen unified & configurable: Create OS for any IoT and smart devices on-the-fly with minimal resources," Tizen Developer Conference, May 2017, talk.

[5] W. Kim, *Tizen IoT / Tizen Building Blocks*, Tizen, 2017. [Online]. Available: https://wiki.tizen.org/Tizen_IoT/tbb#Structure_of_Building_Blocks

[6] H. Lee, "What's next with Tizen?" Tizen Developer Conference Keynote, May 2017, keynote Speech.

[7] "Craftroom," Tizen, 2017. [Online]. Available: https://craftroom.tizen.org

[8] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 199–208. [Online]. Available: http://dx.doi.org/10.1109/ASE.2006.49

[9] P. Smith, *Software Build Systems: Principles and Experience*, 1st ed. Addison-Wesley Professional, 2011.

[10] J. A. Galindo, D. Benavides, and S. Segura, "Debian packages repositories as software product line models. towards automated analysis," in *ACoTA*, 2010.

[11] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in foss distributions: Details and challenges," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, ser. HotSWUp '08. New York, NY, USA: ACM, 2008, pp. 7:1–7:5. [Online]. Available: http://doi.acm.org/10.1145/1490283.1490292

[12] P. Abate and R. D. Cosmo, "Adoption of Academic Tools in Open Source Communities: The Debian Case Study," in *13th IFIP International Conference on Open Source Systems (OSS)*, ser. Open Source Systems: Towards Robust Practices, F. Balaguer, R. D. Cosmo, A. Garrido, F. Kon, G. Robles, and S. Zacchiroli, Eds., vol. AICT-496. Buenos Aires, Argentina: Springer International Publishing, May 2017, pp. 139–150, part 4: Case Studies. [Online]. Available: https://hal.inria.fr/hal-01776283

[13] M. Schroeder, *satsolver SAT Solver for package management*, 2007. [Online]. Available: https://doc.opensuse.org/projects/satsolver/SLE11SP3/

[14] "Zypper." [Online]. Available: https://en.opensuse.org/Portal:Zypper

[15] *Git BUild System*, 2014. [Online]. Available: https://source.tizen.org/documentation/reference/git-build-system

[16] *MIC Image Creator*, 2014. [Online]. Available: https://source.tizen.org/documentation/reference/mic-image-creator

[17] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: the linux virtual machine monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[18] "Open build service (obs)." [Online]. Available: https://openbuildservice.org

[19] E. Foster-Johnson, *Red Hat RPM Guide*. Wiley, 2002. [Online]. Available: https://books.google.co.kr/books?id=L5hOQj-pQrwC

[20] R. J. Streif, *Embedded Linux Systems with the Yocto Project*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2016.

[21] "Gerrit code review." [Online]. Available: https://www.gerritcodereview.com

[22] *Tizen Studio*, 2018. [Online]. Available: https://developer.tizen.org/development/tizen-studio

[23] "Tizen building block rules," 2017. [Online]. Available: https://git.tizen.org/cgit/tools/building-blocks/plain/RULES

[24] "Tizen kickstarter git repository." [Online]. Available: https://git.tizen.org/cgit/platform/upstream/kickstarter