presented at: 5th Conference Simulation and Testing for Automotive Electronics, May 10-11, 2012, Berlin.

Software-in-the-Loop Using Virtual CAN Buses: Current Solutions and Challenges

Thomas Liebezeit, Andreas Junghanns, Mirco Bonin, Roland Serway

Abstract

Software is used today to perform ever more complex automotive functions. This may involve local sensors and actuators; however, sensors, actuators and functions of remote control units are increasingly being used to implement functions of electronic control units (ECUs).
Remote access requires some means of inter-ECU communication. The most commonly used standard in the automotive domain today consists of Controller Area Network (CAN) buses [1].
The development of distributed automotive software functions requires an excellent understanding of the dynamic effects and rigorous testing of these functions. The continuing thrust towards virtualizing development and test (MiL and SiL) makes it increasingly important to support virtual buses.
This paper shows how introducing CAN bus communication into SiL platforms can significantly improve simulated functionalities with a simultaneous dramatic reduction in the overall setup costs by using CAN description formats (e.g. DBC) as specification for the communication between SiL components, such as ECUs, plant models and/or rest bus simulations.

Kurzfassung

Die steigende Komplexität von Fahrzeugfunktionen wird zunehmend durch Software realisiert. Solche Funktionen nutzen dazu hauptsächlich lokale Sensoren und Aktuatoren. Allerdings greifen solche Funktionen auch immer häufiger auf Sensoren, Aktuatoren und Funktionen anderer Steuergeräte zu.
Dieser Fremdzugriff benötigt eine Form der Inter-ECU-Kommunikation. Der heute häufigste Standard für diese Art der Steuergerätekommunikation im Fahrzeug ist das Controller Area Network (CAN) Protokoll.
Die Entwicklung solcher verteilter Fahrzeugfunktionen verlangt grundlegendes Verständnis der dynamischen Effekte in solchen Netzwerken und deren umfassenden Test. Die Zunahme der Virtualisierung in der Steuergeräteentwicklung und –test (MiL und SiL) macht eine Verfügbarkeit von CAN-Netzwerken in diesen virtuellen Entwicklungsumgebungen wünschenswert.
In diesem Vortrag zeigen wir, dass die Nachbildung der CAN-Kommunikation in SiL-Plattformen nicht nur die Zahl der simulierten Funktionen erhöht, sondern gleichzeitig sogar den Setup-Aufwand verringert, weil es möglich ist, ganze Busse durch die Nachnutzung von Netzwerkkommunikationsbeschreibungsformaten (z.B. DBC) mit minimalem Aufwand zu definieren.

1. Motivation

Increasing pressure to save time and cut costs in the development and testing of more and more complex automotive systems requires improved methods for development and testing. Software-in-the-Loop is one such method used to front-load development tasks to a time when physical prototypes do not yet exist. This is achieved by coupling the control software (as virtual ECU) with a simulation of the plant (e.g. a detailed transmission embedded in a roughly modeled car) [2].
Modern powertrain components are connected not just to each other but also to other ECUs in the car. Values and commands are communicated, such as RPM, pressures, temperatures, user-selected values, torque reduction requests and live messages. The most common way to transmit these values is by Controller Area Network (CAN).
The specific implementation of such a network is one of the central development efforts. At the moment, these implementations can only be tested using HiL setups, which entails waiting until physical ECUs are available. Changes in the network configuration require a long change test cycle because a new HEX file has to be sent to all the affected ECUs; this process often involves suppliers.
Current SiL setups connect a large number of signals to (potentially multiple) ECUs and (potentially multiple) plant simulations, with or without rest bus simulations. Signal properties have to be defined for each signal to ensure correct interactions. This is largely a manual process and thus costly and error prone.
Using an existing CAN specification format to configure the communication behavior of each of the SiL components can therefore drastically reduce the effort for defining and implementing the (CAN) communication, while at the same time reducing communication setup errors. For instance, CAN specifications define message and signal names, signal units and their scaling as well as message timing properties.
Moreover and more importantly, when developing some layers of the CAN stack, the inclusion of these layers in the SiL setups entails sending and receiving CAN messages, not just the contained signals.
When CAN messages are sent through the SiL, it will be possible to change message transmission delays and thus test the correctness and robustness of the network communication.
Last but not least: HiL plant models (usually modeled in Simulink®) and rest bus simulations use specific CAN blocks to send and receive CAN messages and their signals. A SiL platform that supports CAN by offering a specific SiL CAN block set can make it very cost-effective to reuse HiL models in a SiL environment.

2. CAN Basics

In a CAN network, messages are transmitted from one transmitter node to many receiver nodes. A message may have up to 8-byte data which encode a number of signals. Therefore, a signal may be between 1 and 64 bits wide.
DBC files specify a CAN network, including the network nodes, the messages sent and received by each node, which signals are contained in a message, how they are encoded and how often a message is sent (cyclic or event-based).
Using a DBC file, it is possible to generate C code that implements the send and receive functionality of a node or to configure an emulation of such a CAN stack.
Each CAN message has a unique priority. Sending multiple messages at the same time means that the message with the highest priority is sent.

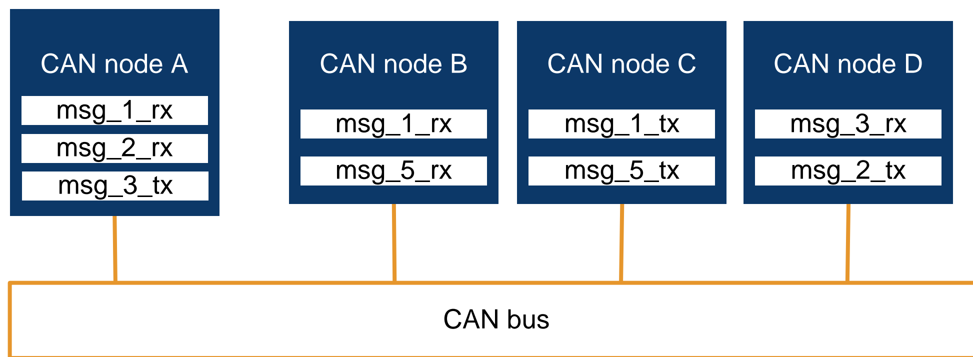

Figure 1: CAN bus

CAN communication is affected by slight variations in transmission delay or even lost messages. Higher-level protocol layers may implement measures to deal with such effects in the transport layer. Messages may contain CRCs and message counter signals in order to ensure message and protocol integrity.
More information about CAN can be found here [1].

3. CAN buses in Silver

CAN bus support in Silver has been added to the Silver Basic Software (SBS) library. This is a C library offering an API similar to ECU basic software layers. Among others, Silver Basic software simplifies access for the application layer of the ECU software to the services of the real-time OS, access to the electronic peripherals (actuators and sensors into the plant) as well as communication stacks, such as a CAN driver.
Silver Basic software is exactly this: an easy-to-use API gives access to a wide range of functions available through Silver and a real-time OS emulation for connecting the application software with the environment.

3.1 CAN transport layer in Silver

Silver offers a transport layer for CAN messages. A message can be sent at a specific moment in time by a Silver module and received by any other module. Silver supports multiple buses. It is possible to inspect each CAN message in the Silver GUI offering simple debugging functionality.
There are two libraries for convenient access to Silver CAN messages:

1. The SBS library is used to send and receive CAN messages from C code, see 3.2.
2. A Simulink block set interfaces with the same C library when exporting the model using Simulink Coder® (formerly Real-Time Workshop®), see 3.3.

There is a number of differences with respect to physical CAN transport layers in the current Silver implementation, however:

1. Signals are transmitted instantaneously and can be received in the next Silver macro step.
2. The bus bandwidth is infinite – all signals are transmitted, regardless of message priority and number of messages sent.
3. If the same message is sent twice during the same macro step, only the last message is transmitted.

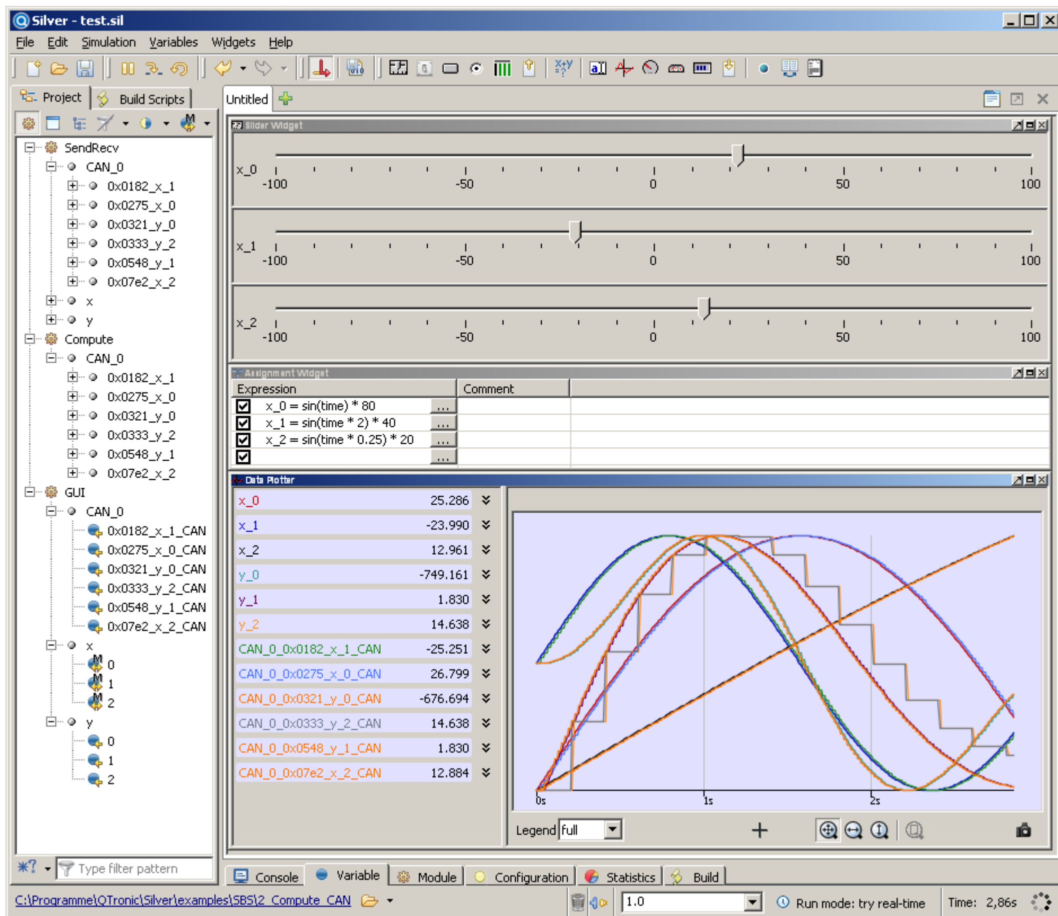

Figure 2: Silver GUI with example CAN

To improve or even remove problem 1) and 2), it is possible to implement modules that delay and lose CAN messages when protocol robustness analyses are required. Such modules could also restrict bandwidth and implement a priority resolution schema. Silver offers comfortable delay functions (in the modifier module and in the assignment widget) that can be used for this purpose.

3.2 The C CAN library functions

The SBS library offers functions to access CAN messages in Silver. The following is a brief overview of the most important functions, as a complete description of all functions would go beyond the scope of this paper.

- `SBS_CAN_RxMsgStatus`: Used to inquire if a new message has arrived
- `SBS_CAN_RxMsg`: Receive a message
- `SBS_CAN_TxMsg`: Send a message
- `SBS_CAN_TxMsgStatus`: Verify sending of a message

There are, of course, functions to start and stop communication, as well as sending and receiving messages, very similar to traditional driver interfaces.
The main difference, however, is that there are a number of functions for accessing individual signals conveniently, such as:

- `SBS_CAN_RxMsgSigValue`: Extract the physical (unscaled) signal value from an 8-byte message
- `SBS_CAN_TxMsgSigValue`: Insert the physical signal value into an 8-byte message
- `SBS_CAN_TxMsgSigValueRaw`: Insert the raw (scaled) signal value into an 8-byte message

- `SBS_CAN_GetSigValue`: Receive a physical signal value from the corresponding message as received previously
- `SBS_CAN_SetSigValue`: Send a physical signal value with the next corresponding message
- `SBS_CAN_GetSigValueRaw`: Equivalent to `SBS_CAN_GetSigValue`, just with a raw signal value
- `SBS_CAN_SetSigValueRaw`: Equivalent to `SBS_CAN_SetSigValue`, just with a raw signal value

For handling signals and messages, raw and physical values, sending cyclic messages and messages on demand, SBS needs a CAN specification in form of a DBC file:

- `SBS_CONF_AddDBC`: Configures the CAN communication with
  - the CAN bus ID for distinguishing multiple CAN buses in Silver,
  - the DBC file defining the current network,
  - the node name that this Silver module enacts (possibly a number of node names if this Silver module simulates multiple CAN nodes, such as a rest bus simulation),
  - a number of flags to influence the behavior of the CAN functions, and
  - optionally the name of a file specifying a list of messages (not) to be sent and received (by message ID or message name)
- `SBS_CONF_AddCANMsg`: In case a DBC file is not available, cannot be published or tool support for DBCs is missing, SBS offers a function to add (and configure) one message at a time. Signal extraction is not supported in this case, only 8-byte messages can be received and sent.
- `SBS_CONF_SetCallbackDLL`: When changing the content of a message by inserting signal values, the project-specific CRC for this message will be invalid. Moreover, some messages require message counters to be incremented in a project-specific way whenever a message is being sent. `SBS_CONF_SetCallbackDLL` allows the specification of a DLL which contains a function called `updateMsg` to fix any signal inside the message (CRC and/or counters, ...). Other optional functions include
  - `checkMsg` to check for validity of a message, and
  - `allowTx` to ask if it is ok to send this message now.

This callback mechanism allows project-specific aspects of the CAN handling, not specified in the DBC file, to be handled correctly and hide IP, such as the project-specific CRC algorithm.

For easy debugging, SBS unpacks all sent messages and creates outputs for each of the signals contained in the messages. This makes debugging of signal values much simpler than manually decoding 8-byte message data. User flags are available to help SBS create unique Silver names.

Silver comes with several working examples to demonstrate how to use CAN messages and signals with SBS.

3.3 The Simulink Silver CAN block set

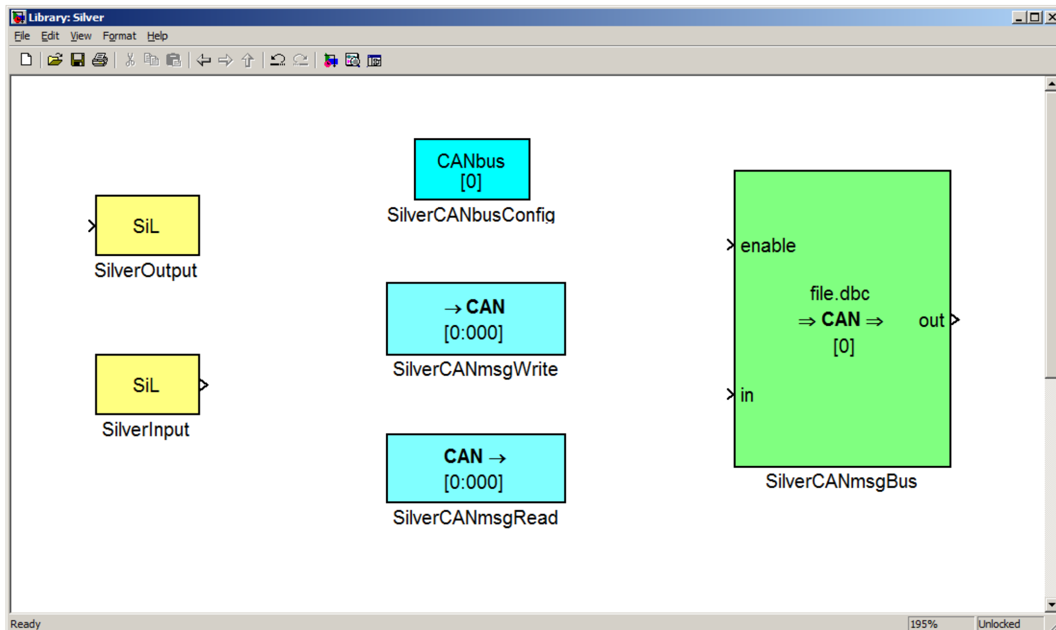

Figure 3: Silver Simulink library

Silver is supplied with a Simulink block set to access CAN messages (Figure 3). When exporting a Simulink model that contains such blocks, Simulink Coder turns these blocks into C code calling into the SBS functions introduced above. The following CAN-related blocks are available:

- `SilverCANbusConfig`: This block exists once per CAN bus and its parameters specify all the inputs similar to `SBS_CONF_AddDBC`.
- `SilverCANmsgRead`: This block has all the signals of a message as outputs. It has parameters that specify whether
    - the signals should be physical or raw values,
    - the receive time should be added as an output signal,
    - if the message should be checked for integrity (via callback, see above) and the result made available as an output signal as well.
- `SilverCANmsgWrite`: This block allows a specified message to be sent. All the signals to this message are generated as inputs to this block. The user can specify if the inputs contain raw or physical values.
- `SilverCANmsgBus`: This block builds up a whole CAN rest bus at once. It takes an input bus with all messages to be sent and provides an output bus with read messages. An enable bus allows node enabling/disabling. The block parameters are comparable with `SBS_Conf_AddDBC` which uses the DBC file, the bus ID, the active node name(s) and optionally an ignore file.

Besides building plant models in Simulink specifically for Silver SiL simulations, reuse of HiL (plant) models for SiL was the primary motivation for developing this block set. HiL models usually exist in sufficient quality for initial SiL tests to make them a good starting point when setting up SiLs. This block set makes it possible to replace CAN blocks that access physical CAN cards on HiL rigs with blocks that access the SiL CAN messages.

Note: Other changes might be necessary to adapt HiL models for SiL, such as hardware timer blocks [2].

3.4 DBC files to specify the behavior of a CAN node

Using DBC files to specify the behavior of a node in a (virtual) CAN network is convenient, reduces the chances of errors and permits fast introduction of updates, as DBC files are usually well maintained and tested by many engineers. The use of DBC files this early in the development process also helps to test and debug these DBC files.
Silver permits building modules with SBS where the DBC files can be specified at runtime, allowing the end user to incorporate newer versions of the DBC if necessary without rebuilding the module itself.
Changing the DBC file after coding may cause critical problems at runtime, because of the signal names used in the C code. If, for instance, a signal was renamed or removed from a message in the DBC file, the C code will still query its value and create a runtime error. A safe change is a change in how a signal is coded inside a message, as those details are hidden from the C code.

3.5 Accessing physical CAN networks from Silver

Silver offers a convenient way of accessing physical CAN networks (Figure 4). A specialized Silver CAN module lets Silver access a CAN network device and communicate through it with all the other devices on that physical CAN bus.
At the moment, this solution supports accessing Vector CAN cards and CAN USB. A hardware abstraction layer allows easy addition of other CAN hardware in the future.
The Silver CAN module is configured using a DBC file and the network node name Silver is enacted. The CAN module automatically generates its own input and output variables for Silver, sending and receiving messages on the CAN network according to the protocol specification of the DBC file. It is also possible to specify a callback DLL for updating CRCs and message counters just before they are sent to the physical CAN network (for details see above).
When using the Vector CAN driver, the CAN module can also be interfaced with the Vector virtual CAN bus. This permits interaction with a wide range of Vector tools, such as CANape® and CANoe®.
The CAN module is currently used quite extensively to build rapid control prototyping setups. A virtualized controller software runs in Silver and is attached through the CAN module with the physical ECU in the car; here it overwrites parts or the entire function software by communicating directly with the basic software of the ECU, setting actuators and reading sensor values using special CAN messages.

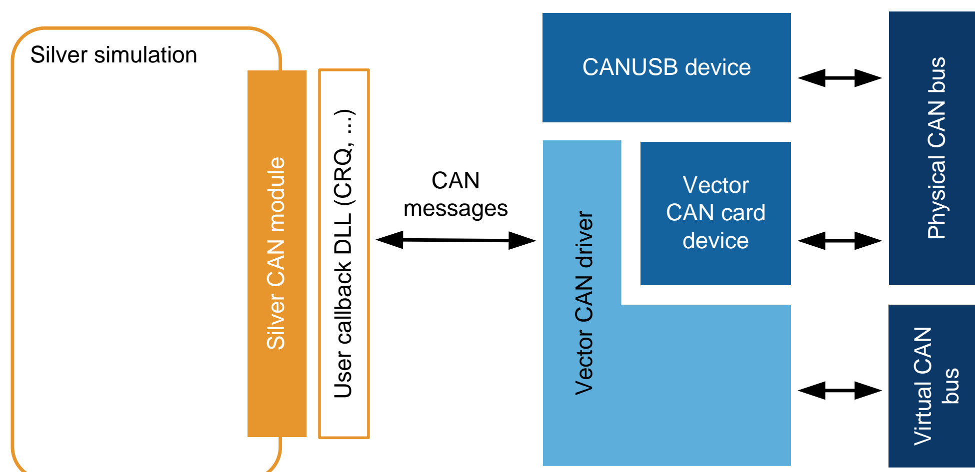

Figure 4: Accessing the physical CAN bus from Silver SiL

4. Example

IAV uses the newly developed CAN support to set up an improved SiL simulation of a transmission control software in mass production.

4.1. General setup

The setup consists in principle of a virtual ECU with control software and an environment model. The following diagram sketches this setup. For a detailed description see [3].

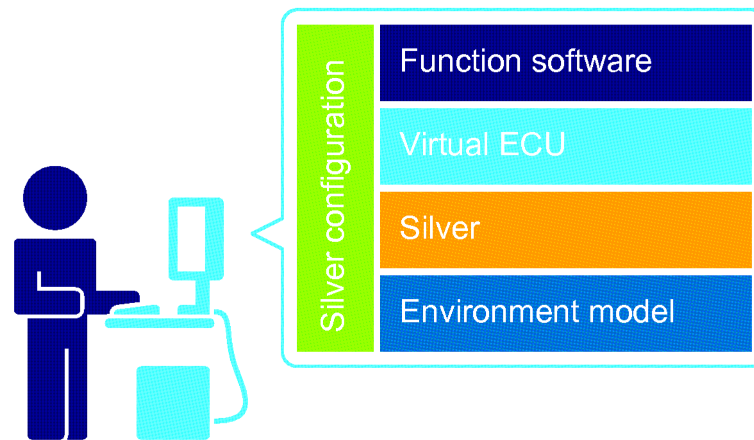

Figure 5: SiL setup for transmission control software

Data exchange between those main simulation components uses exactly the same interface definitions as in the car (CAN with more than 30 relevant messages, less than 20 input ports and less than 20 output ports). Thus the original car CAN definition (a DBC file) is used and the environment model serves as rest bus simulation for the ECU.

4.2. Implementation

The virtual CAN has to be implemented on the virtual ECU as well as in the environment model.

The ECU part implements the CAN interface directly via SBS functions from 3.2. These are specified with just a few lines of C code.
The following source code example shows SBS implementation of a CAN bus specified by a DBC file with a node. The layout (number of rx- and tx-messages) depends on the definition of the DBC “Transmission” node.

```
void SBS_USER_get_module_interface20 (void *sbs, int argc, char **argv) {
  // first comes the definition of tasks
  ...
  // second is the Silver input/ output variable definition
  ...
  // now we want to create a can bus using a dbc file, node and ignore file
  SBS_CONF_AddDBC (
    sbs,            // sbs handle
    1,              // can bus ID
    "trans.dbc",    // file name
    "Transmission",// node
    "ignore.txt",   // ignore file name
    0x0,            // flags, z.B. SBS_DBC_ENFORCE_RANGES
    0x0,            // channel mask
    NULL            // modified signals, e.g. message counter and CRCs
  );
  return DLL_OK;
}
```

The BIOS functions for interactions with the CAN drivers are mapped using corresponding SBS calls.

```
void BIOS_RX_Can_Message (Can_Message* msg) {
  int msg_handle = SBS_CAN_GetMsgHandle(sbs, SBS_IO_INPUT, 0, msg->id);
  // set message flag to not received
  msg->flag = SBS_CAN_STAT_RX_NOT_REC;
  // is handle valid?
  if (SBS_CAN_IsValidHandle(sbs, msg_handle) != 0) {
    // has a new message arrived?
    if (0 == SBS_CAN_RxMsgStatus(sbs, msg_handle)) {
      // receive can message
      SBS_CAN_RxMsg(sbs, msg_handle, msg->data, msg->size, &msg->flag);
    }
  }
}

void BIOS_TX_Can_Message (Can_Message* msg) {
  int msg_handle = SBS_CAN_GetMsgHandle(sbs, SBS_IO_OUTPUT, 0, msg->id);
  // initialise message flag to not transmitted
  msg->flag = SBS_CAN_STAT_TX_TRANS_NOT_OK;
  // is handle valid?
  if (SBS_CAN_IsValidHandle(sbs, msg_handle) != 0) {
    // ready to transmit?
    if (0 == SBS_CAN_TxMsgStatus(sbs, msg_handle)) {
      // transmit can message
      SBS_CAN_TxMsg(sbs, msg_handle, msg->data, msg->size, &msg->flag);
    }
  }
}
```

The environment model is branched from an existing Simulink model used as HiL plant model. The Silver CAN block set is used here to substitute blocks that access the physical hardware on the HiL with equivalent blocks that allow access to the Silver CAN infrastructure.

While one global block configures the CAN, one block generates (reads from CAN) or terminates (sends to CAN) the existing signals for each CAN message. These blocks are configured by simply specifying the CAN bus ID and the message ID or name in the block's parameter dialog. The timing is realized by Silver using the DBC information. See figure 6 for an example.

Checksum generation is enabled by specifying a DLL containing a generation function which is called for selected messages (by name) before sending.

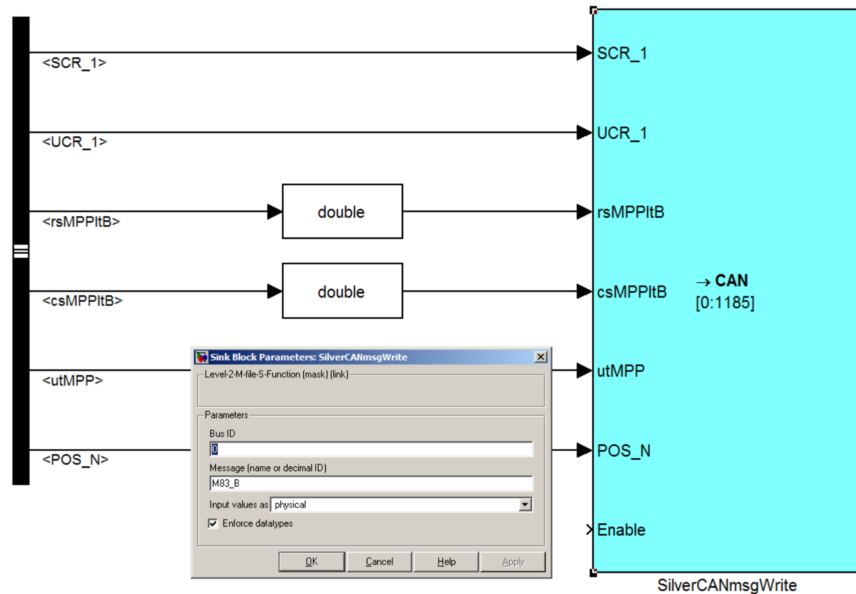

Figure 6: Silver CAN block in Simulink

The DBC file defines a large number of messages being sent on the CAN network, many of which are not part of the required CAN rest bus (Figure 7). A white list of CAN messages (by ID) is used to reduce this list of the DBC-defined messages to the CAN messages actually used in the SiL. A black-list approach is also possible but is less convenient here because of the number of signals to be excluded.
The SiL setup uses a Silver CAN module for mirroring the SiL-internal CAN bus to a Vector virtual CAN bus. This enables reading of all CAN messages by CANape in addition to the analysis inside Silver, where each CAN signal is accessible in the GUI.

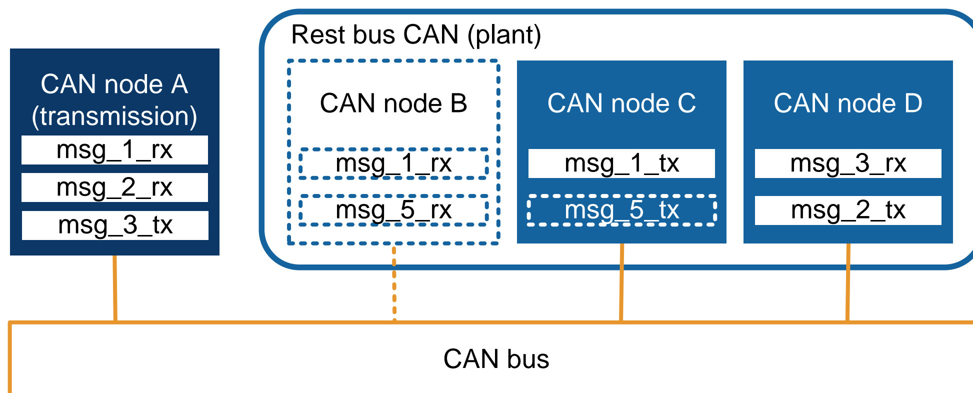

Figure 7: CAN rest bus in Silver simulation

4.3. Example summary

The described solution drastically simplifies the setup of the SiL system compared to previous, non-SBS-based setups. It reduces the amount of work for SiL signal mapping by using existing standard CAN definitions.
It makes the simulation even more realistic and lets users compare signal sequences directly with in-car behavior. Furthermore, CAN signal extraction and setting, as an important part of the function software, are now also used in the SiL simulation.
IAV engineers were instrumental in improving and validating the current version of SBS as well as the SBS-based CAN blocks for Simulink.

5. Summary and conclusions

Increasing pressure on engineering efficiency on the one hand and growing system complexity on the other hand makes it a logical step to virtualize systems of interconnected components. Network communication is an important part of such a system.
This paper describes how to connect controller software and Simulink models to a virtual CAN network in Silver using the Silver Basic Software. Silver is also used to connect such a virtual CAN network to a physical CAN network.
It shows how existing CAN description files of the DBC format can be used to reduce configuration effort while at the same time improving the consistency of definitions between different CAN modules.
Using Silver as transport layer makes it possible to test more parts of the CAN stack than previously possible, including CRC computations and message-counter algorithms – all in a virtual environment, before physical prototypes exist.

Literature

[1] W. Zimmermann, R. Schmidgall: Bussysteme in der Fahrzeugtechnik, 3. Auflage 2008, Vieweg + Teubner Verlag

[2] Liebezeit, Bräuer, Serway, Junghanns: Virtual ECUs for developing automotive transmission software, Presented at CTI Symposium Innovative Fahrzeug-Getriebe Hybrid- und Elektro-Antriebe, 5. - 8.12.2011, Berlin

[3] Liebezeit, Junghanns, Bonin, Serway: Silver Basic Software: Building Virtual ECUs Quickly and Economically, ATZ Electroink, 03/2012