COMPL$_e$T$_e$ - A *COMmunication Protocol vaLidation Toolchain* using Formal and Model-Based Specifications and Descriptions

Sven Gröning, Christopher Rosas, and Christian Wietfeld

Communication Networks Institute (CNI), TU Dortmund University,
Dortmund 44227, Germany,
Email: {sven.groening, christopher.rosas,
christian.wietfeld}@tu-dortmund.de,
home page: www.cni.tu-dortmund.de

Abstract. Because of shorter software development cycles for communication protocol stacks, the risk of design failures rises. Therefore, even within the protocol specification phase, appropriate validation should be performed in order to detect failures as early as possible.
In the light of electric vehicle integration in a smart grid environment, the complexity of charging processes increases e.g. for demand management, and thus also complexity of requirements for associated communication protocols increases. Accordingly, it lends to describe the behavior of communication protocols by abstraction in form of models. The use of model checking processes can validate properties of future behavior, hence failures may be detected earlier.
COMPL$_e$T$_e$ is a toolchain for validation of communication protocols, represented in an adapted version of UML-Statecharts. The toolchain uses the SPIN model checker and its composition is based on techniques of Model-Driven Software Development (MDSD). The applicability of this toolchain will be presented by modeling an exemplary communication protocol for electric vehicle charging.

Keywords: Communication Protocol Validation, COMPL$_e$T$_e$, SPIN, UML-Statecharts, Electric Mobility

1 Introduction

Communication protocols in general, define the way of information exchange between devices or other entities on a network. To reach an agreement by involved parties about the way of information flow, the protocol description should be developed as a technical standard. Some standards already include a formal description, however only in rare cases. Furthermore, the description of the protocol behavior may also have a high level of complexity.

Especially in the context of electric mobility, a future widespread use of electric vehicles requires the deployment of reliable, uniform and comprehensive battery charging infrastructures. Therefore, the communication between all systems becomes an important factor for future acceptance.

By use of model checking techniques, the behavior of new communication protocol standards can be validated within the specification process. For this purpose, it is required to describe the behavior in a formal description language, which can be used by state-of-the-art model checking tools like SPIN[5]. COMPL$_e$T$_e$ combines the possibility of an abstract behavior description represented as Unified Modeling Language (UML)-Statechart models, with a formal representation in PROMELA, which is used by SPIN as input language. Accordingly, COMPL$_e$T$_e$ facilitates the formal description process.

The remainder of this paper is structured as follows. In Section 2 an overview of the designed toolchain is given. The applicability of the proposed toolchain will be presented in Section 3 by modeling an exemplary communication protocol for electric vehicle charging. Section 4 closes with a conclusion and an outlook including future work.

2 Concept and Design of COMPL$_e$T$_e$

COMPL$_e$T$_e$ realizes a COMmunication Protocol vaLidation Toolchain, by using formal and model-based specifications and descriptions. The concept should take the following requirements into account.

First, the support for creation and modification of graphical models which represent communication protocols shall be developed. Moreover, an automatic transformation of the constructed graphical models to the input language of a corresponding model checker is needed. This transformation builds the link between the front-end and the back-end component in Figure 1. The back-end component shall integrate a model checker tool. Furthermore, support for editing the transformed models, based on the input language of the model checker is required. In addition, properties must be definable, so that models can be checked against. Second, beside the more functional requirements the toolchain shall be

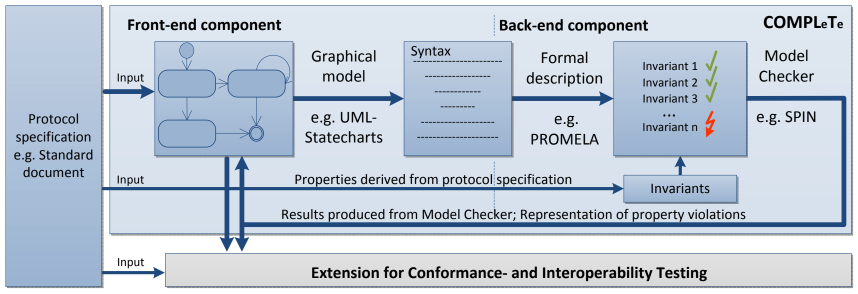

Protocol specification e.g. Standard document
Input
Front-end component
Back-end component
COMPLeTe
Graphical model
e.g. UML-Statecharts
Syntax
Formal description
e.g. PROMELA
Invariant 1
Invariant 2
Invariant 3
...
Invariant n
Model Checker
e.g. SPIN
Input
Properties derived from protocol specification
Invariants
Results produced from Model Checker; Representation of property violations
Input
Extension for Conformance- and Interoperability Testing

Fig. 1. Architecture of COMPL$_e$T$_e$

used within the Eclipse Integrated Development Environment (IDE). This will ensure that components within the toolchain can easily be exchanged or modified (modularity and extensibility) and new components can be integrated in a simple way. Furthermore, open-source or free available existing tools shall be used in order to consider reusability.

2.1 Realization of Front-End Component

The front-end component realization of COMPL$_e$T$_e$ utilizes a combination of approaches described in [3] and [7] to use UML-Statecharts for modeling communication protocols and SPIN as model checker for verification purposes. In [3] an automatic transformation from UML-Statecharts based on the domain-specific UML-Statecharts Description Language (UDL) to PROMELA source code is described. However, such models are only created in a textual form so that a graphical editor has to be created. In [7] a meta-model for UDL and PROMELA is constructed in order to define a Model-to-Model (M2M) transformation, which represents a homomorphic mapping between meta-model elements. Only a PROMELA meta-model has to be constructed, because an UDL meta-model has already been created in [3].

Figure 2 shows the development process (steps D1 - D5) which is grouped into the categories meta-modeling, transformation and modeling. These conform to the paradigm of Model-Driven Software Development (MDSD) and ensure modularity and extensibility of the toolchain. For this reason, the combination of the two approaches were chosen. The generation of meta-models for UDL and PROMELA is realized by use of Xtext and the Eclipse Modeling Framework (EMF) giving an Extended Backus-Naur Form (EBNF) grammar for UDL and PROMELA. The Model-to-Model (M2M) transformation from UDL to PROMELA model instances is provided by mapping rules between elements of the generated meta-models in the Atlas Transformation Language (ATL). Therefore the rules described in [3] are used as a basis for the appropriate mapping. The Model-to-Text (M2T) transformation from PROMELA model instances into PROMELA source code is achieved by the Xpand template language. The graphical editor for UDL respectively UML-Statecharts is generated via the Graphical Modeling Framework (GMF).

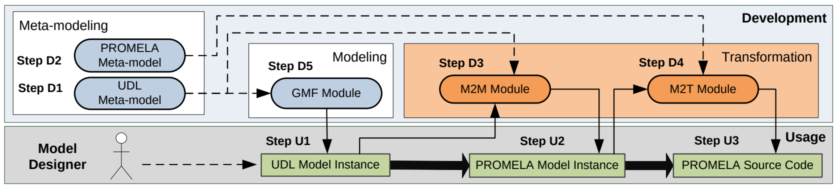


Fig. 2. Overview of development process in front-end component of COMPL$_e$T$_e$

With the COMPL$_e$T$_e$ front-end component, a model designer is able to create abstractions of communication protocols in form of UML-Statechart models which can be transformed into PROMELA models and subsequently into PROMELA source code. This is indicated in Figure 2 by steps U1-U3.

2.2 Realization of Back-End Component

The development of a back-end component comprises the integration of SPIN model checker. Figure 3 summarizes the implemented functionality of COMPL$_e$T$_e$. The bottom layer shows the prerequisites and basic functions for the usage of

SPIN. These are also partly described and supported by similar approaches in [4] and [6]. In addition, several extensions are implemented in COMPL$_e$T$_e$ which build on top of these basic functions. As an example the invocation of an interactive and interactive-random simulation can be conducted enabling user-interaction. Furthermore, a *MSC-View* is included to allow visualization of the communication flow between PROMELA processes during simulations. This view is complemented with a *SimData-View* which shows variable values and queues of the PROMELA model. For verification purposes a specific *LTLProperty-View* is built to simplify the user interface. In addition it provides support of a so called *Multi-verification*. Thereby, for a given PROMELA model it can be invoked on a number of selected invariants. In case of a violation the *Multi-verification* is terminated.

Another extension is the *Automata-View*, in which the finite state machines of a corresponding PROMELA model are displayed by use of the Zest/DOT tooling. The *FSMSimulation* represents a combination of the *Automata-View* and a simulation. This allows to display a complete simulation-path by highlighting the visited states, from the beginning up to the occurrence of the invariant violation.

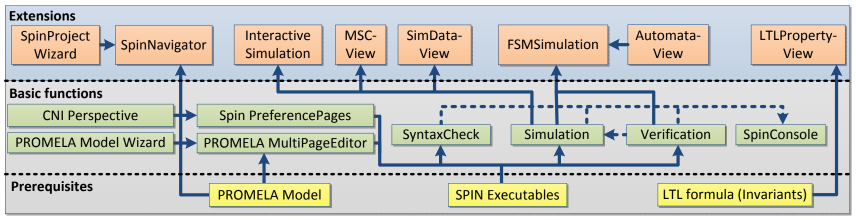


Fig. 3. Functions of back-end component of COMPL$_e$T$_e$

3 Application of COMPL$_e$T$_e$ in Electric Mobility Context

The applicability of COMPL$_e$T$_e$ is presented by modeling an exemplary communication protocol for electric vehicle charging. One representative is the *Smart Charge Communication Protocol Specification (SCCPS)* [1], which is the predecessor for the standardization process of *ISO/IEC 15118* [2]. The binding process is a self-contained part of SCCPS and therefore an ideal candidate to be applied to COMPL$_e$T$_e$. In general the binding process ensures a successful setup of a point-to-point connection between a Charge Point (CP) and an Electric Vehicle (EV) on IP-Level. In a first step a simplified SCCPS binding process is modeled in UDL respectively as UML-Statecharts, using the front-end component of COMPL$_e$T$_e$. Subsequently an automatic invocation of the translation process takes place, which produces corresponding PROMELA source code from the constructed UDL models. The second step uses the PROMELA source code to process the verification of the SCCPS binding. For this purpose the following example properties are defined in the *LTLProperty-View*:

– Eventually the binding process will be completed. $\diamond\, bindingComplete$
– The connection of EV and CP always implies the completion of the binding process (invariant property). $\square\,(instancesConnected \implies bindingComplete)$

Especially the invocation of verifications is a crucial point in the evaluation of $\mathrm{COMPL_eT_e}$'s capabilities. The verification for the first defined property succeeds without any errors but the second property fails. For analysis purposes, a guided simulation is conducted. Within $\mathrm{COMPL_eT_e}$ the message flow is visualized in the *MSC-View*. In addition, the *FSMSimulation* shows the traversing of the states, which are iteratively highlighted in the *Automata-View*. By analyzing the SCCPS model using the *MSC-View* and the *FSMSimulation*, the failed verification results in a loss of a message during the SCCPS binding process. Thus, it is necessary to adapt the simplified UDL model in order to correct the error.

4 Conclusion and Future Work

In this paper the concept and realization of $\mathrm{COMPL_eT_e}$ was introduced, which enables validation of communication protocols. The toolchain can be applied in the context of electric mobility as well as in further domains of interest. The communication protocol behavior is represented as a graphical UDL model. A conversion into an equivalent PROMELA code is accomplished via a M2M and a M2T transformation. For verification purposes, $\mathrm{COMPL_eT_e}$ integrates the SPIN model checker and allows simplified usage by enabling mechanisms to specify invariant properties as LTL formulas and additional analysis extensions.

For future work, the construction of a "complete" model of SCCPS as well as the upcoming ISO/IEC 15118 [2] standard is intended, in order to validate their related protocol behavior. Furthermore interoperability and conformance testing capabilities of $\mathrm{COMPL_eT_e}$ shall be considered.

Acknowledgment The work in this paper was funded by the *NRW Ziel 2* Program 2007-2013 (EFRE) of the European Union, the MBWSV and the MKULNV of NRW as part of the TIE-IN project with reference number 64.65.69-EM-1022A.

References

1. Project e-Mobility: Smart Charge Communication Protocol Specification Part A & B. Tech. rep., RWE, Daimler, INSYS Microelectronics, EMSYCON (2010)
2. ISO/IEC DIS 15118, Road vehicles - Vehicle to grid Communication Interface (2012)
3. Ammann, C.: Verifikation von UML-Statecharts unter besonderer Berücksichtigung von Speicherverbrauch und Laufzeit des Model Checkers [Verification of UML-Statecharts with particular attention of memory usage and runtime of the model checker]. Softwaretechnik-Trends 31(3) (2011)
4. De Vos, B., Kats, L.C.L., Pronk, C.: EpiSpin: An Eclipse Plug-in for Promela/SPIN using Spoofax. In: Proceedings of the 18th international SPIN conference on Model checking software. pp. 177–182. Springer-Verlag, Berlin, Heidelberg (2011)
5. Holzmann, G.J.: The model checker SPIN 23(5), 279–295 (1997)
6. Kovše, T., Vlaovič, B., Vreže, A., Brezočnik, Z.: Eclipse Plug-In for SPIN and st2msc Tools-Tool Presentation. In: Proceedings of the 16th International SPIN Workshop on Model Checking Software. pp. 143–147. Springer-Verlag, Berlin, Heidelberg (2009)
7. Mcumber, W.E., Cheng, B.H.: A general framework for formalizing UML with formal languages. In: Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on. pp. 433–442 (2001)

A Oral Tool Presentation

A.1 Structure of the Presentation

This section gives a structure of the demonstration of $\mathrm{COMPL_eT_e}$ and is divided into the following three parts.

1. Motivation for designing $\mathrm{COMPL_eT_e}$
   - $\mathrm{COMPL_eT_e}$ in the electric mobility context
   - Protocol validation including verification (SPIN/PROMELA) and testing
2. Detailed development of $\mathrm{COMPL_eT_e}$:
   - Front-end: Model design and transformation
   - Back-end: SPIN integration and extensions
3. Case study: Application of SCCPS binding process in $\mathrm{COMPL_eT_e}$
   - Description of SCCPS and its binding process
   - Front-end component: Presenting UDL model and invoke automatic transformation into PROMELA source code
   - Back-end component: Demonstrate several functions of back-end component $\rightarrow$ *PromelaEditor, MSC-View, SimData-View, FSMSimulation, Automata-View, LTLProperty-View, SyntaxCheck*, Various simulation runs (random, guided, interactive simulation runs), Verification and *Multi-Verification* runs, Definition of invariants with regard to the generated PROMELA source code of the SC-CPS binding process and at last show analysis capabilities of $\mathrm{COMPL_eT_e}$ by explaining the failed verification

A.2 Front-end: Model Design and Transformation

In the meta-modeling steps D1 and D2 (see Figure 2) appropriate meta-models on basis of an EBNF grammar, Xtext and EMF for both UDL and PROMELA are generated. The Listing 1.1 offers an excerpt of UDL grammar rules for creation of a corresponding UDL meta-model. Listing 1.2 shows an excerpt of the PROMELA grammar representing a rule for a PROMELA *proctype*.

```
Model:
  (imports+=UDLInclude)*
  (variable+=UDLData)*
  (behaviour=UDLBehaviour)? ;

//State Rules
UDLState:
  UDLSimpleState |
  UDLCompositeState |
  UDLFinalState |
  UDLInitialState ;

UDLSimpleState:
  "simplestate" name=ID
  "{"
    (entry=UDLEntryAction)?
    (exit=UDLExitAction)?
    (out+=UDLTransition)+
  "}";
```

Listing 1.1. Excerpt of UDL grammar

```
spec: // PARSER RULES
  (specname=ID)?
  (modules += module+);

module:
    proctype    /* proctype declaration */
  | init      /* init process        - max 1 per model */
  | never     /* never claim         - max 1 per model */
  | trace     /* event trace         - max 1 per model */
  | utype     /* user defined types */
  | mtype     /* mtype declaration  */
  | decl_lst    /* global vars, chans */
  | inline
  | preprocess ;

proctype:
  (active=active)?
  PROCTYPELABEL name=ID PARENOPEN
  (dlist=decl_lst)?
  PARENCLOSE
  (priority=priority)?
  (enabler=enabler)?
  BLOCKBEGIN
  seq=sequence
  BLOCKEND
  (SEMICOLON)* ;
```

Listing 1.2. Excerpt of PROMELA grammar

Step D3 indicates the M2M transformation from UDL to PROMELA model instances via ATL. Listing 1.3 gives an exemplary M2M mapping rule in ATL describing the translation of UDL Enumerations into PROMELA *mtypes*. The M2T transformation from PROMELA model instances to PROMELA source code via Xpand templates is conducted in step D4. Listing 1.4 offers an excerpt of a Xpand template rule which shows the transformation of a PROMELA *proctype* element into its corresponding code fragment.

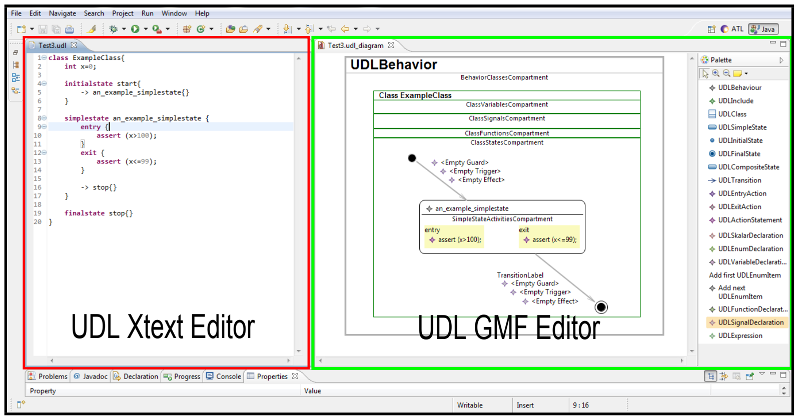
```
-- @path UDLMM=/com.statechartverification/src-gen/com/statechartverification/UDL.ecore
-- @path PMLMM=/org.xtext.draft.promela/src-gen/org/xtext/draft/promela/PromelaDSL.ecore

module UDL2PML;
create OUT: PMLMM from IN: UDLMM;

-- Transform UDLEnumDeclare into Promela mtype declaration
rule UDLEnumDeclare2Promela {
    from
        udl_enum_declare_in: UDLMM!UDLEnumDeclare
    to
        pml_out: PMLMM!mtype (
            name <- udl_enum_declare_in.getFirstEnumElementName(),
            name <- udl_enum_declare_in.next
                ->  collect(e |
                    udl_enum_declare_in.getEnumDeclareNamePrefix() + e.name)
    )
}
```

Listing 1.3. Excerpt of ATL transformation file

```
«IMPORT promelaDSL»

«DEFINE generateSpec FOR spec»
  «IF this.specname != null»
    «FILE this.specname+".promela"»
      «EXPAND generateModules FOR this-»
    «ENDFILE»
  «ELSE»
    «FILE "Test.promela"»
      «EXPAND generateModules FOR this»
    «ENDFILE»
  «ENDIF»
«ENDDEFINE»

«DEFINE generateModule FOR proctype»
  //Proctype ModuleDeclaration
  «IF this.active != null»
    «EXPAND generateActive FOR this.active»
  «ENDIF-»proctype «this.name-»(
    «IF this.dlist != null»
      «EXPAND generateDeclarationList FOR this.dlist-»
    «ENDIF»)
    «IF this.priority != null»
      «EXPAND generatePriority FOR this.priority»
    «ENDIF-»
    «IF this.enabler != null»
      «EXPAND generateEnabler FOR this.enabler»
    «ENDIF-» {
      «IF this.seq != null»
        «EXPAND generateSequence FOR this.seq»
      «ENDIF»
}
«ENDDEFINE»
```

Listing 1.4. Excerpt of Xpand template file representing the M2T transformation

Step D5 describes the development of a graphical UDL Editor in Eclipse via GMF. Figure 4 shows the constructed graphical UDL Editor and the corresponding textual UDL Editor from [3].

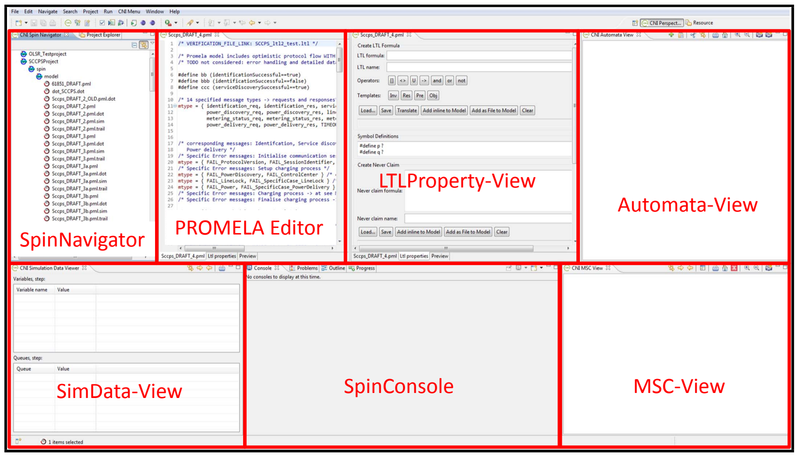


```
class ExampleClass{
    int x=0;

    initialstate start{
        -> an_example_simplestate{}
    }

    simplestate an_example_simplestate {
        entry {
            assert (x>100);
        }
        exit {
            assert (x<=99);
        }

        -> stop{}
    }

    finalstate stop{}
}
```

Fig. 4. Comparison textual UDL Editor and generated graphical UDL GMF Editor

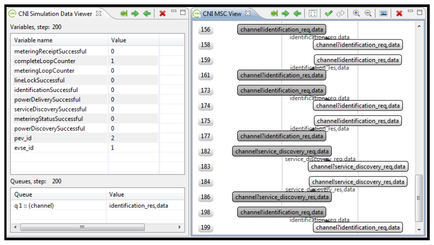
A.3 Back-end: SPIN Integration and Extensions

The result of SPIN integration in COMPL$_e$T$_e$ as Eclipse Plugin is shown in Figure 5. With regard to simulation and verification capabilities Figure 6 show the *MSC-View*, *SimData-View* and *Automata-View*.



```
/* VERIFICATION_FILE_LINK: SCCPS_ltl2_test.ltl */

/* Promela model includes optimistic protocol flow WITH
/* TODO not considered: error handling and detailed dat

#define bb (identificationSuccessful==true)
#define bbb (identificationSuccessful==false)
#define ccc (serviceDiscoverySuccessful==true)

/* 14 specified message types -> requests and responses
mtype = { identification_req, identification_res, servi
          power_discovery_req, power_discovery_res, lin
          metering_status_req, metering_status_res, met
          power_delivery_req, power_delivery_res, TIMEO


/* corresponding messages: Identifcation, Service disco
   Power delivery */
/* Specific Error messages: Initialise communication se
mtype = { FAIL_ProtocolVersion, FAIL_SessionIdentifier,
/* Specific Error messages: Setup charging process */
mtype = { FAIL_PowerDiscovery, FAIL_ControlCenter } /*
mtype = { FAIL_LineLock, FAIL_SpecificCase_LineLock } /
mtype = { FAIL_Power, FAIL_SpecificCase_PowerDelivery }
/* Specific Error messages: Charging process -> at see
/* Specific Error messages: Finalise charging process -
```

Fig. 5. Overview of the Spin Eclipse Plugin within COMPL$_e$T$_e$



CNI Simulation Data Viewer

Variables, step: 200

Variable name	Value
meteringReceiptSuccessful	0
completeLoopCounter	1
meteringLoopCounter	0
lineLockSuccessful	0
identificationSuccessful	0
powerDeliverySuccessful	0
serviceDiscoverySuccessful	0
meteringStatusSuccessful	0
powerDiscoverySuccessful	0
pev_id	2
evse_id	1

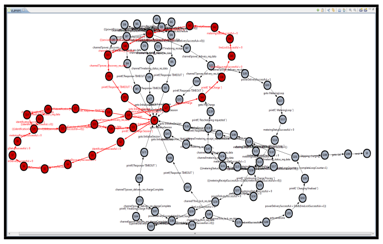
Queues, step: 200

Queue	Value
q 1 :: (channel)	identification_res,data

CNI MSC View

```
156  channel!identification_req,data
         identification_req,data
158                          channel?identification_req,data
159                          channel!identification_res,data
         identification_res,data
161  channel?identification_res,data
173  channel!identification_req,data
         identification_req,data
174                          channel?identification_req,data
175                          channel!identification_res,data
         identification_res,data
177  channel?identification_res,data
182  channel!service_discovery_req,data
         service_discovery_req,data
183                          channel?service_discovery_req,data
184                          channel!service_discovery_res,data
         service_discovery_res,data
186  channel?service_discovery_res,data
198  channel!identification_req,data
         identification_req,data
199                          channel?identification_req,data
```

(a) SimData-View and MSC-View



(b) Automata-View with FSMSimulation

Fig. 6. Focus on SimData-View, MSC-View and Automata-View with FSMSimulation

A.4 Case Study: Application of SCCPS Binding Process in COMPL$_e$T$_e$

The applicability of COMPL$_e$T$_e$ is demonstrated on the SCCSP communication protocol for electric vehicle charging. The sequence of the SCCPS binding process between a Charge Point and an EV is explained in Figure 7. At first the front-end component is used in order to model the SCCPS binding process and to transform the models into executable PROMELA source code. Afterwards the source code is applied to SPIN model checker via the back-end component of COMPL$_e$T$_e$.

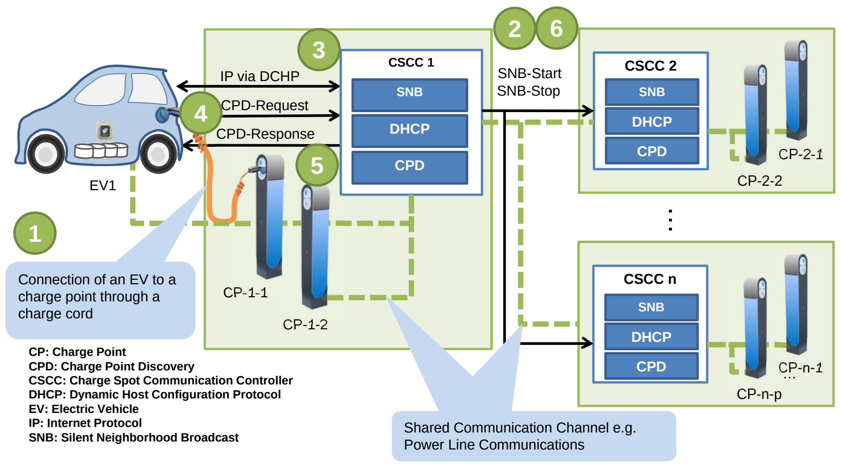
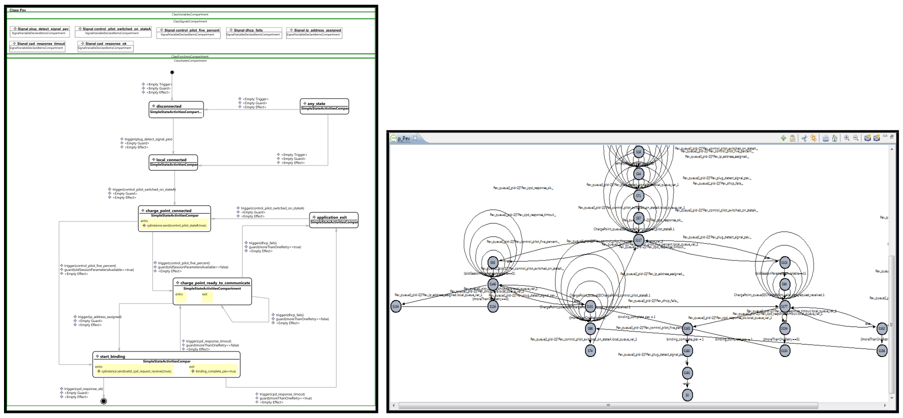

1
2
6
3
4
5
IP via DCHP
CPD-Request
CPD-Response
SNB-Start
SNB-Stop
CSCC 1
SNB
DHCP
CPD
CSCC 2
SNB
DHCP
CPD
CSCC n
SNB
DHCP
CPD
EV1
CP-1-1
CP-1-2
CP-2-1
CP-2-2
CP-n-1
CP-n-p
Connection of an EV to a charge point through a charge cord
Shared Communication Channel e.g. Power Line Communications
CP: Charge Point
CPD: Charge Point Discovery
CSCC: Charge Spot Communication Controller
DHCP: Dynamic Host Configuration Protocol
EV: Electric Vehicle
IP: Internet Protocol
SNB: Silent Neighborhood Broadcast

Fig. 7. SCCPS binding process

Front-end Component: Presenting UDL Model and invoke Automatic Transformation into PROMELA Source Code On basis of the described SCCPS specification the front-end component of COMPL$_e$T$_e$ is used to model the SCCPS binding process resulting in UDL models for the Charge Point and the EV. The EV model is illustrated in Figure 8. Afterwards an automatic M2M and M2T transformation takes place. Listing 1.5 shows an excerpt of the generated PROMELA source code for the SCCPS binding process.

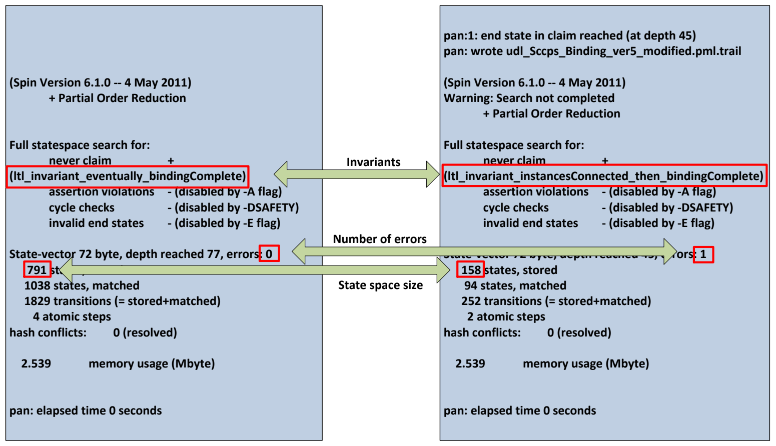
Back-end Component: Demonstrate Several Functions of Back-end Component The generated PROMELA source code of the SCCPS binding process is taken as input for SPIN. Therefore the invocation of a *Syntax Check* as well as several simulation runs are presented. The verification of the SCCPS binding process in PROMELA is demonstrated by use of defined LTL formulas from Section 3. For illustration purposes Figure 8 depicts the statemachine of the EV in the *Automata-View*.

The verification on the first defined property succeeds without any errors but the verification on the second property fails. Figure 9 shows an contrasting comparison of the generated SPIN outputs. An example of the *FSMSimulation* for the Charge Point and the EV is shown in Figure 10.



(a) UDL-Statechart of EV (b) EV statemachine in Automata-View

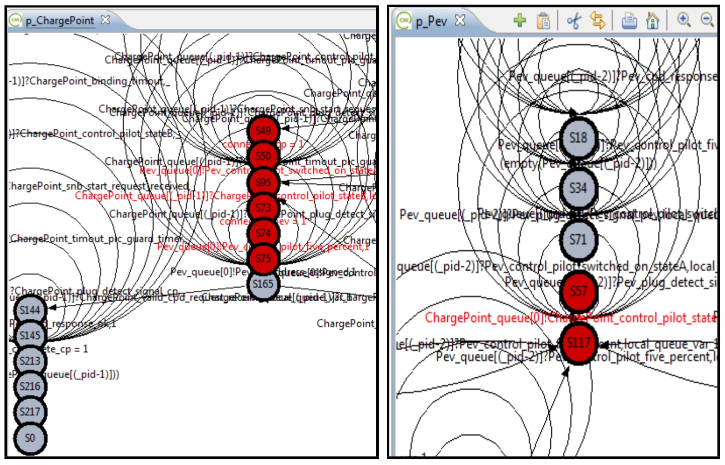
Fig. 8. EV models



```
(Spin Version 6.1.0 -- 4 May 2011)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
(ltl_invariant_eventually_bindingComplete)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by -E flag)

State-vector 72 byte, depth reached 77, errors: 0
    791 st
    1038 states, matched
    1829 transitions (= stored+matched)
      4 atomic steps
hash conflicts:         0 (resolved)

    2.539       memory usage (Mbyte)


pan: elapsed time 0 seconds
```

```
pan:1: end state in claim reached (at depth 45)
pan: wrote udl_Sccps_Binding_ver5_modified.pml.trail

(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             +
(ltl_invariant_instancesConnected_then_bindingComplete)
        assertion violations    - (disabled by -A flag)
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      - (disabled by -E flag)

State-vector 72 byte, depth reached 45, errors: 1
    158 states, stored
     94 states, matched
    252 transitions (= stored+matched)
      2 atomic steps
hash conflicts:         0 (resolved)

    2.539       memory usage (Mbyte)


pan: elapsed time 0 seconds
```

Fig. 9. Comparison of SPIN output for a successful and failed verification run



(a) CP statemachine (b) EV statemachine

Fig. 10. Paths of EV and Charge Point statemachines for the error case

```
//Mtype ModuleDeclaration
mtype = {ChargePoint_plug_detect_signal_cp, ChargePoint_timout_plc_guard_timer,
       ChargePoint_snb_start_request_received, ChargePoint_control_pilot_stateB, ChargePoint_binding_timout,
       ChargePoint_valid_cpd_request_received, ChargePoint_snb_stop_request_received,
       ChargePoint_timout_wait_binding_timer, ChargePoint_timout, ChargePoint_pilot_off_timout,
       Pev_plug_detect_signal_pev, Pev_control_pilot_switched_on_stateA, Pev_control_pilot_five_percent,
       Pev_dhcp_fails, Pev_ip_address_assigned, Pev_cpd_response_timout, Pev_cpd_response_ok,
       Environment_env_signal }
//Decl_lst ModuleDeclaration
bool oldSessionParametersAvailable;
bool moreThanOneRetry;
bool binding_complete_cp;
bool binding_complete_pev;
bool connected_cp;
bool connected_pev;
chan ChargePoint_queue[1] = [2] of {mtype, bool};
chan Pev_queue[1] = [2] of {mtype, bool};
chan Environment_queue[1] = [2] of {mtype, bool};
//MacroDeclaration
#define ChargePoint_queue_access(x)(x-1)
//MacroDeclaration
#define Pev_queue_access(x)(x-2)
//MacroDeclaration
#define Environment_queue_access(x)(x-3)

//Proctype ModuleDeclaration
proctype ChargePoint(){
goto start;
//STMNT Labeled Statement
start:
if
::
empty(ChargePoint_queue[ChargePoint_queue_access(_pid)]);
goto disconnected
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_plug_detect_signal_cp, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_timout_plc_guard_timer, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_snb_start_request_received, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_control_pilot_stateB, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_binding_timout, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_valid_cpd_request_received, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_snb_stop_request_received, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_timout_wait_binding_timer, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_timout, _ ;
goto start
::
ChargePoint_queue[ChargePoint_queue_access(_pid)]?ChargePoint_pilot_off_timout, _ ;
goto start
fi;
}

//Proctype ModuleDeclaration
proctype Pev(){
goto start;
//STMNT Labeled Statement
start:
if
::
empty(Pev_queue[Pev_queue_access(_pid)]) ;
goto disconnected
::
Pev_queue[Pev_queue_access(_pid)]?Pev_plug_detect_signal_pev, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_control_pilot_switched_on_stateA, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_control_pilot_five_percent, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_dhcp_fails, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_ip_address_assigned, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_cpd_response_timout, _ ;
goto start
::
Pev_queue[Pev_queue_access(_pid)]?Pev_cpd_response_ok, _ ;
goto start
fi;
}
```

Listing 1.5. Generated PROMELA source code of the SCCPS binding process