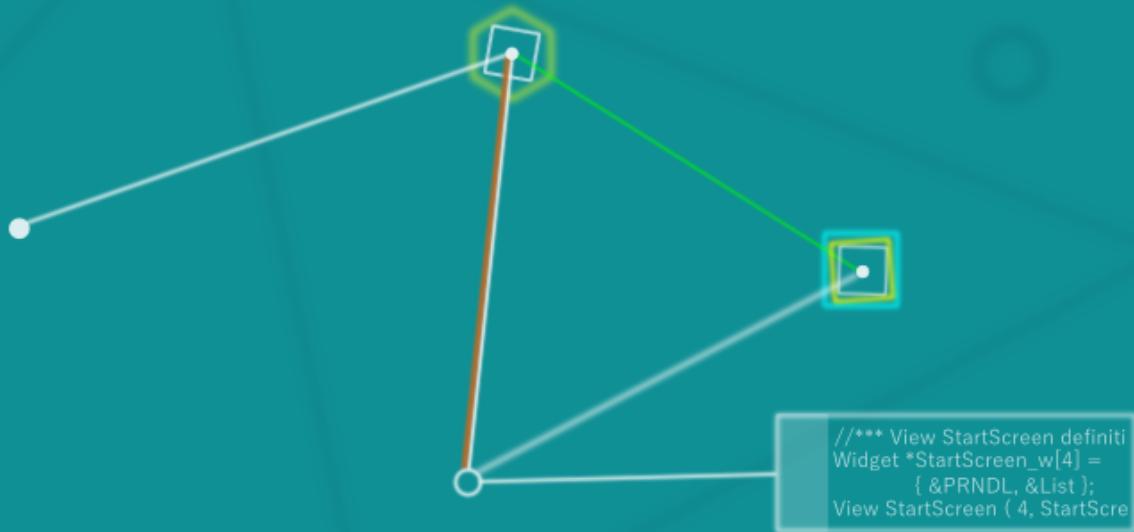


The HMI Design Framework **'DAM Designer'**

by  Intellics

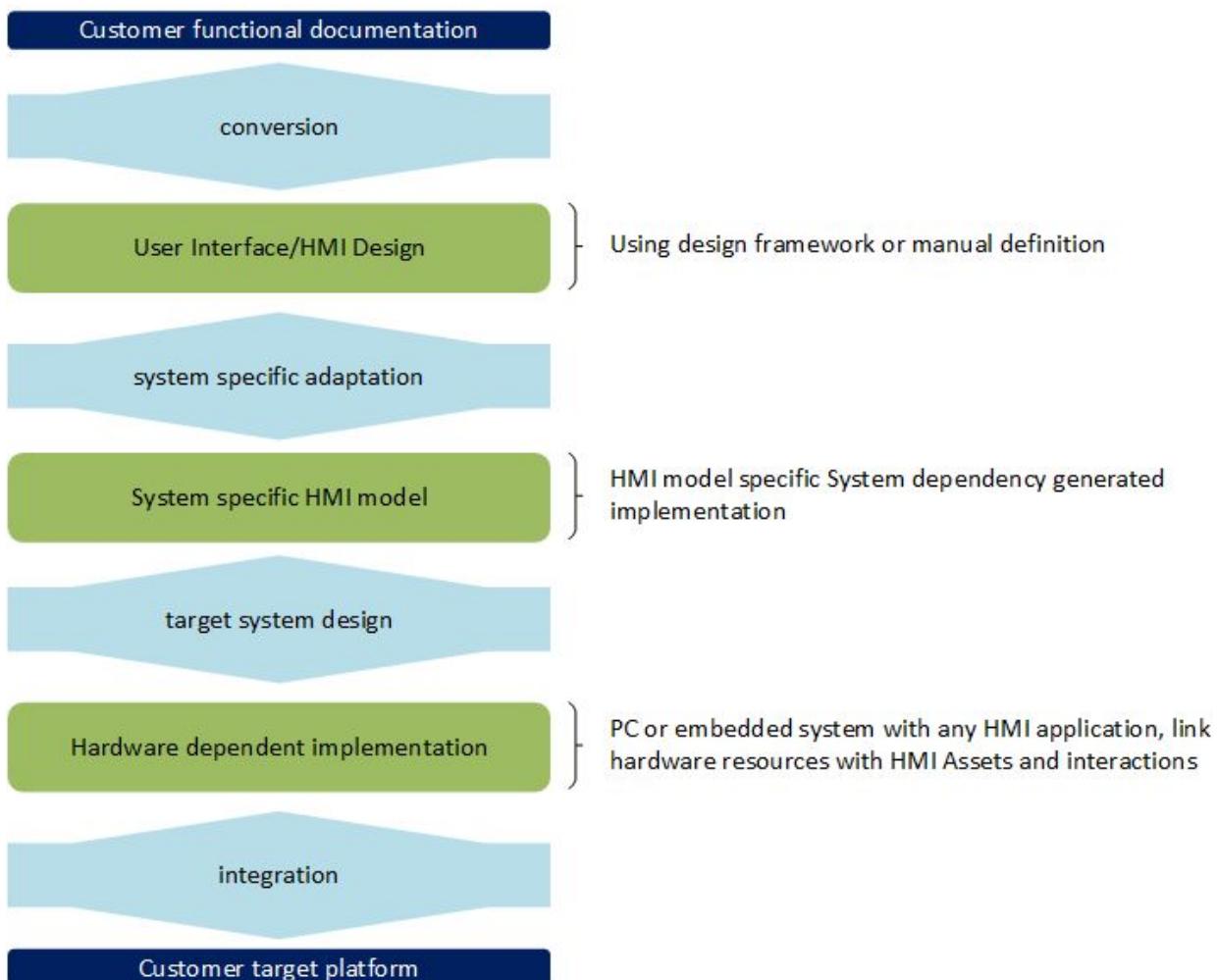


The missing link
in embedded HMI development

I. Overview

Intellics DAMDesigner is an end-to-end solution for designing and implementing HMI middleware. It provides an accomplished set of mechanisms for seamless integration of HMI stacks to underlying platforms incl. visual and textual modeling tool-chain, code generation, state machine modeling and management and target framework to code against.

The major rationale behind DAMDesigner is to facilitate the User Interface development flow as shown below by providing a solution for each major step and helping customers achieve their goals.



General User Interface Development Flow

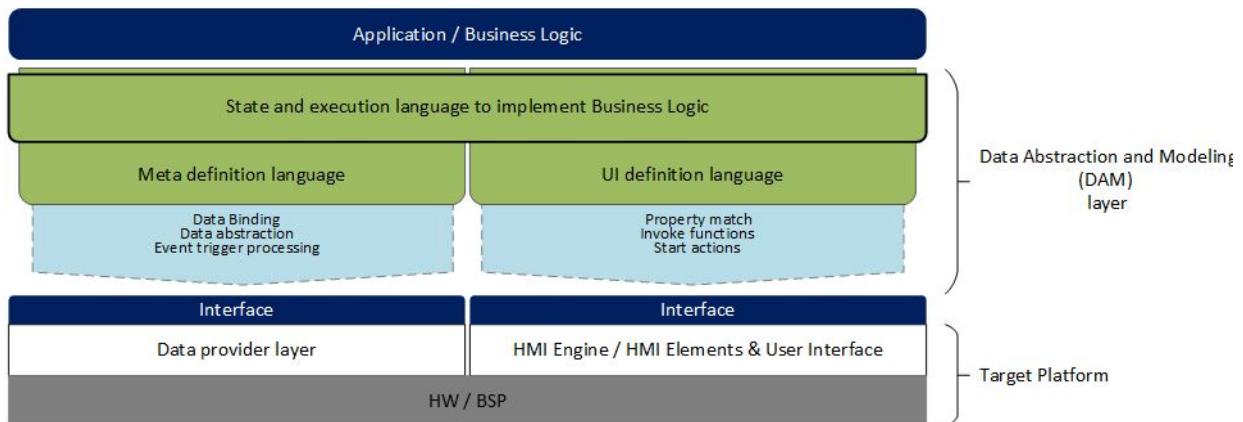
DAMDesigner is based on a flexible modular design and thus can be easily tailored towards variety of HMI stacks and Service platforms, incl. Rightware Kanzi, EB Guide, Qt, HTML5, GENIVI platform, AUTOSAR or practically any custom proprietary solution. An appropriate plug-ins are available to facilitate seamless conversion and interface generation.

II. Concept and Features

1. Overall concept

By integrating DAMDesigner into the HMI development workflow you can easily manage data coming from the underlying data storage (either communication messages data or data abstraction library output), configure events handling and define functions to process, bind and interact with the HMI model itself.

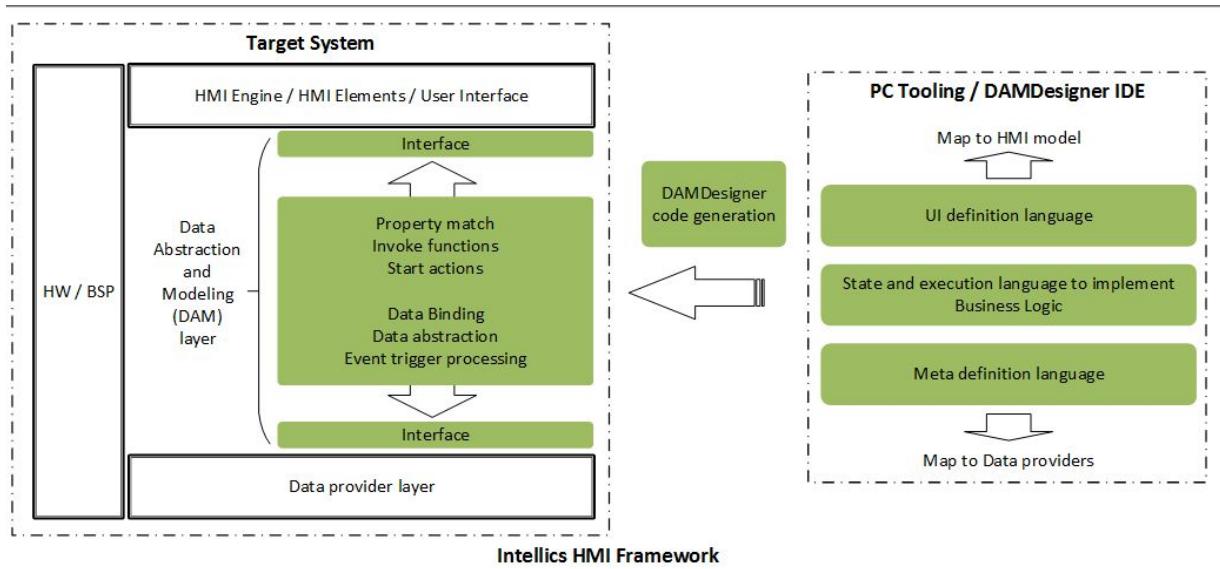
All underlying abstraction of data presentation, data or function binding and code generation - DAMDesigner framework is handling that fully transparently.



DAMDesigner concept view

2. Domain view

DAMDesigner provides an extensive set of feature that are realized both as a toolchain running on PC, as well as code-generation and frameworks that go to the target system itself. These two more or less separate domains are illustrated on the figure below.



DAMDesigner domain view

DAMDesigner IDE is a comprehensive Integrated Development Environment that allows you to:

- Model your GUI state machines both by using a textual DSL (domain specific language) and a graphical WYSIWYG editor. The textual and graphical models are automatically bi-directionally synchronized.
- Reference in your HMI model GUI elements which are described in a dedicated HMI description DSL. The DSL content can be generated out of the GUI model described in the GUI design tool itself, e.g. Rightware Kanzi.
- Reference in your HMI model System service interfaces which are described in a standard interface description language, e.g. Franca IDL.
- Do code generation from your HMI model to create the C/C++ sources that can build with the embedded project together.

The target System hosts the complete software stack where the HMI is actually executed. Normally that would be an embedded System of some kind running on a platform like Embedded Linux, Android, AUTOSAR, etc. However that could also be a PC application running on PC Linux, MS Windows, etc.

DAMDesigner provides a flexible set of components and frameworks for the target System that contain as much platform-independent code as possible. All of that comprises the Data Abstraction and Modeling (DAM) layer providing the following functionality:

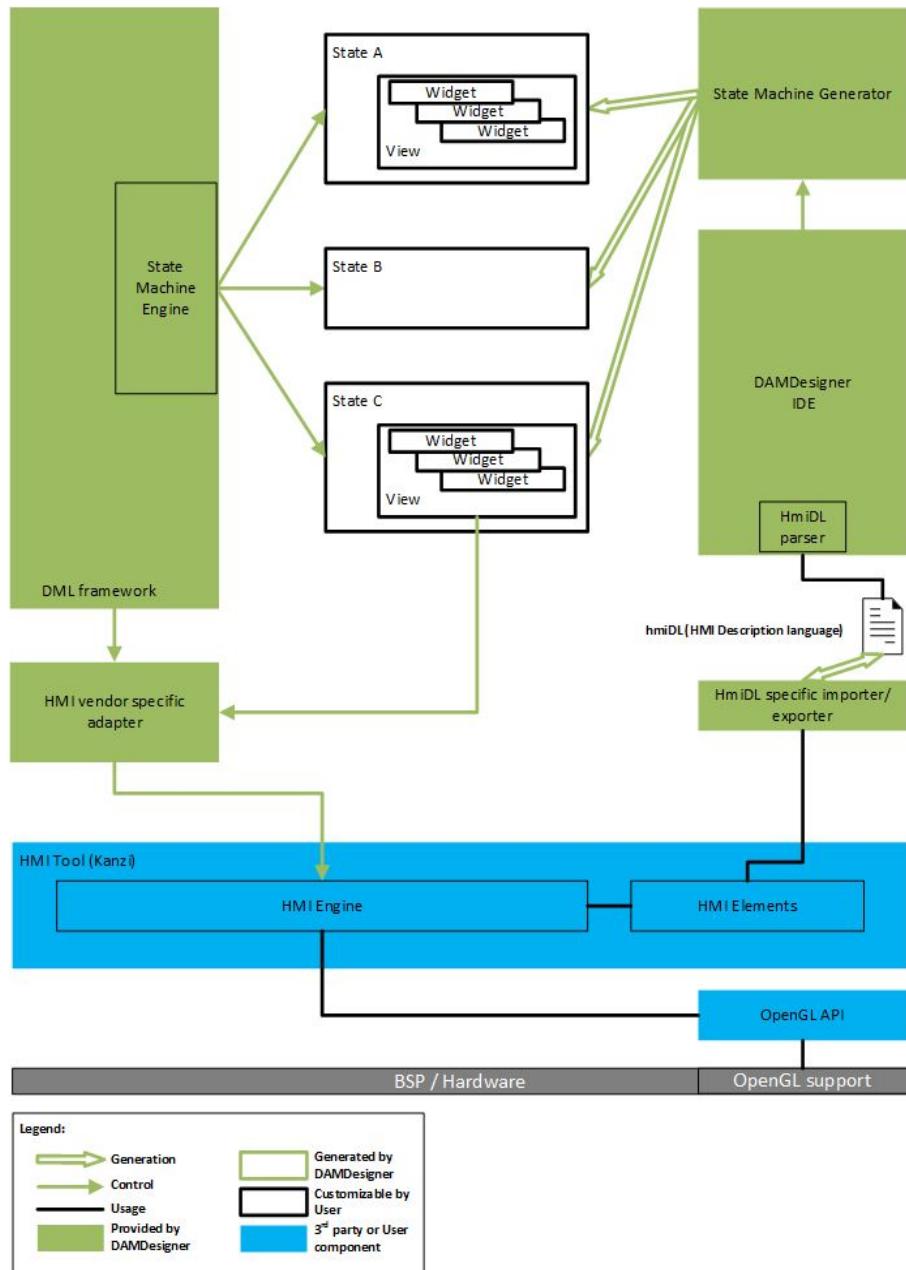
- A state machine engine that can actually “play” the state machine designed in the DAMDesigner IDE.

- Data-pool realization that can handle data in regards of caching, synchronization, queuing, etc.
- Provide bindings to HMI elements as they are referenced in the HMI model designed in the DAMDesigner IDE. This is realized by code generation according the HMI engine (e.g. KANZI SDK) APIs.
- Bindings to Service data interfaces as they are referenced in the HMI model designer in the DAMDesigner IDE. This is realized by code generation according the Service data interfaces (e.g. DBus binding according the FIDL definitions).
- Provide code generation of application skeletons for user-specific implementation.

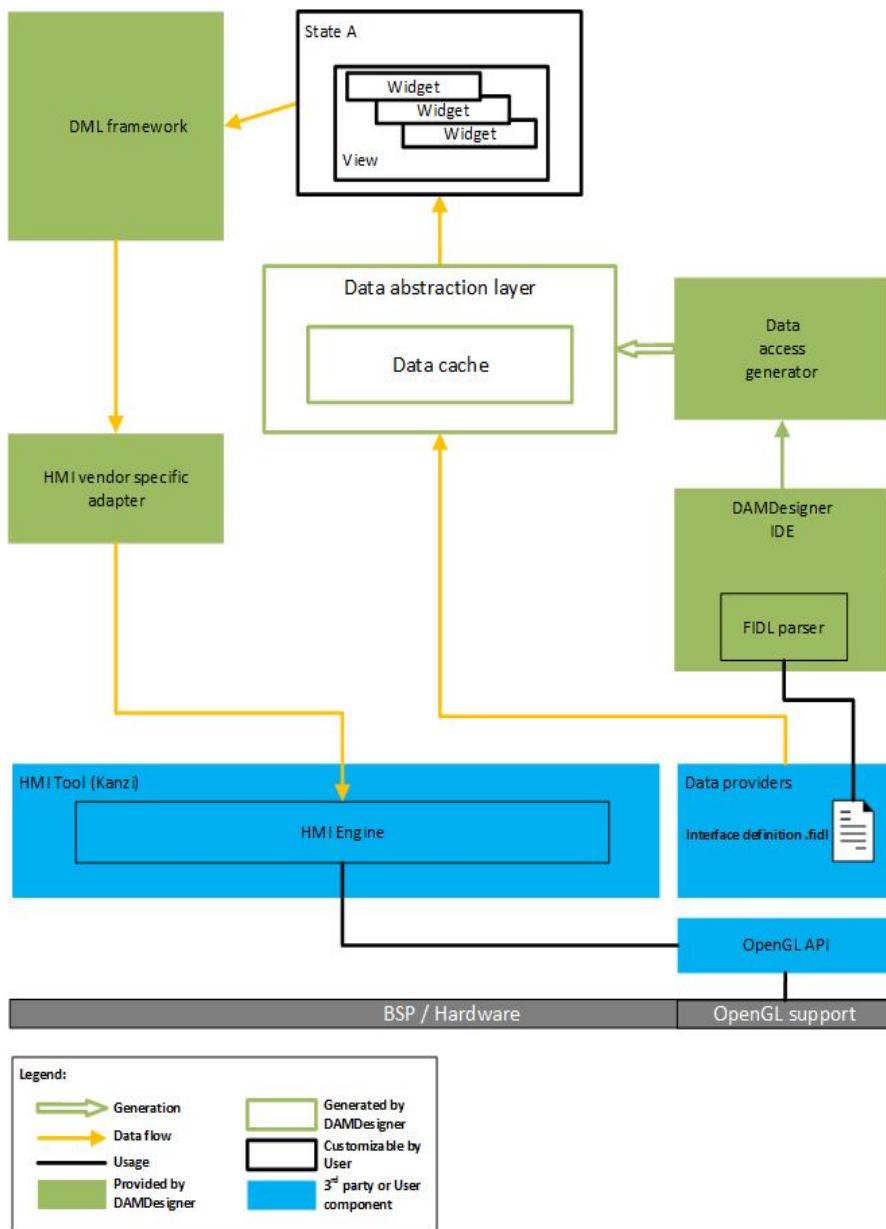
3. HMI and Data Concepts

DAMDesigner's concept architecture regarding HMI and data management is shown on the figures below respectively. Do your modeling in terms of HMI element definition, state design, connections to System data interfaces, etc. in the DAMDesigner IDE. Data interfaces and HMI elements can be directly referenced, as already described in the System's Interface Definition Language and HMI tool itself. I.e. you can "bind" a GUI control property to the value provided by a particular Data provider interface. No intermediate descriptions, wrappers and mapping are needed.

Further, generators are involved to generate source code related to HMI state definition and data connections. Generation is done against the DML framework as well as the particular Data providers and HMI engine APIs.



DAMDesigner HMI concept



DAMDesigner Data concept

4. HMI prototyping

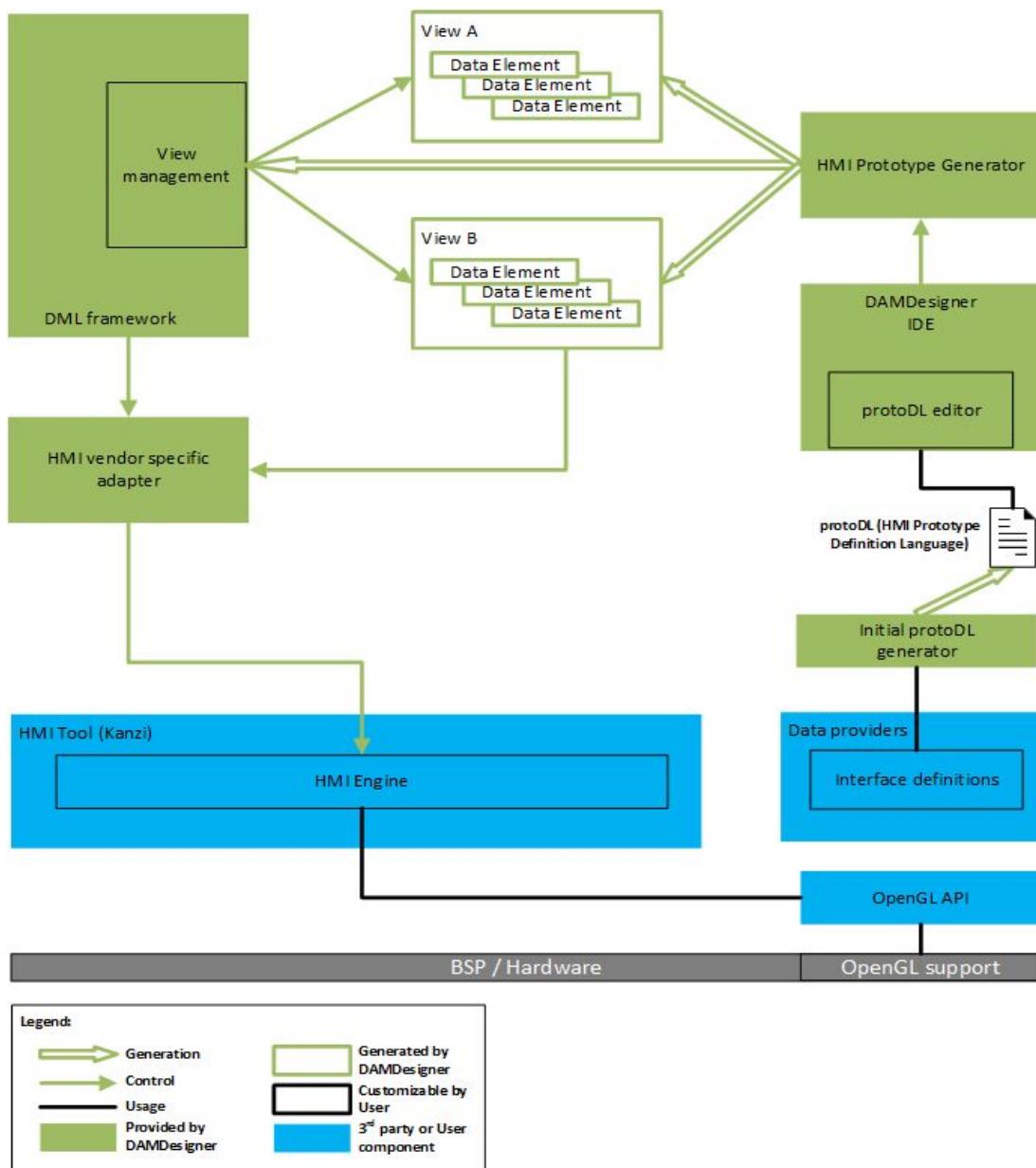
DAMDesigner also provides a solution for simplified HMI development, called RapidHMI. It is utilized by means of simplified descriptions and generation of default screens, controls, states, etc. out of the Data provider interfaces.

You can generate a complete prototype HMI application out of your interfaces definition by just a few clicks in DAMDesigner IDE. The prototype application generated will provide basic data visualization

and state management, based on predefined rules. No additional modeling or configuration is necessary in the referenced HMI tool.

RapidHMI addresses the common use-case of decoupling HMI and underlying service development. Data providers can be bound to such a prototype HMI application (and thus execute in a fully integrated System) much before the fully-featured HMI has been developed. The purpose is to visually monitor and test the underlying system behaviour in the real environment.

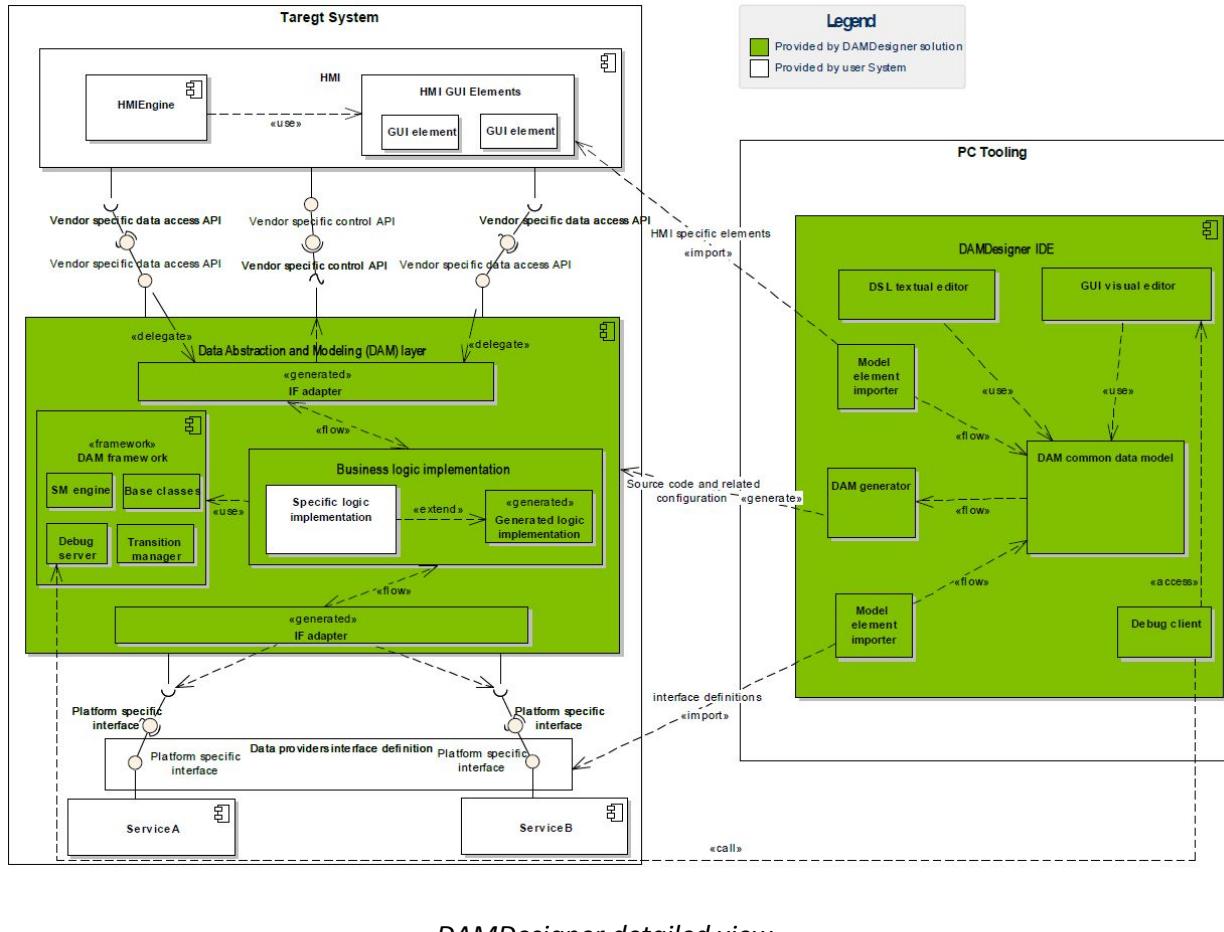
The overall concept regarding RapidHMI is illustrated on the figure below.



DAMDesigner RapidHMI concept

III. Detailed View

The main elements composing DAMDesigner in terms of both PC Tooling and Embedded System side are shown on the diagram below.



The *DSL textual editor* and *GUI visual editor* both can be used by the user to construct and modify the *DAM common data model*. Since data they operate on is common, both editors' content is automatically synchronized in both direction. I.e. you can either “type” your HMI state machine design, data bindings, options, etc. in a dedicated State Machine description language (stmDL) or “draw” those in the visual editor. And you can switch forth and back at any moment...

The *Model element importers* also operate at the PC-side. The HMI importer generates a dedicated HMI description language (hmiDL) out of the content of the HMI design itself. Then the content generated can be used directly in the *DAM common data model*, by referencing it in the editors. Similar role is played by the Data Provider interface imported. It allows that those interfaces are

referenced in the user's model, i.e. you can directly bind them to HMI properties, states, views, etc. No wrappers, types mapping and other kind of domain translation are needed.

The *DAMGenerator* is the main “link” connecting the PC tooling and the Target System by performing code generation according the *DAM framework*. The framework itself provides a *State Machine engine* that can execute the state machine designed under the *DAMDesigner IDE*, *Transition Manager* to handle HMI view transitions, *Debug Server* which enables debug features like state machine animation during program execution over TCP/IP and of course a library of *Base classes* that provide foundation for the code generation and user custom implementation.

The *Business logic implementation* is a blend of generated and custom user code. Code generation is done regarding the model described in DAMDesigner by stmDL and hmiDL. User's *Specific logic implementation* is needed to extend the generation and realize the actual custom behavior of the System. Template methods and functions are used as much as possible to provide a convenient set of placeholders to write or reference custom implementation.

The Interface adapters (*IF adapter*) are constructed by platform-specific code generation, according the APIs of the exact *Data providers interface definition* and *HMI Engine*, regarding the mappings and bindings defined in the user's HMI model. I.e. all the “connectivity” code related to accessing data from D-Bus or similar communication system, setting attributes of and providing control command to the HMI Engine is generated according the user's definitions defined in the *DAMDesigner IDE*.

IV. Comparison matrix

Feature	DAMDesigner	Qt	HTML5
<i>General</i>			
Awareness of HMI model and semantics during state modeling	Yes (hmiDL DSL representing Views/Widgets/Properties, etc.)	no* *can be achieved via custom implementation	Yes
Implicit synchronisation (sync State machine with HMI execution)	Yes (dedicated component)	no* *can be achieved via custom implementation	no

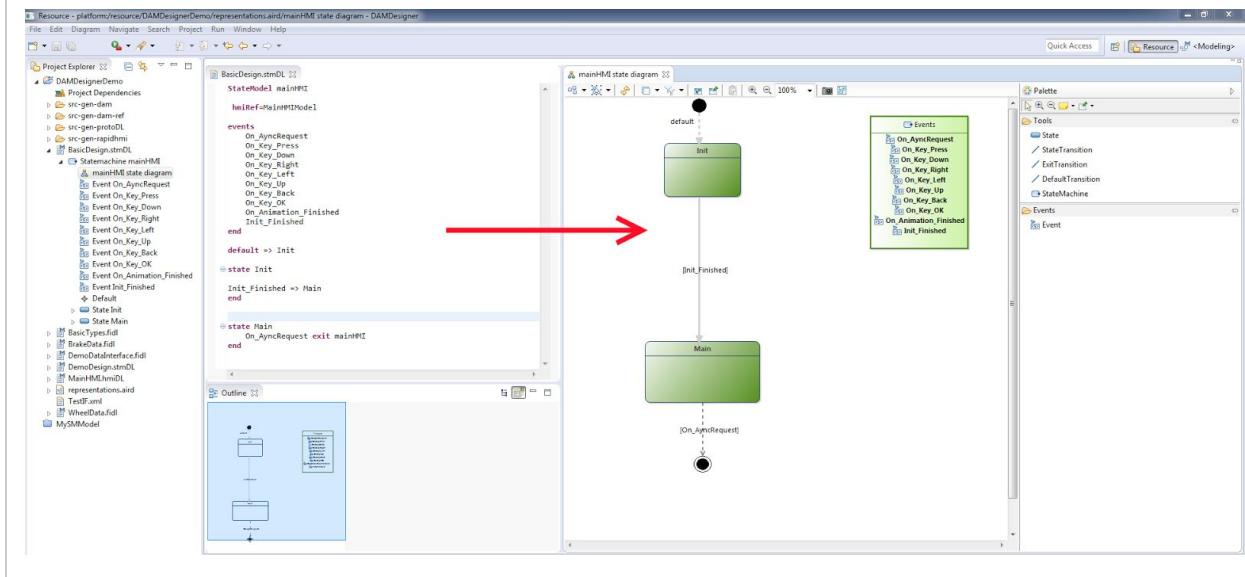
Modular based design with explicit exchangeable interaction layers	Yes	no	no
<i>Data access and binding</i>			
Reference data providers service interfaces	Yes (binding to FIDL definitions)	Yes (database, cloud...)	no
Reference HMI visual elements	Yes (currently Knazi widgets and properties supported)	Yes (QT widgets)	Yes
Binding HMI visual elements to service interface elements	Yes	no* *can be achieved via custom implementation	no
<i>State machine</i>			
Support of parallel state machine	Yes	Yes	no* *can be achieved via custom implementation
Support of events and transitions	Yes	Yes	no* *can be achieved via custom implementation
Support of custom "hook" functions	Yes	Yes	no* *can be achieved via custom implementation
<i>Tooling and User interface</i>			
Dedicated business logic modeling tool (describe state machines, data references, bindings, etc.)	Yes (DAMDesigner IDE)	partial* *3rd party modeling tools	Yes* *no embedded - dedicated solution
Support of textual modeling	Yes (Eclipse DSL language: highlighting, in-editor real	partial (c++ code)	Yes

	time validation, quick fix, etc.)		
Support of visual modeling	Yes (bi-directional sync with the textual definition)	no	no
<i>Code generation</i>			
C/C++ code for embedded target system	Yes	through source code	no
Service data access/binding code generation	Yes	through source code	no
<i>Debug features</i>			
Support of state machine player (for debug)	Yes (TCP/IP connectivity to target system)	no	no* *can be achieved via custom implementation

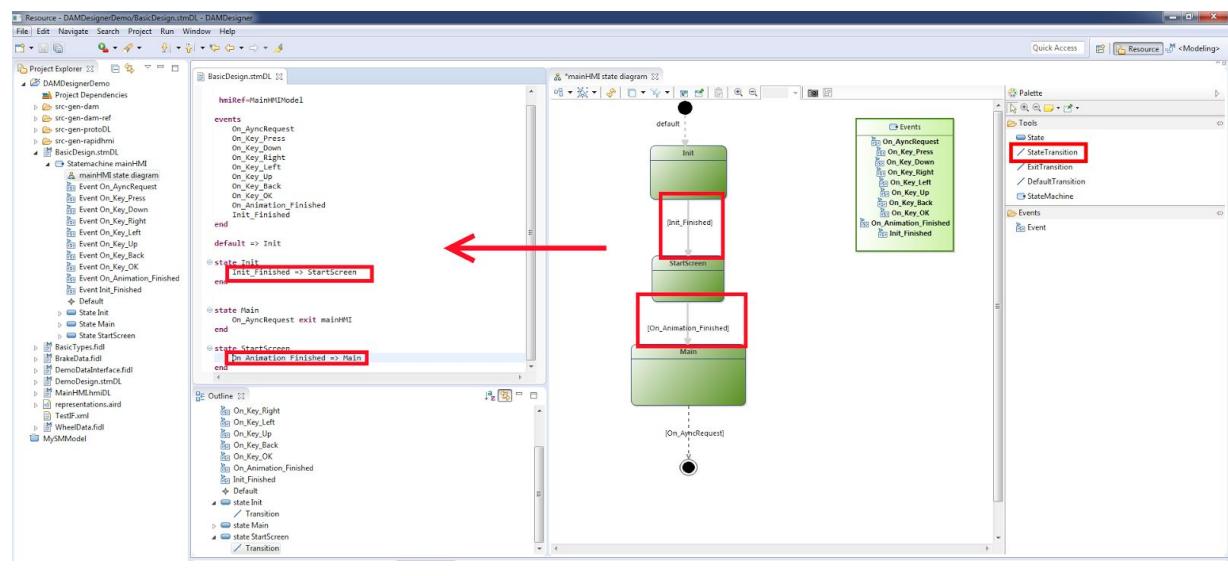
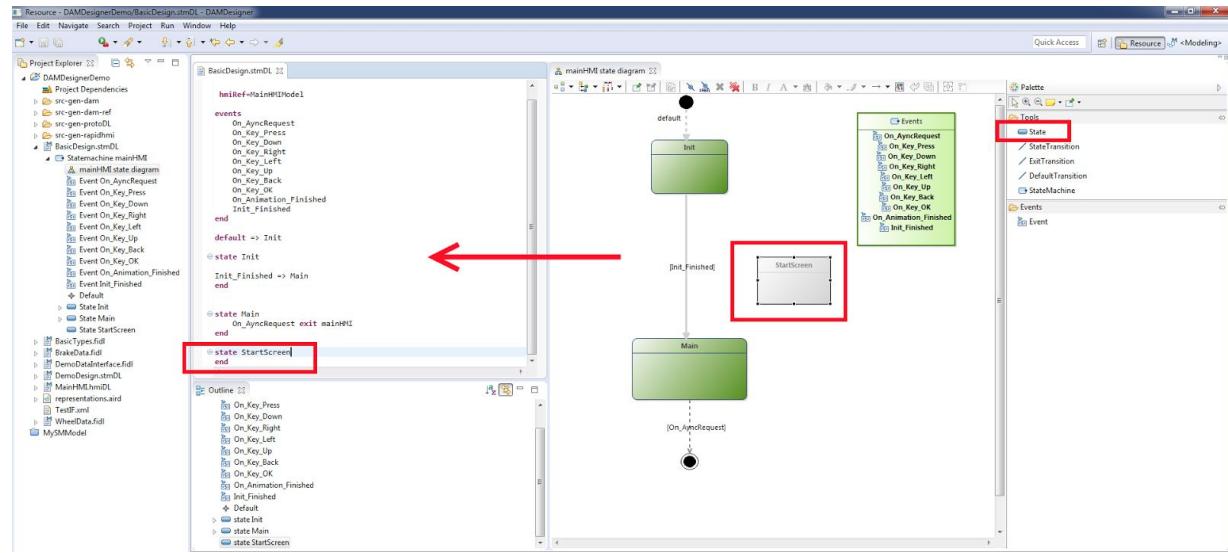
DAMDesigner By Example

because a picture is worth a thousand words

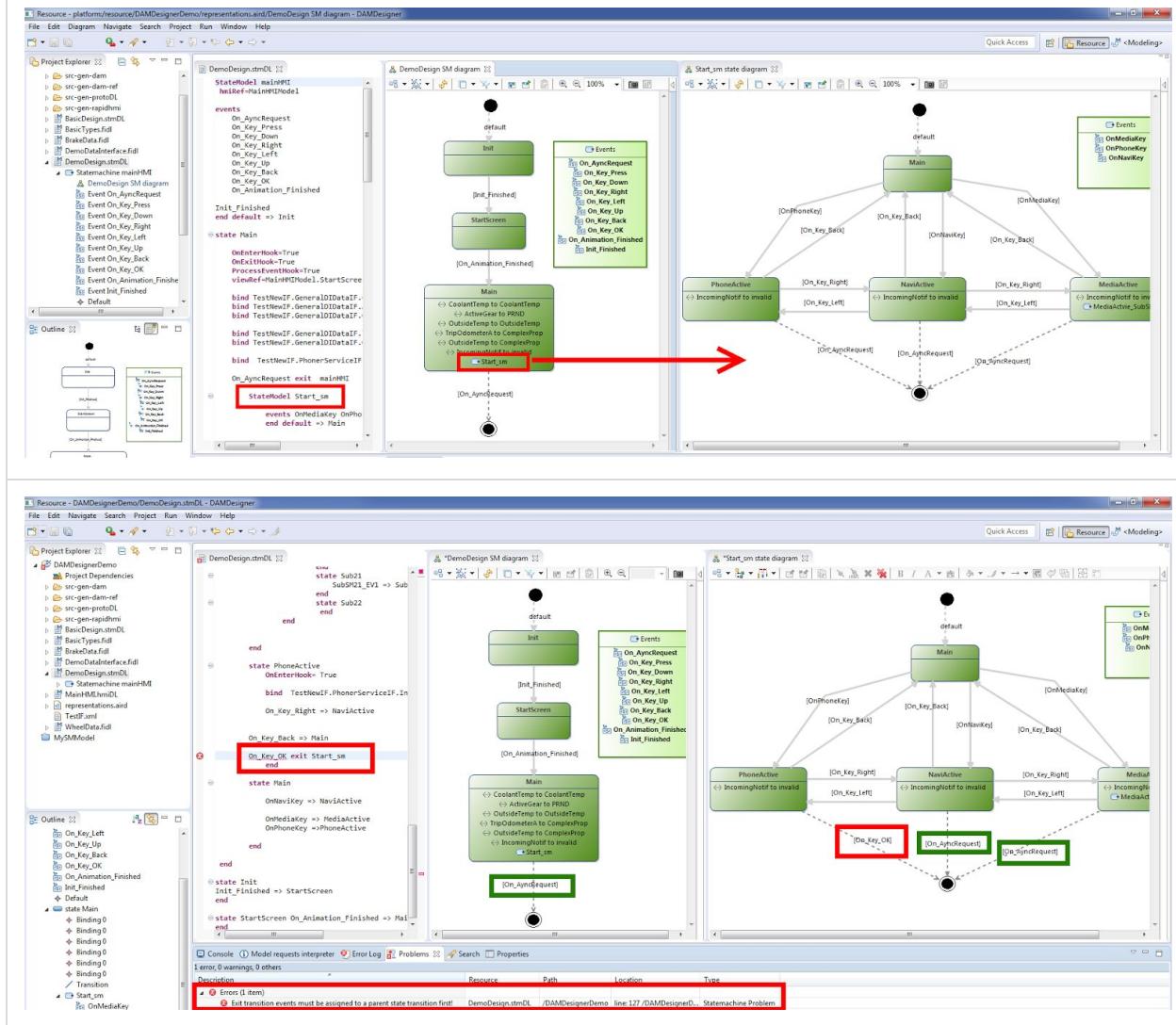
*In the DAMDesigner IDE you can start with defining states and transitions in *stmDL*. The GUI representation is **automatically** constructed (yet you may change some purely visual attributes like position, color, etc.)*



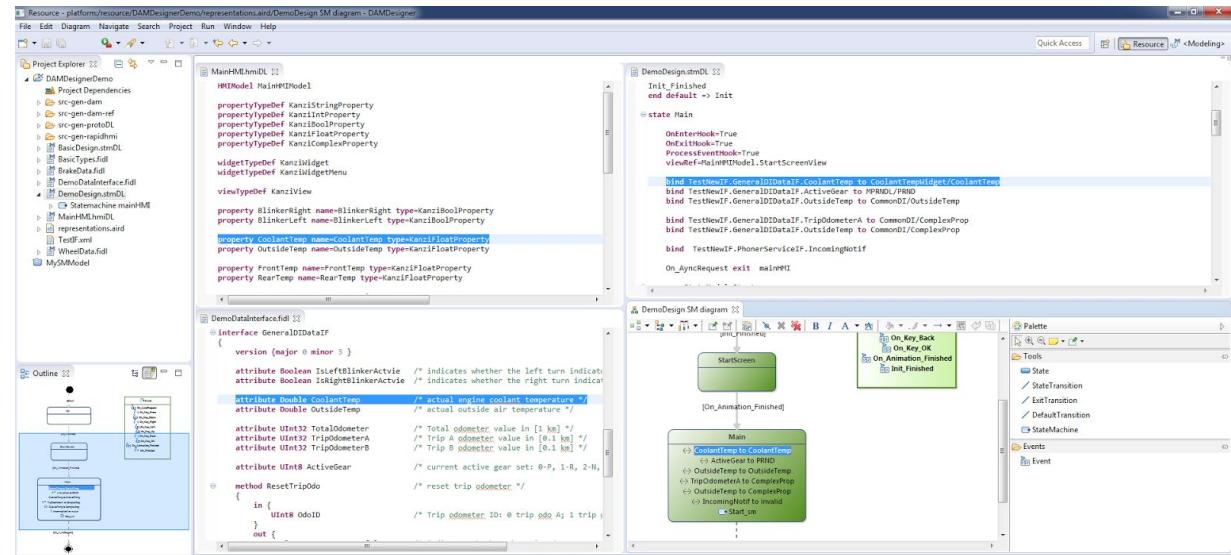
Then you can add new states and transitions in the GUI editor itself. The *stmDL* will be automatically “aligned” to your design.



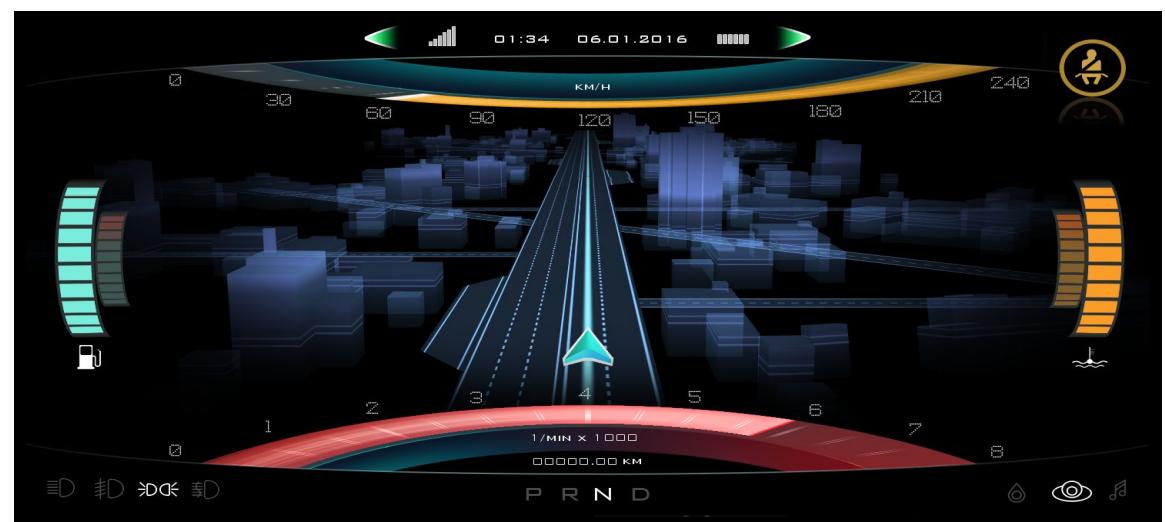
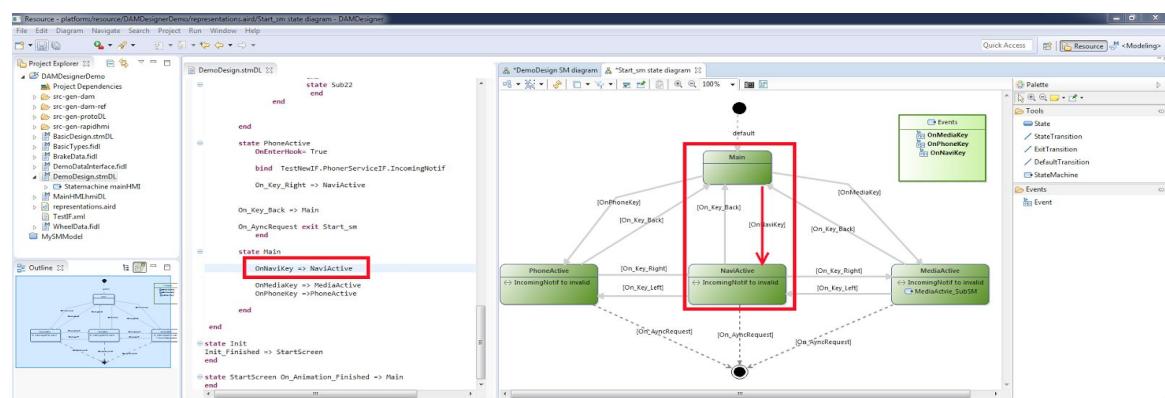
State model nesting with corresponding semantic handling is allowed.



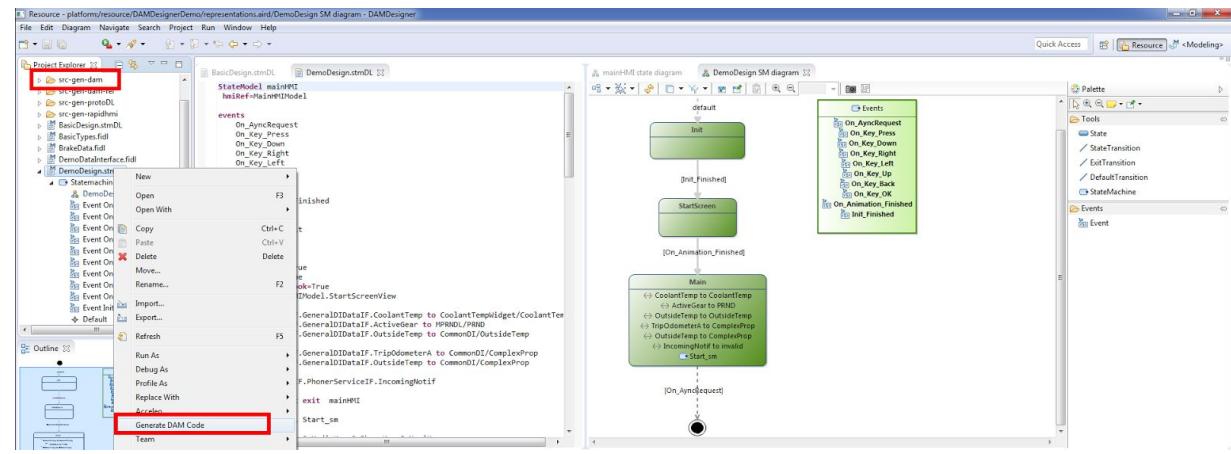
In DAMDesigner you can bind FIDL data elements (attributes) to HMI user elements (nodes, properties).



You can also model states, transitions and events that will drive the HMI model itself.



*Code generation is performed on command from the user's model.
Some fragments shown below.*



The screenshot shows the DAMDesigner interface with the "mainHMI_sm_hook_functions_tmpl.cpp" file open. The code defines hook functions for the "Start" state:

```

#include "view_definition_g.hpp"
#include "event_definition_g.hpp"
#include "main_sm_definition_g.hpp"

//----- State Start hook functions -----
ErrorType OnEnter_Start()
{
    // TODO: place custom implementation here
}

ErrorType OnExit_Start()
{
    // TODO: place custom implementation here
}

//----- State PhoneActive hook functions -----
ErrorType OnEnter_PhoneActive()
{
    // TODO: place custom implementation here
}

```

The screenshot shows the DAMDesigner interface with the "view_definition_g.cpp" file open. It contains key event handlers for the "Start" state:

```

#include "work_includes.h"
#include "custom_properties.hpp"
#include "widget_definition_g.hpp"
#include "main_sm_definition_g.hpp"

#ifndef __cplusplus
extern "C" {
#endif // __cplusplus

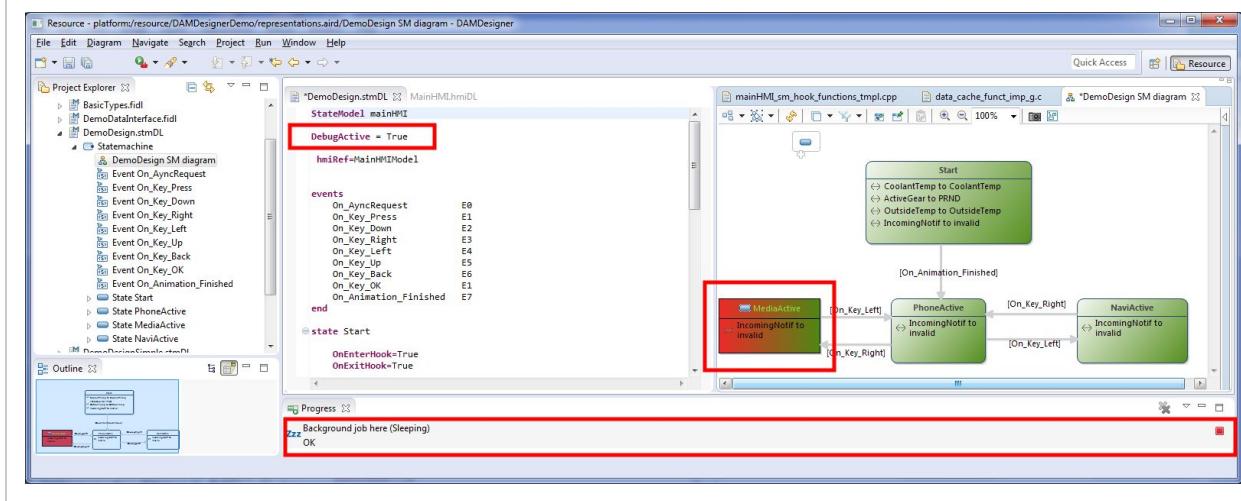
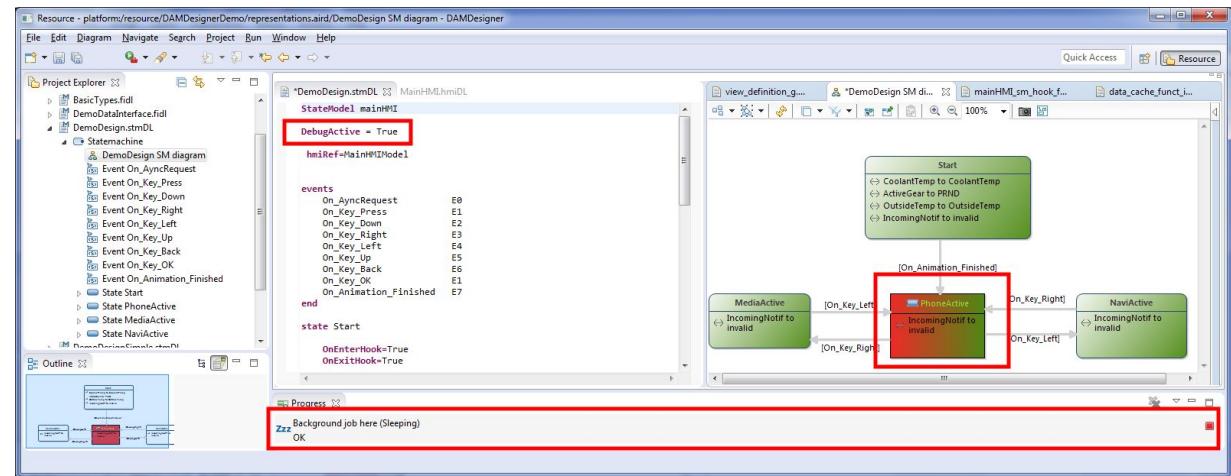
void dp_IncomingNotif_cb(dp_string_size_t size, const int8_t * const value)
{
    // TODO: place custom implementation here
}

#endif // __cplusplus

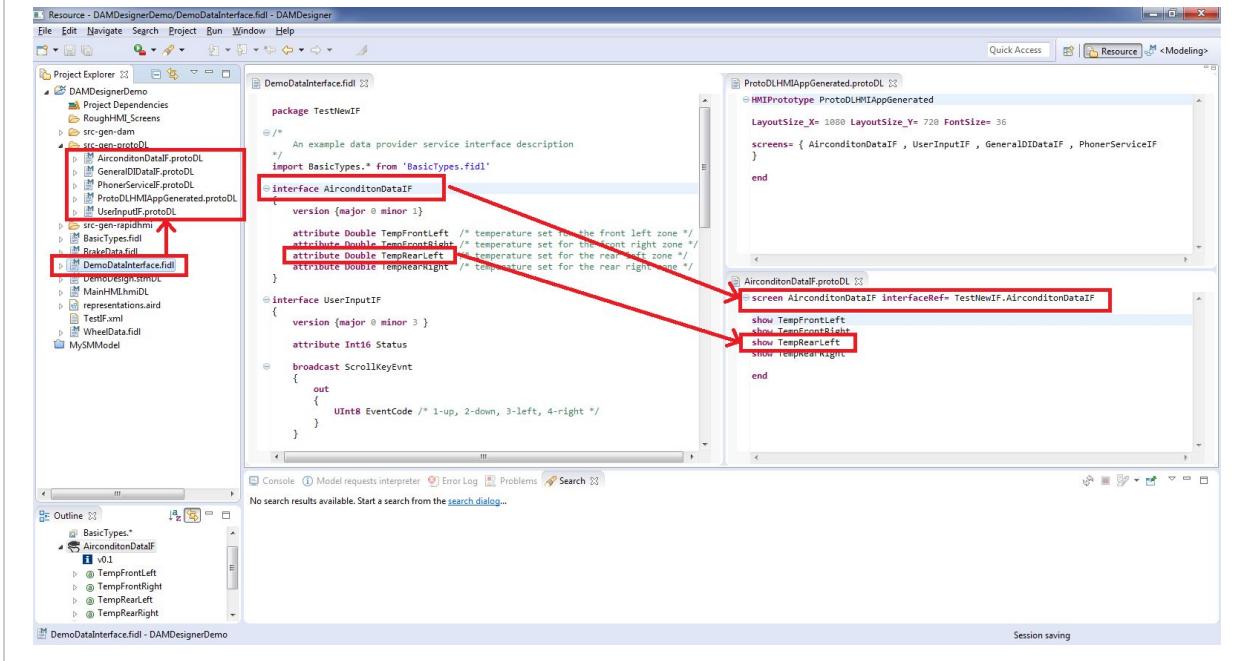
```

You can activate “Debug” mode for your model, then the state machine player is started as a background job within the IDE, so enabling visual indications on the state machine design during program execution on the target System*.

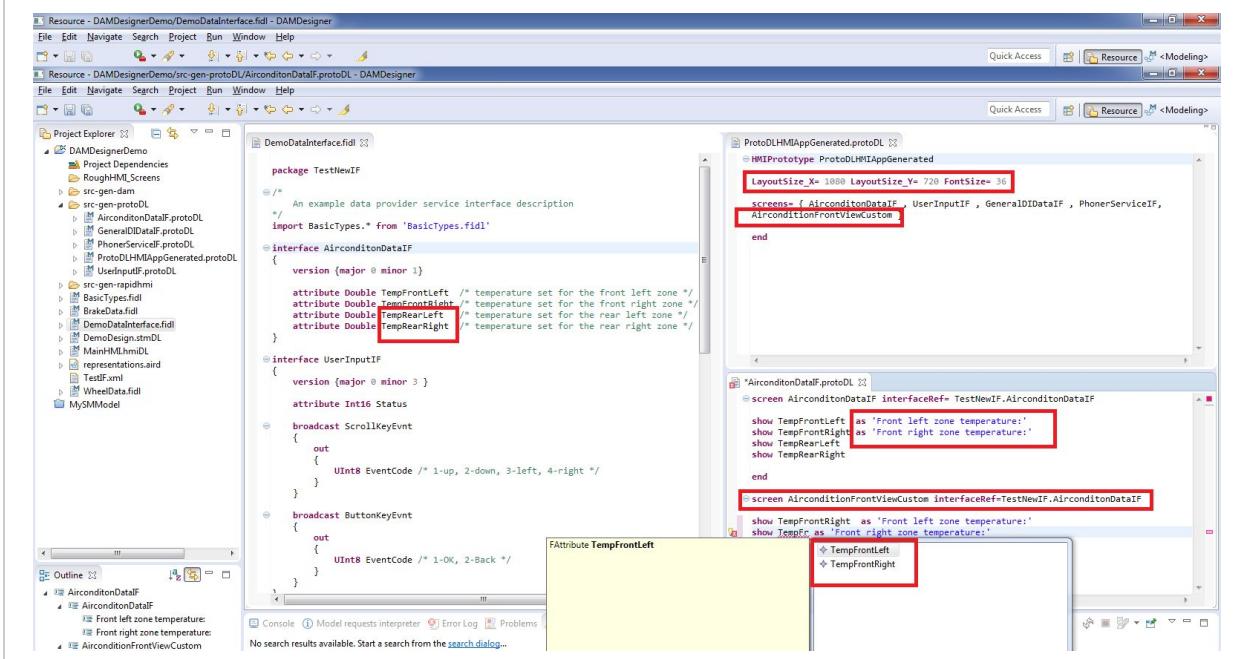
* TCP/IP connectivity to target is required.



A HMI prototype application (RapidHMI) is defined by the Prototype Description Language (protoDL). You can write your protoDL manually, generate it out the FIDL definitions, or go for a mixture of both: initial generation plus manual additions, if needed.



FIDL data elements are directly referenced in protoDL. Some extra visualization properties like font size, label names, screen resolution, etc. can be additionally defined.



The source code of the target HMI application is fully generated by the protoDL and FIDL definitions. All FIDL data that is mapped by the **show** clauses is displayed on the corresponding screen.

