# A Run-Time Verification Framework for Smart Grid Applications Implemented on Simulation Frameworks

Selim Ciraci
Pacific Northwest
National Laboratory
Richland, WA, USA
selim.ciraci@pnnl.gov

Hasan Sözer
Özyeğin
University
Istanbul, Turkey
hasan.sozer@ozyegin.edu.tr

Bedir Tekinerdogan
Bilkent
University
Ankara, Turkey
bedir@cs.bilkent.edu.tr

***Abstract*—Smart grid applications are implemented and tested with simulation frameworks as the developers usually do not have access to large sensor networks to be used as a test bed. The developers are forced to map the implementation onto these frameworks which results in a deviation between the architecture and the code. On its turn this deviation makes it hard to verify behavioral constraints that are described at the architectural level. We have developed the ConArch toolset to support the automated verification of architecture-level behavioral constraints. A key feature of ConArch is programmable mapping for architecture to the implementation. Here, developers implement queries to identify the points in the target program that correspond to architectural interactions. ConArch generates runtime observers that monitor the flow of execution between these points and verifies whether this flow conforms to the behavioral constraints. We illustrate how the programmable mappings can be exploited for verifying behavioral constraints of a smart grid application that is implemented with two simulation frameworks.**

## I. Introduction

Smart grid combines the electric grid with communication and information systems to provide automated and efficient control of grid operations. These grids include millions of sensor and control nodes that collect information about the status of the grid (e.g., supply/demand) and adjust the grid operations [1]. Applications running on these nodes are required to efficiently process the collected information and control the grid operations accordingly. These applications also have to be reliable and dependable due to the mission critical nature of the grid.

In general, mission critical systems like smart grid applications are subject to behavioral constraints. Most of these constraints are systematic and likewise defined at the architecture deign level as sequences of interactions between components/connectors. To verify these behavioral constraints the software architecture needs to be evaluated [2], [3], and the implementation should be tested with respect to the defined behavioral constraints.

In the literature, various approaches have been introduced to verify the implementation with respect to the behavioral constraints defined at the architecture level [3], [4], [5]. An often used approach is runtime verification. Hereby, the behavioral constraints defined for the architecture are mapped to the implementation. Then, **runtime observers** are generated and integrated in the software system. At runtime, these observers log all the events that are defined as part of the documented scenarios. The verification tools provided with the approaches check the consistency of the collected logs with respect to the documented scenarios and report mismatches to the user.

Usually, the required vast amount of grid control units and/or sensor networks are not available to test smart grid applications. As such, these applications are often implemented and tested with simulation frameworks such as network and power distribution simulators [6]. However, such implementations deviate from the original software architecture of the application as they have to the follow the structure and/or the architecture of the simulation frameworks. This makes it challenging to map the constraints defined for the architecture to the implementation and verify these behavioral constraints.

In general, the proposed approaches from the literature require the implementation of the software system to follow strict rules. For instance, they require an interaction $I$ between components $A$ and $B$ to be implemented as a call from class $A$ to class $B$'s method $I$. Such rules are required in order to map the documented architecture to the code and generate runtime observers. Unfortunately, implementations with simulation frameworks generally do not follow these rules as the interactions between components/connectios are facilitated with the simulation framework. Therefore the proposed approaches are less feasible to check behavioral constraints in this context.

In contrast to existing approaches, Conarch framework [7] is designed to provide programmable architecture to code mapping for verifying behavioral constraints with respect to the implementation. Here, developers implement mapping queries that ConArch uses to identify the points in the target program that correspond to the interactions. ConArch automatically generates runtime observers that monitor the flow of execution between these points.

978-1-4673-6280-1/13/$31.00 © 2013 IEEE

SE4SG 2013, San Francisco, CA, USA

Unlike most of the other approaches, ConArch performs online monitoring and verification. At runtime, the observers report the flow of execution to ConArch's runtime verifier. The verifier determines whether the flow matches to the documented behavioral constraints.

The programmable mapping is the key feature of ConArch for verifying whether implementations with simulation frameworks follow the behavioral constraints defined for the architecture. As with this feature, developers can map interactions to the communication mechanisms provided by the simulation framework. In this paper, we show that ConArch is applicable and effective for such implementations due to the flexibility of programmable mapping. We illustrate it's applicability on a smart grid application implemented with two simulation frameworks.

We have also studied various power and communication network simulation frameworks (the necessary simulation frameworks when testing smart grid applications) and identified common styles for implementing interactions of the behavioral scenarios in the simulation frameworks. We implemented these styles as mapping queries and extended ConArch with a library containing these rules. This library can be utilized by developers to facilitate mapping of behavioral models to implementation in a language-independent way.

This paper is organized as follows: The following section provides background on smart grids and simulation frameworks. In Section III, we introduce a smart grid application that is implemented with simulation frameworks to motivate the need for conformance checking. This application is also used as a running example throughout the paper. Section IV describes the ConArch framework. In section V, we present how ConArch is used for the smart grid application. Section VI presents the related work and, finally, Section VII provides the conclusions.

## II. Smart Grids and Simulation

Smart grid is the application of data processing and communications to the power grid. Millions of sensors gather information about the demand from the consumer appliances and the supply in stations. This information is used by controllers to adjust the supply or the demand at the consumer appliances. Due to mission critical nature of the grid, smart grid applications, such as sensor and controller software, needs to be rigourously tested before deployment. Usually, developers do not have an access to a large smart grid setup to be used as a test bed. As such, simulators are used to test these applications.

Simulation of the smart grid is generally realized by combining the power system and the computer network simulators [6]. Developers extend the power simulator to implement the sensor and controller software of the application to be tested. The network simulator, on the other hand, is extended with a middleware for sending/receiving messages from the power simulator. Figure 1 presents the general co-simulation setup for smart grids.

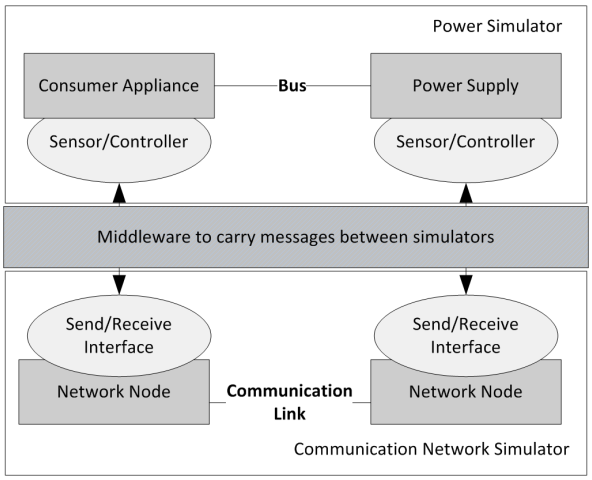


Fig. 1. Simulation setup for smart grid applications.

Power system simulators provide modules representing various power components, such as transformers and air conditioning units. Using these modules the user implements the distribution network model that will be simulated. This model consists of power components, buses attached to these components, and initial voltage/current values.

Tick-based simulation is generally used for simulating the complex dynamics of the power system. Here, the simulator sends tick messages to the modules, which means that time has progressed a certain amount. The modules, in turn, calculate new voltage/current values. The modules also let the simulator know the time they expect a change in power demand. For example, a module representing an air conditioner lets the simulator know when it will turn off. The simulator gathers this information and calculates next minimum time $t$ that the demand will change. It, then, sends a new tick message where $\triangle t = t - t_{before}$. The simulation continues until none of the components report a demand change.

Discrete event simulation is used for simulating the computer networks. Network simulators provide modules representing various network components, such as transfer and link layer protocols. The user implements the network model to be simulated using these modules. A network model consists of nodes, applications running at these nodes, and the links between the nodes.

During the simulation, the applications at the nodes generate packets to be transferred across the network and pass the packet to the network stack. The network stack, in turn, schedules events for discovering the route and transferring the packets to their desired destinations. These events are scheduled with respect to the time it would take the links to deliver the message to the next hop. The simulator dispatches these events, which simulates the routing of the messages.

## III. Motivating Example: Smart Grid Market Bid Application

The market bid application is used for finding the cleared (equilibrium) price of the electricity. A typical execution of this application is as follows: control nodes (controllers) at homes monitor the power demands of the air conditioning units. Depending on this demand, they make a bid for the price of electricity. The auction house, a sensor node attached to the power supply, collects the bids from the control nodes and calculates the cleared price for the electricity. It broadcasts the cleared price to the control nodes, which in turn adjust the thermostat settings on the air conditioning units to match the price.

The execution scenario above gets executed every 5 minute intervals where the auction house recalculates the cleared price with the updated demand. This scenario is a behavioral constraint that the sensor and controller nodes need to follow in order to successfully calculate a cleared price. Usually, constraints like this are defined during the architecture design and it is tested whether the software architecture supports these constraints. It is also important to verify whether the implementation follows such constraints. For the market bid application, deviation in the implementation from the constraint may result overcharging the customer or the air conditioners operating with incorrect settings.

For a 'normal' implementation (i.e., an implementation without the simulators), one could use existing approaches to check the conformance of the market bid application with respect to it's behavioral constraint during the testing phase. However, a large controller network that supports a variety of communication technologies (e.g., LTE, WiMAX, or etc.) is not available for testing. Hence, the designers decided to test the application using simulators and implemented it using simulation frameworks, specifically using Gridlab-D [8] and ns-3 [9] simulators.

Unfortunately, the implementation of the market bid application using the simulation frameworks is different from the planned architecture. The components controller and auction house are implemented as 4 classes, 2 for component controller and 2 for component market, residing in different modules of the simulation framework . This makes it hard to assess whether the implementation follows the behavioral constraints. Violation of the behavioral constraint might lead to incorrect results in cleared price calculations. Because the simulators tend to generate large output files, it might be hard for the designers to capture that the incorrect results are due to the mismatch between the implementation and the expected behavior. Hence, they might assume the error is caused by some other bug in the implementation and spend unnecessary time in searching for this bug.

The above mentioned problem can be generally observed in smart grid applications that are mapped to simulation frameworks. Due to the different structure of the architecture and the simulation framework the implementation will deviate easily from the architecture.

## IV. Overview of ConArch Approach

Figure 2 depicts the typical usage of ConArch framework for checking the conformance between the implementation and the behavioral constraints. Here, the inputs are *i)* Component - Connector model of the software system, *ii)* the behavioral constraints modeled as UML 2.x sequence diagrams, *iii)* the source files, and *iv)* the mapping of behavioral models to the source code. With this input, ConArch generates the **runtime verification specification** and the source code instrumented with **runtime observers**.

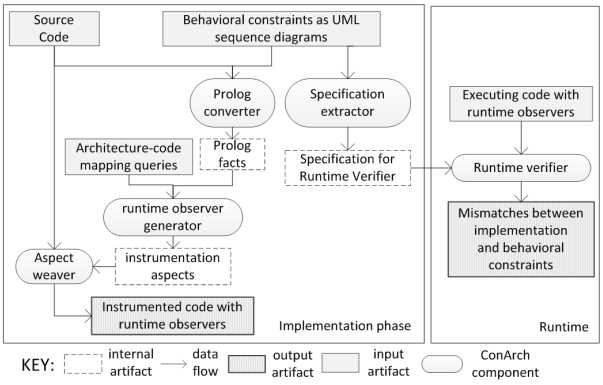


Fig. 2. ConArch framework: inputs, components, and output artifacts.

The runtime verification specification is a formalization of the behavioral constraints, and it is derived from the user specified sequence diagrams. The specification is used by the runtime verifier to detect whether the executing software system conforms to the behavioral constraints.

Using the behavioral models and the source code, ConArch constructs a list of architectural elements that needs to be mapped to the source code. We utilize Prolog [10] to provide programmable mapping. Here, the abstract syntax tree (AST) of the input source files are converted to Prolog facts. Using these facts, software engineers implement queries that return the points in the target program corresponding to the interactions. We chose to utilize Prolog in ConArch as it has been successfully used in the literature for querying the AST [11], [12].

ConArch uses the mapping queries while generating the runtime observers. The runtime observers are aspects [13] that intercept the execution when the target program reaches a point that corresponds to an interaction. Runtime observers are generated to collect information about the interception point. ConArch can generate runtime observers as aspects in AspectJ [13] or AspectC++ [14]. After generating the runtime observers, ConArch executes the appropriate aspect compiler to weave them to the source code.

To check the conformance, the software system is executed with ConArch's runtime verifier. Runtime observers send the information they collected to the runtime verifier. The runtime verifier assesses whether the execution of the target program follows the specified behavioral constraints. If the execution deviates from any constraint, it notifies the developer with

information about the deviation. The software engineer can use this information to formulate new behavioral constraints or correct the implementation. In the remainder of this section, we briefly describe the ConArch framework. Interested readers are referred to the literature [7] for a detailed description.

### *A. Modeling Behavioral Constraints and Generation of Runtime Verification Specification*

ConArch requires the behavioral constraints for the architecture to be modeled using UML sequence diagrams. In literature, UML sequence diagrams are often employed for modeling interactions between components/connectors, and it is a widely known modeling language. Figure 3-(a) shows a scenario modeled as a sequence diagram. This diagram shows the interaction between the auction house and one controller, although the same interaction occurs between many controllers and the auction house.

The optional frames, in the model, capture the interactions that are executed depending on a condition. For example, the controller (or the control node) is notified when the temperature change causes the air conditioner to turn on. ConArch supports both optional and alternative frames for modeling conditional executions.

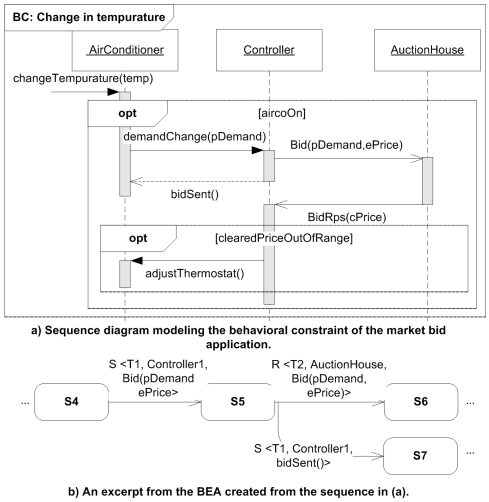


Fig. 3. A behavioral constraint of the market bid application.

Interactions can be synchronous or asynchronous. In the former, the component/connector sending the message waits until the receiver completes execution. In the latter, the sender does not wait for the receiver to complete. In this scenario, for example, asynchronous interactions are used to model the message exchange between the controller and the auction house as the auction collects all the bids from the controllers and then sends a response message.

ConArch converts the input sequence diagrams to a state machine with start and accept states. This state machine is called **Behavioral Execution Automata** (BEA). The reason for this conversion is twofold:

*i)* Due to alternative/optional frames and asynchronous interactions, the sequence diagram for a behavioral constraint contains multiple paths that the execution can follow. In a BEA, each of these are explicitly captured as a path from the start state to an accept state.

*ii)* At runtime, an interaction can correspond to the execution of multiple events in a sequence. For instance, in order to carry out an interaction, a series of method calls/message exchanges can be executed. Identifying where each one of these events are implemented for conformance checking can be a time consuming task. Hence, we designed ConArch to verify the send and receive message events of an interaction. In BEA these events are explicitly shown as transitions; that is, an interaction is represented as two consecutive transitions one for the send event and one for the receive event.

The transitions of a BEA are of the form $Event < Theard_{id}, Component, Interaction >$. Event can be send ($S$) or receive (R). The $Theard_{id}$ represents the identifer of the thread executing the event. During conversion, ConArch assigns logical thread identifiers to the transitions. At runtime, these identifiers are bound to the actual thread identifiers from the executing software system. Figure 3-(b) shows an excerpt form the BEA generated from the behavioral constraint shown in Figure 3-(a). Here, the transitions between states $S4$ and $S6$ correspond to the send and receive events of the asynchronous interaction *Bid()* between *Controller* and *AuctionHouse*. Note that these transitions are assigned different thread identifers, namely $T1$ and $T2$. In asynchronous interactions, the thread executing the send event is different from the one executing the receive events. Hence, different thread identifiers are used.

After an asynchronous interaction, the execution follows in two (interwind) distinct paths: one path contains the events executed by the thread receiving the message, and the other one contains the events executed by the thread sending the message. In Figure 3-(b), this is represented with the two transitions from state $S5$. The transition leading to state $S6$ represents the receive event of the interaction *Bid()*. The transition leading to state $S7$, on the other hand, corresponds to the interaction *bidSent()*.

### *B. Mapping Architecture to the Code*

The mapping is a semi-automated process; ConArch generates the necessary 'API' for architecture-code mapping. The users implement the mapping queries by means of this API. The generated API consists of two parts:

*i) Prolog facts representing the source code.* ConArch includes tools for converting Java and C++ AST to Prolog facts. Note that ConArch only converts the class structure and call statements to Prolog facts as these are the AST elements relevant to mapping. Figure 4-(a) lists an excerpt of the facts that are derived from the implementation of the market bid application. The same figure also details the AST elements that these facts represent.

*ii) Prolog rules representing each distinct event in the BEA specification.* Users implement the mapping queries in the bodies of these rules. For example, the user implements the mapping for the event send $Bid()$ from the component $Controller$ in the rule shown at the bottom of Figure 4-(a).

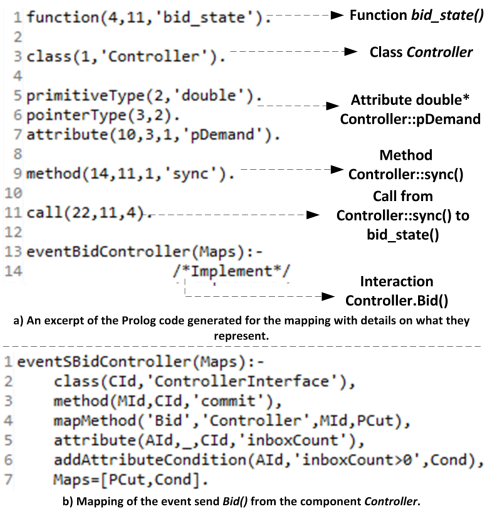


Fig. 4. Generated Prolog file, and the mapping of the event send $Bid()$ from the component $Controller$.

A mapping query for the event $event$ < *Interaction*> <*Component*> ( $Maps$ ) is the query on the AST that returns the points in the target program corresponding to $event$. ConArch utilizes aspects to monitor the execution of the program (similar to most runtime verification frameworks [15]). Thus, ConArch requires events to be mapped to points in the target program that can be intercepted with aspects. In addition to such points, ConArch provides facilities to map an interaction to a sequence of call/execute pointcuts or specify a condition for a mapping. For example, users can map the event send $Bid()$ from the component $Controller$ to the execution of the method *ControllerInterface.commit()* with the value of the attribute *inboxCount* greater than zero. This is considerably more flexible and fine-grained specification compared to traditional approaches, where the same interaction would be mapped to any call to the method *Controller.bid()*.

The user implements the mapping queries using the facts representing the AST. Figure 4-(b) shows how the event send $Bid()$ from the component $Controller$ is mapped to the execution of the method *ControllerInterface.commit()*. Here, the predicates at lines $2-4$ identify the method *ControllerInterface.commit()* and add a pointcut to intercept the execution of this method. The predicate $mapMethod$, a part of the mapping API, formulates an *execution* pointcut specification to intercept the method passed in the first argument.

The predicates at lines $5-6$, on the other hand, identify the attribute *Interface::inboxCount* and add a condition on this attribute. The predicate $addAttributeCondition$ is also part of the mapping API, and it is used for specifying conditions on the attributes that should be stratified for the mapping to succeed.

In addition to the predicate *mapMethod*, ConArch provides other predicates to be used in mapping queries: *i)* If the event maps to a call, then predicate $mapCall$ is used. This predicate formulates *call* pointcuts. *ii)* If the event maps to the return from a call, then the predicate $mapReturn$ is used. This predicate also formulates a *call* pointcut but an after advice is generated. *iii)* If the event maps to a sequence of calls and/or method executions, then the predicate $mapSequence$ is used. This predicate generates more than one pointcut specification to intercept all these calls and executions. ConArch generates the runtime observes to follow the flow of execution between these pointcuts.

```
1 aspect SendBid{
2     pointcut mapped(ControllerInterface *thisptr):
3         execution("% ControllerInterface.commit(...)") &&
4         that(thisptr)
5
6     advice mapped(thisptr) : before(ControllerInterface *thisptr){
7         long int tid=(long int)pthread_self();
8         if(thisptr->inboxCount>0)
9             RuntimeClient.newEvent("S(Controller.Bid",tid);
10    }
11 }
```

Fig. 5. Aspect for capturing the execution of the event send $Bid()$.

### C. Runtime Observation Aspects and Online Verification

Once the mapping queries are implemented, the user loads the Prolog file in ConArch's runtime observer generator. This tool executes the queries and generates the runtime observer aspects. For example, executing the Prolog rule shown in Figure 4-(b), returns an execution pointcut *ControllerInterface.commit()* and a condition on the attribute *Interface::inboxCount*. ConArch uses these to formulate the aspect shown in Figure 5. Here, the call at line 10 notifies the ConArch's runtime verifier that the target program executed the send event $Controller.Bid()$. Note that this notification is only sent when the condition on the attribute *Interface::inboxCount* is $true$.

The runtime verifier traces the BEA with the notifications about the events it receives from the runtime observers. If this trace leads to an accept state then that execution conforms to the behavioral constraint. If, however, the trace does not lead to an accept state then the execution does not conform to the constraint. In this case, the verifier prints out the trace in the BEA with the last notification that was received from the observer.

A major process in tracing the BEA is matching the notification about an event with the real thread identifer to the BEA transition with logical thread identifer. This process can be summarized as follows: Let's assume that the previous notifications lead the verifier to the BEA state $S_i$ with an outgoing transition $E_t < T_t, C_t, I_t >$ and that a runtime observer has sent the notification $E_n < T_{id}, C_n, I_n >$. Then, runtime verifier executes the following to determine whether the notification matches the outgoing transition from $S_i$:
*i)* Determine whether the executed event matches the event in the transition. That is, whether $E_t = E_n$, $C_t = C_n$, and $I_t = I_n$.
*ii)* Determine whether the real value of the thread identifier matches the logical thread identifier of the transition. This is determined according to two rules: If this is the first BEA transition with logical thread identifier $T_t$ then **bind** $T_t$ to $T_{id}$ (i.e., set $T_t$ to $T_{id}$). If $T_t$ is already bound (i.e., there was

transition in the path leading to $S_i$ with logical thread identifier $T_t$), then check if the bound value of $T_t$ is equal to $T_id$.
In case any of the steps described above fails, then the runtime verifier declares that execution does not conform with the behavioral constraint.

## V. Verifying the Behavioral Constraints of the Market Bid Application

This section demonstrates how ConArch is used for checking the conformance between the market bid application implemented with simulators and it's behavioral constraint shown in Figure 3-(a). Before going into details of the application, we describe the re-usable mapping queries we implemented for mapping the interactions to the facilities provided by the simulator frameworks.

### A. Styles for Implementing Interactions in Simulation Frameworks

When mapping the implementation of an application to the simulator, developers often need to separate the components of the applications into the modules of the simulator. Due to this separation, conventional inter-object communication mechanisms (such as method calls) cannot be used. The developers need to implement interactions with the facilities provided by the simulator. This is one of the major reasons why such implementations deviate from the original designed architecture.

We have studied various power and network simulators and identified implementation styles used to facilitate the communication between objects of separate modules (i.e., inter-module communication). As these styles can be employed by developers when implementing an application with a simulator, we implemented re-usable mapping queries to simplify the mapping process. Hereby, the developers only provide the names of the involved classes and methods. The queries locate the implementation style in the AST and return the points to be intercepted by means of the observer aspects. These queries are integrated to ConArch. We list the identified implementation styles below:

**Tick handler method and data sharing:** In tick based simulators, every object that needs to be notified about time updates registers a tick handler method. The simulator calls these handlers in an order. These simulators usually also provide mechanisms for data sharing between objects, which can be exploited to provide communication between objects during a tick. Here, an object $o_1$ receiving the tick call sets the attributes of another object $o_2$. When $o_2$ receives the tick call, it reads the value of this attribute and executes the computations accordingly. This style of communication can be used for implementing the send/receive events of an interaction.
In fact, the event send $Bid()$ from the component $Controller$ is implemented using this communication style, where controllers that want to send a *Bid* message to the auction house increment the value of the attribute $ControllerInterface :: inboxCount$. The method $ControllerInterface :: commit()$ is the tick handler method; when an instance of the class $Controller$ receives the $commit$ call, it checks if the value of the attribute $inboxCount$ is greater than zero. If so, it formulates a *Bid* message and sends it to the auction house.
We implemented the mapping query *mapTickSharing* ( $Interaction$, $Component$, $Class$, *TickHandler*, *SharedAttribute*, $Condition$ ) for mapping events of the interactions implemented with this style. For example, instead of the query shown in Figure 4-(b), one can use *mapTickSharing* ( *'Bid'*, *'Controller'*, *'ControllerInterface'*, *'commit'*, *'inboxCount')*, *'inboxCount>0'* ) for mapping the event send $Bid()$.

**Tick notification method:** An object receiving a call to it's tick handler method might also correspond to an event of an interaction. For mapping such events, we implemented the mapping query $mapTick$ ( $Interaction$, $Component$, $Class$, *TickHandlerMethodName* ).

**Scheduling *events*:** Inter-module communication can be achieved using the *event* scheduling mechanism provided by discrete event simulators (for clarity, we use *event* for referring to events of a discrete event simulator). Here, an object $o_1$ wanting to send a message to another object $o_2$, schedules an *event* destined to $o_2$. It also attaches any data items it wants to pass to $o_2$ with this message. The simulator dispatches the event by calling $o_2$'s *event* handler method.
The send/receive event of an interaction can be implemented by scheduling events. We implemented the mapping query *mapScheduleEvent* ( $Interaction$, $Component$, *Class*, *MethodSchedulingEvent*, *EventScheduleMethod*, *EventData*, *Condition* ) for mapping the events of the interactions implemented using this style. Here, the last two arguments are optional, and they are used for implementing a condition on the data attached to the *event*.

**Dispatch of *events*:** The send/receive event of an interaction can correspond to an object receiving the *event* dispatched by the simulator. For mapping such events, we implemented the mapping query *mapDispatchEvent* ( $Interaction$, $Component$, *Class*, *DispatchedMethod*, *EventData*, *Condition* ). Similar to the previous mapping query, the last two arguments are optional.

### B. Application of ConArch to Market Bid Application

The market bid application has two main components: auction house and controller. The parts of these components related to sensing the demand/supply, calculation of the cleared price, and controlling power components are integrated to the power grid simulator GridLab-D. These parts are implemented in 7 classes/structures with $\approx$ 12KLoc.

The parts related sending bid/response messages and network address configuration are implemented with the network simulator ns-3. These parts are constitute 2 classes with $\approx$ 1KLoc. Developers used a custom middleware to handle the communication between these two simulators.

```
1 eventRBidRpsController(Maps):-
2     class(CId,'ControllerInterface'),
3     class(CTId,'CommGLDNs3'),
4     method(MId,CTId,'getMessage'),
5     mapReturn('Bid','Controller',MId,CId,PCut),
6     addReturnValueCondition(MId,'msg_type==RSP_MSG',Cond),
7     Maps=[PCut,Cond].
```

Fig. 6. Mapping the event receive *BidRsp* to the return of the call to the method *CommGldNs3::getMessage()*.

To verify the behavioral constraints in Figure 3, we used these 9 classes. The conversion of the AST to Prolog facts completed in $\approx$ 6minutes and generated 17488 facts (using a laptop with core $i5$ processor). Out of the 10 events of the constraint, only two required us to implement a mapping query. The remaining events were all mapped using the queries listed about; mostly the *Tick handler method and data sharing* implementation style is used as developers of GridLab-D frequently employ this style to implement inter-object and inter-module communications.

The two events that required a custom mapping query dealt with receiving the messages from the middleware. Figure 6 presents the mapping query we used for one of these events, namely the event component *Controller* receive *BidRsp*. Here, the predicates at lines $2-4$ locate the class *ControllerInterface* and the method *CommGldNs3::getMessage()*. With the predicate *mapReturn* this event is mapped to return of call the from an instance of the class *ControllerInterface* to the method *CommGldNs3::getMessage()*. Note that, the query also sets a condition on the return value of the method.

The mapping completed in $\approx$ 15seconds, as the queries on the AST were not complicated. With the supplied mappings, ConArch generated 10 aspects that are weaved to the code using the AspectC++ compiler. We have executed the market bid application with a power grid model consisting of 61 components. Simulations of this application usually take around 45 minutes to complete; the runtime observes added a $\approx$ 11 overhead which was not that much compared to the overall execution time. On the course of the simulation ConArch's runtime verifier detected various violation of the behavioral constraint. The output showed that on certain occasions the component $Controller$ executed a send $Bid()$ event instead of a send $adjustThermostat()$ event. The developers confirmed that this is a known problem, and it is due to improper handling of the delayed response message from the auction house. If this message gets delay, the component *Controller* assumes it's bid message is lost and resend a bid message.

Developers confirming the findings of ConArch shows that it can be indeed used to verify the behavioral constraints of software systems implemented with simulators. The programmable mapping allowed us to map interactions to employed inter-module communication facilities. In this way, ConArch was able to trace the interactions and detect the violations. Below important findings of this case study are listed:

1) Programmable mappings provided sufficient flexibility to map interactions to implementation with simulator frameworks.
2) It is necessary to verify the behavioral constraints on implementation with simulator frameworks. As errors such as the one detected by ConArch can happen. In this case, the error caused unnecessary network traffic and delayed other important messages.
3) Re-useable mapping queries for mapping interaction to common inter-module communications simplified the mapping process. Similar queries can be formulated for mapping interactions to frequently used inter-object communications (e.g., method calls). As future work we plan to extend ConArch with a repository containing mapping queries for mapping such communication styles.

## VI. Related Work

There have been dynamic analysis techniques introduced [4], [5] for analyzing the runtime behavior of a system. These techniques are mainly employed for the purpose of reverse engineering, and not on verifying the consistency of an architecture documentation at runtime. The derived behavioral models could be checked (offline) with respect to the existing documentation. However, there is a lack of formalized mapping between the generated models and existing documents. As such, these approaches do not facilitate automated consistency checking.

Recently, ArchSync [3] was introduced as a tool approach that assists architects to check the conformance between a scenario-based architectural description and the implementation. This approach does not facilitate a flexible mapping between the architectural elements and the implementation elements (limited to Java classes and methods).

In [11], the data obtained from the dynamic analysis is represented as Prolog facts. Users implement Prolog rules defining violations of the communication constraints of the software architecture. These rules are evaluated over the facts to identify method calls that should not have been executed. Many other dynamic analysis and architecture reconstruction techniques are surveyed in [16]. These are also introduced for reverse engineering, but not on verifying the consistency of an architecture documentation at runtime.

Mosaik [17] is proposed as an integrated co-simulation framework for smart grid applications. Here, naming conventions are used for mapping different models to implementation for checking the correspondence. Unfortunately, mapping based on naming conventions is fragile and simulation frameworks might have their own naming conventions for modules to follow. ConArch, on the other hand, does not enforce any conventions on the implementation. It provides a querying mechanism for developers to describe how interactions of the behavioral scenarios are implemented. As we have shown in Section V, we were able to map the interactions behavioral models to the implementation even when the names of methods did not match the names of interactions.

Previously we introduced ConArch [7], which is a runtime verification approach for detecting inconsistencies between the dynamic behavior of the documented architecture and the actual runtime behavior of the system. As a major limitation, ConArch required the user to manually implement the mapping between the design elements and source code elements in the form of Prolog rule. In this work, we have introduced a domain-specific, reusable and extensible repository of mapping rules. We have derived these rules based on common implementation styles that are followed by developers to implement interactions in simulation frameworks. This extension enabled ConArch to perform automated mapping for the verification of smart grid control applications.

## VII. Conclusion

Developers usually do not have direct access to a large sensor network for testing the smart grid applications. As such, very often simulators are used to analyze and predict the behavior of these grids. Due this mapping, the implementation easily deviates from the documented architecture, which impedes the analysis of behavioral constraints that are defined at the architecture design level.

In this paper, we show that the programmable mapping feature of ConArch can be utilized for checking the conformance between the implementation with simulators and the behavioral constraints. In ConAcrch, behavioral constraints are modeled as sequences of interactions between components and connectors. ConArch derives a formal verification automata from these models. Users implement mapping queries on the abstract-syntax trees that return the points in the target program corresponding to the interactions. These queries can, for example, be used to map interactions to the inter-object communications mechanism provided by the simulator framework.

ConArch generates aspects that observe the execution and notify the runtime verifier when it reaches a point corresponding to an interaction. ConArch's runtime verify traces the automata (generated from the behavioral constraint) with these notifications. If a notification is not accepted by the automata then there is a mismatch between the implementation and the constraint. Hence, ConArch displays an error message showing the notification and it is mapping to the implementation.

We have illustrated ConArch for the verification of behavioral constraints of a smart grid application implemented with two simulators. With the programmable mappings, we were able to provide ConArch locations corresponding to the interactions. Thus, ConArch was able observe the execution and identify executions that did not follow the behavioral constraints. The developers of the application confirmed that these executions were indeed erroneous.

We also extended ConArch with re-usable mapping queries for mapping interactions to frequently used inter-object communication styles used in simulators. Instead of implementing a query on the abstract syntax tree, developer can use these queries by just providing the names of the classes and methods involved in the communication. In fact, we have heavily used these queries during the case study which eased the mapping process considerably. Similar re-usable queries can be provided for other common inter-object communication styles (such as method calls). As future work, we plan to extend ConArch with a repository of mapping queries, where user can select the communication style and provide the names of the classes.

## References


[1] I. Gorton, Z. Huang, Y. Chen, B. Kalahar, S. Jin, D. Chavarría-Miranda, D. Baxter, and J. Feo, "A high-performance hybrid computing approach to massive contingency analysis in the power grid," in *E-Science '09*, 2009, pp. 277–283.

[2] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–654, 2002.

[3] J. Diaz-Pace, A. Soria, G. Rodriguez, and M. Campo, "Assisting conformance checks between architectural scenarios and implementation," *Information and Software Technology*, vol. 54, no. 5, pp. 448 – 466, 2012.

[4] G. Huang, H. Mei, and F.-Q. Yang, "Runtime recovery and manipulation of software architecture of component-based systems," *IEEE Trans. on Software Engineering*, vol. 13, no. 2, pp. 257 – 281, 2006.

[5] L. Qingshan et al., "Architecture recovery and abstraction from the perspective of processes," in *WCRE*, 2005, pp. 57–66.

[6] J. Nutaro, "Designing power system simulators for the smart grid: Combining controls, communications, and electro-mechanical dynamics," in *IEEE Power and Energy Society General Meeting*, 2011, pp. 1 –5.

[7] S. Ciraci, H. Sozer, and B. Tekinerdogan, "An approach for detecting inconsistencies between behavioral models of the software architecture and the code," in *COMPSAC*, 2012, pp. 257 – 266.

[8] D. Chassin, K. Schneider, and C. Gerkensmeyer, "Gridlab-d: An open-source power systems modeling and simulation environment," in *T&D IEEE/PES '08*, 2008, pp. 1 –5.

[9] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and Tools for Network Simulation*, K. Wehrle, M. Günes, and J. Gross, Eds. Springer Berlin Heidelberg, 2010, pp. 15–34.

[10] M. A. Covington, D. Nute, and A. Vellino, *Prolog programming in depth*. Scott, Foresman & Co., 1987.

[11] C. Riva and J. Rodriguez, "Combining static and dynamic views for architecture reconstruction," in *CSMR*, 2002, pp. 47–55.

[12] S. Ciraci, P. van den Broek, and M. Aksit, "Graph-based verification of static program constraints," in *SAC '10*, 2010, pp. 2265–2272.

[13] G. Kiczales et al., "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220 – 242.

[14] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "Aspectc++: an aspect-oriented extension to the c++ programming language," in *CRPIT '02*, 2002, pp. 53–60.

[15] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *EEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859 – 872, 2004.

[16] D. Pollet et al., "Towards a process-oriented software architecture reconstruction taxonomy," in *CSMR*, 2007, pp. 137 – 148.

[17] S. Schutte, S. Scherfke, and M. Troschel, "Mosaik: A framework for modular simulation of active components in smart grids," in *SGMS'11*, 2011, pp. 55–60.