SaVeWoT: Scripting and Verifying Web of Things Systems and Their Effects on the Physical World

Justus Fries[1 [0000−0003−3433−7245]], Michael Freund[2 [0000−0003−1601−9331]], and Andreas Harth[2,3 [0000−0002−0702−510X]]

[1] Technical University of Munich, Munich, Germany
[2] Fraunhofer Institute for Integrated Circuits IIS, Nuremberg, Germany
[3] Friedrich-Alexander-Universität Erlangen-Nürnberg, Nuremberg, Germany

Abstract. We introduce SaVeWoT (Scripting and Verifying Web of Things Systems), an approach for designing, formally verifying, and deploying decentralized control systems based on the W3C WoT. SaVeWoT consists of two main parts: the SaVeWoT language and the SaVeWoT compiler. The SaVeWoT language models devices (i.e., Things), controllers that orchestrate Things, virtual composite Things (i.e., subsystems consisting of multiple Things), interactions between these components, and their effects on the physical world. The SaVeWoT compiler uses Thing Descriptions (TDs) and SaVeWoT behavior descriptions along with correctness specifications in Linear-time Temporal Logic (LTL) to automatically generate a Promela model, which is validated using the SPIN model checker. We demonstrate the feasibility of the SaVeWoT approach by verifying a conveyor belt system as an example and conducting an empirical evaluation.

Keywords: Web of Things · Static Verification · Safety.

1 Introduction

The Web of Things (WoT) and the WoT Thing Description (TD) ontology enable interoperable interaction with Internet of Things (IoT) devices (Things). When building networked automation and control systems, e.g., in manufacturing [1] or building automation [2], interoperability enables the use of devices from various vendors and manufacturers. Interoperability is helpful for deployment and during the development and design phase of a system, as engineers can compare and use different components (e.g., devices, subsystems, and solutions) from various vendors. Because controllers interact with Things according to a specification developed by an engineer [2], integrating Things requires not only a description of the Things but also of the interactions between controllers and Things (i.e., all system components). These interactions can become complex as the system grows and cause unintended but subtle effects or bugs. With SaVeWoT we enable engineers to find such unintended effects via model checking [3]. We focus on physical world effects and want to integrate SaVeWoT into the design phase of system development to find bugs as early as possible.

Integrating model checking into any design process leads to two main challenges: Ease of use and choosing the right level of abstraction for the model to check [3]. Existing approaches to model checking WoT systems [4] solve these issues by relying on rule-based programming and introducing strong assumptions (discussed in section 3.2) about how Things interact. For SaVeWoT, we introduce and use a scripting language based on JavaScript. SaVeWoT eases the assumptions, which enables SaVeWoT to integrate the physical world into model checking through descriptions of behavior via scripts.

Additionally, previous work on model checking in IoT and WoT has primarily focused on the Trigger-Action Programming (TAP) where TAP rules are specified independently and trigger based on events to execute actions. While TAP rules are useful for smart home scenarios, the W3C WoT covers various domains, and the WoT Scripting API [5] is based on JavaScript and not rules. In addition, many previous approaches did not consider the physical world as a part of the system and instead formally verified the state of Things.

Our approach, SaVeWoT, enables model checking WoT systems using WoT TDs and scripts. Central to our approach is a model of the physical world described using ontologies [6], specifically the TD ontology [7] as well as the Sensor, Observation, Sampler, and Actuator (SOSA) and Semantic Sensor Network (SSN) ontologies [8]. Things sense or actuate so-called Features of Interest (FoIs) of the physical world, e.g., the temperature in a room. Such a model of the physical world is machine-readable, usable for inference or reasoning, and can be integrated with other physical world models. For SaVeWoT we define composite Things [9, 10] as Things that control other Things, i.e., subsystems of the WoT system. Concurrency in WoT systems is introduced via concurrently running controllers.

We define a formal language based on JavaScript and the WoT Scripting API that is executable and designed for translation to formally verifiable models. When describing behavior in scripts, interactions with the physical world are expressed via reading (e.g., sensing the temperature) and writing (e.g., setting the temperature on a thermostat) variables representing FoIs.

After introducing our approach using an example (section 2), we provide related work (section 3). The contributions of this paper are:

- A formal WoT system model (section 4) that explicitly facilitates formal verification of physical world effects.
- An executable, formally verifiable language (section 5) based on W3C WoT standards.
- A proof-of-concept implementation (section 6) and an evaluation that illustrates the feasibility of our approach (section 7).

Finally, section 8 summarizes the most important aspects of our approach and highlights future work.

2 SaVeWoT by Example

We now illustrate our approach with an example WoT system for a conveyor belt that moves objects on the belt in one direction, see figure 1 (1). The system consists of the following components (figure 1 (2)): a motor drives the conveyor belt, two photoelectric sensors detect objects on the belt, and a controller. The controller orchestrates the motor to move objects, e.g., work pieces, from the start to the end of the belt. The physical world is modeled using two FoIs that are sensed by the respective sensors. The first sensor measures the physical space at the start of the conveyor belt (`objectAtStartFoI`), while the second sensor measures the physical space at the end of the belt (`objectAtEndFoI`)

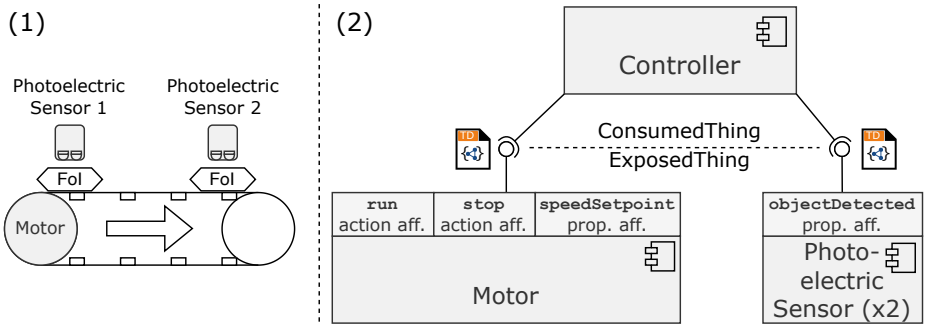

Fig. 1. Physical layout with Features of Interest (1) and components and interfaces (2) of the example WoT system. We refer to the FoI measured by "Sensor 1" as `objectAtStartFoI`, while the FoI measured by "Sensor 2" is called `objectAtEndFoI`.

2.1 Web of Things by Example

In the context of WoT, the motor and photoelectric sensors are Things described with a TD. The motor has three interaction affordances: the action affordances `run` and `stop`, and an integer property affordance for reading and writing the `speedSetpoint`. If the `speedSetpoint` is not set to a value greater than zero, the motor does not run. The sensor has a read-only boolean property affordance indicating whether the sensor detected an object or not, called `objectDetected`. Both photoelectric sensors have an `objectDetected` property affordance. However, each sensor measures its respective FoI. The relations between interaction affordances and FoIs are part of the TDs of the sensors and actuators.

The WoT Scripting API defines `ConsumedThing` and `ExposedThing` interfaces. A controller script is written against `ConsumedThing` interface methods based on the interaction affordances in a TD. On the other hand, the behavior of a Thing is described by a set of handlers as defined by the `ExposedThing`

interface. Each handler describes the behavior of the Thing for a specific interaction affordance (e.g., the action `run`) and operation (in this example: invoke, read or write).

The script for controller behavior is shown in listing 2.1. If an object is at the start of the belt, the controller runs the motor to move the object. Once the object is at the end, the motor is stopped. Listing 2.2 shows the script for the Thing behavior. The motor moving an object on the belt is modeled through the FoIs. The motor affects the physical world in a single step, i.e., an object is either at the start or end of the belt. As a result, stopping the motor does not affect any FoI.

```
while (true) {
   objectAtStart = sensorStart.readProperty(objectDetected);
   objectAtEnd = sensorEnd.readProperty(objectDetected);
   if (objectAtStart && !objectAtEnd) {
      motor.invokeAction(run);
   } else {
      motor.invokeAction(stop);
}}}
```

Listing 2.1. Controller script based on the `ConsumedThing` interface that moves objects on the conveyor belt. This controller script has a bug: It does not set the `speedtSetpoint`.

```
motor.setActionHandler(run, () => {
   if (speedSetpoint > 0 && objectAtStartFoI) {
      objectAtStartFoI = false; objectAtEndFoI = true;
   } else {
      if (speedSetpoint > 0 && objectAtEndFoI) {
         objectAtEndFoI = false;
}}});
motor.setActionHandler(stop, () => {});
motor.setPropertyReadHandler(speedSetpoint, () => {
   return speedSetpoint; });
motor.setPropertyWriteHandler(speedSetpoint, (arg) => {
   speedSetpoint = arg; });
sensorStart.setPropertyReadHandler(objectDetected, () => {
   return objectAtStartFoI; });
sensorEnd.setPropertyReadHandler(objectDetected, () => {
   return objectAtEndFoI; });
```

Listing 2.2. Thing script with motor and sensor handlers based on the `ExposedThing` interface.

2.2 Model Checking by Example

SaVeWoT is designed to reuse the syntax of existing W3C WoT standards while ensuring formally verifiable execution semantics. For each handler (listing 2.2)

and the controller (listing 2.1), the SaVeWoT compiler adds a process to the Spin model, and for each FoI, a variable is introduced. The controller can interact with Things using channels (see section 4.1), and standalone Things can change the physical world by changing FoI variables. The FoI variable for `objectAtStart-FoI` is initialized as true and `objectAtEndFoI` as false to represent that there is an object at the start.

The example controller script contains a bug that causes the WoT system to not move objects: the motor driving the belt must have its speed set point set to a value > 0 to move any object on the conveyor belt, which the controller does not do. With SaVeWoT and model checking, an engineer can specify the requirement "the conveyor belt must eventually move objects to the end of the belt," using the "existence" LTL pattern [11] "`objectAtEndFoI` must be true at least once" (see section 4.2). Checking the LTL pattern on the Spin model shows that the program does not satisfy the pattern.

3 Related Work

We group previous work into behavioral extensions to WoT and formal verification, including translation approaches, of control, IoT, and WoT systems.

3.1 W3C Web of Things Behavior

Several proposed extensions for WoT describe behavior through annotations. Such extensions include permitted sequences of interactions [12], JSON-based descriptions corresponding to UML sequence diagrams that represent mashups [13], effects of action affordances on property affordances [14], effects expressed as JavaScript embedded in TDs [15], and extending TDs with physics simulations [16]. Generally, these approaches just represent applications built on top of WoT interfaces and do not support any form of formal verification or mechanisms to enforce restrictions or requirements.

3.2 Formal Verification

Recently, multiple tools and approaches have been proposed where a programming language is used to generate both models and deployments. Lin *et al.* [3] propose model checking models generated from Lingua Franca, a coordination language for Cyber-Physical Systems. Similarly, for distributed systems, Hackett *et al.* [17] present the language MPCal and the compiler PGo to translate TLA+ models [18] to Go. Lastly, Lattuada *et al.* [19] enable formal verification of a subset of the Rust language for systems programming.

Related work on formally verifying control systems is primarily based on Programmable Logic Controllers (PLCs). Approaches with a focus on usability using LTL patterns [20], on modeling communication link failures [21], or on verifying communication time constraints [22] have been proposed. An early

case study for model checking a production cell [23, 24] modeled a production cell without concrete PLCs.

Previous work [25–29] on model checking IoT systems focused on TAP (also called Event-Condition-Action (ECA)) rules and interactions between rules/programs due to a focus on smart home IoT. The general goal is to verify that rules do not cause bad situations or interfere with each other. Some approaches focus purely on the interactions of rules, while others incorporate models for IoT devices, traces of the system, or timed automata. Instead of TAP rules, we use a DSL based on JavaScript to ensure compatibility with existing WoT standards.

Krishna *et al.* [4] model check WoT systems based on a predecessor of the W3C WoT. The authors create models of Things from just TDs using the following assumptions: Actions directly cause the emission of corresponding events, actions modify the properties of a Thing, and Things cannot be composed to create composite Things. SaVeWoT, in contrast, does not restrict how interaction affordances relate to each other and instead relies on the descriptions of Thing behavior in scripts.

4 Modeling Things and the Physical World

We first introduce the components of a WoT system and next show how the architectural composition of Things is represented in SaVeWoT. We explain how the physical world, communication and behavior are described in SaVeWoT and finally show how these previous design decisions enable formally verifying effects on the physical world via LTL formulas.

4.1 WoT System Components

In SaVeWoT, a WoT system consists of Things and controllers. Formally, we define a set of Things T and a set of controllers C. Each Thing can be uniquely identified by an IRI. A Thing can have property affordances P, action affordances A, and event affordances E. Conceptually, interaction affordances are uniquely identifiable individuals and belong to exactly one Thing, while a Thing may have multiple affordances: $\mathsf{pAff} : \mathsf{P} \to \mathsf{T}$, $\mathsf{aAff} : \mathsf{A} \to \mathsf{T}$ and $\mathsf{eAff} : \mathsf{E} \to \mathsf{T}$.

Property affordances are either read-only, write-only or read-write. We introduce a set of possible read-write combinations $\mathsf{RW} = \{r, w, rw\}$ and assign one to each property affordance: $\mathsf{propRW} : \mathsf{P} \to \mathsf{RW}$. Reading a property returns data to the controller synchronously, while writing pushes data to the Thing.

Action affordances are invoked by controllers. When invoking an action affordance, a controller can provide input parameters and receive output data.

Finally, event affordances enable a controller to subscribe to receive notifications from a Thing. The Thing can then emit events that contain data to the controller.

Composition SaVeWoT enables the design of subsystems through the composition of Things. An example of such a subsystem is the initial conveyor belt

example. If a conveyor belt is also modeled as a Thing with interaction affordances, the belt is a composite Thing that controls a motor. Composite Things are Things that control other Things. Relationships between Things can be expressed using the SSN ontology through the hosts relation.

Restrictions Based on W3C Implementation Report TDs For the development of SaVeWoT, we surveyed the TDs from the W3C WoT TD implementation report[4] using SPARQL queries to find an expressive and often used subset of interaction affordances and data types (i.e., data schemas) to limit the state space of WoT system models. The surveyed TDs contain 90 Things with 368 property affordances, 66 action affordances and five event affordances. Due to the low prevalence of event affordances, the initial version of SaVeWoT does not support event notifications and instead focuses on the more relevant property and action affordances.

Modeling the Physical World and Standalone Things The initial conveyor belt example showed how FoIs can be used. FoIs can cover a wide variety of aspects of the physical world: FoIs can represent a window, a room in a house, or the house itself or the physical space deployed in front of a sensor[5]. Running a conveyor belt's motor moves objects on the belt toward the sensor. We model that sensors measure the results of actuations by introducing global variables for FoIs in the system model. Actuators can change a FoI variable's value, and sensors read the value. How the actuator changes the value is described in a script.

```
@prefix td: <https://www.w3.org/2019/wot/td#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix ssn: <http://www.w3.org/ns/ssn/> .
_:objectAtEndFoI a sosa:FeatureOfInterest .
[ a td:PropertyAffordance ;
   ssn:forProperty [ a sosa:ObservableProperty ;
            ssn:isPropertyOf _:objectAtEndFoI ]].
[ a td:ActionAffordance ;
   ssn:forProperty [ a sosa:ActuatableProperty ;
            ssn:isPropertyOf _:objectAtEndFoI ]].
```

Listing 4.1. TD (in Turtle) with a FoI and two interaction affordances (based on [30]).

Formally, we introduce a set F of all FoIs. The relation between FoIs and Things is based on previous work by Charpenay and Käbisch [30] and the W3C WoT TD standard [7, sec. 7.1] that aligns SOSA/SSN with the TD ontology. Listing 4.1 shows an RDF example of the FoIs of a `objectDetected` property affordance and a `run` action affordance in a conveyor belt system. An interaction

[4] https://w3c.github.io/wot-thing-description/testing/report.html; available as a single JSON-LD file at https://www.vcharpenay.link/talks/td-sem-interop.html.

[5] See https://www.w3.org/TR/vocab-ssn/#SOSASample.

affordance is related to an observable (for sensors) or actuatable (for actuators) property (*a different concept than property affordances*). A property is associated with (`ssn:isPropertyOf`) a FoI. For our formalization, we ignore properties and instead introduce a direct relation from affordances to FoIs: $hasFoI \subseteq (\mathsf{P} \cup \mathsf{A} \cup \mathsf{E}) \times \mathsf{F}$.

Interactions as Channel Communication Verifiable models consist of sequential processes communicating with each other using channels, which are FIFO queues containing messages [31]. Due to our focus on property and action affordances, we can model interaction affordances using a request-response pattern. When a controller uses an interaction affordance, in the Spin model, the controller sends a request message on a channel and immediately waits for a response message.

In the translation to Spin, we introduce a channel for each interaction affordance operation of each Thing and translate controllers and each affordance handler to processes that interact via channels. As a result, different handlers of a Thing can be executed concurrently. However, one controller can use only one handler at a time.

4.2 Specifying Required Physical Effects

With SaVeWoT we aim to enable formal verification of effects on the physical world. For formal verification, correctness properties that the system must fulfill, i.e., requirements, are specified in a temporal logic. LTL has emerged as one of the main temporal logics for model checking [32].

LTL is based on propositional logic. Thus, LTL formulas contain atomic propositions (APs) that are either *true* or $false$ in a system model's states. Our approach aims to verify effects on the physical world, which we achieve by introducing FoIs as global variables. Since FoIs are variables, they can be used in LTL formulas in value comparisons. For example, a presence sensor measuring occupancy of a physical space can be expressed as the AP `objectDetected=true`.

Additionally, LTL formulas consist of the operators from propositional logic (e.g., $\rightarrow$, $\neg$, $\vee$, $\wedge$) as connectives between APs, and temporal operators can be used as connectives to refer to time. The relevant temporal operators are: "Always" ($\Box\varphi$) means the system model always has to satisfy φ, "Eventually" ($\Diamond\varphi$) means the system model has to satisfy φ now or at some point in the future, and "Until" ($\varphi_1 \,\mathcal{U}\, \varphi_2$) means the model has to satisfy φ_1 until the model satisfies φ_2.

- Eventually there is an object at the end of the conveyor belt (*existence pattern*): $\Diamond$`objectAtEndFoI`
- An object on a conveyor belt cannot be at two locations (FoIs) at the same time (*absence pattern*): $\Box\neg$(`objectAtStartFoI` $\wedge$ `objectAtEndFoI`)
- An object progresses from the start to the end of the belt (*response pattern*): $\Box$(`objectAtStartFoI` $\rightarrow \Diamond$`objectAtEndFoI`)

- Trivial LTL formula that is satisfied by any system model (used in section 7): $\square\, true$
- After an object is at the start of a belt, the object must be detected at the end next (used in section 7): $\square(\texttt{objectAtStartFoI} \rightarrow \texttt{objectAtStartFoI}\, \mathcal{U}\, (\neg\texttt{objectAtStartFoI} \wedge (\neg\texttt{objectAtStartFoI}\, \mathcal{U}\, \texttt{objectAtEndFoI})))$

LTL formulas, including nested formulas, are generally difficult to grasp for non-experts [33]. A solution could be LTL patterns or templates that enable non-experts to specify correctness properties [34].

5 Behavior Language

Now we present the formal grammar of the SaVeWoT language. The language is a subset of JavaScript combined with the WoT Scripting API. The grammar has C-style control flow with `if` and `while` statements. We split the grammar into two parts, the grammar for controllers and the grammar for Things.

5.1 Controller Scripts

```
ControllerScript ::= statement+
statement ::= ifStmt | whileStmt | exprStmt | assignStmt | interactAffStmt
ifStmt ::= "if" parenExpr blockStmt ("else" blockStmt)?
whileStmt ::= "while" parenExpr blockStmt
exprStmt ::= expression";"
assignStmt ::= idOrField "=" (interactAffExpr | expression)";"
interactAffStmt ::= ID "." (invActExprStmt | writePropStmt)";"
parenExpr ::= "(" expression ")"
blockStmt ::= "{" statement* "}"
interactAffExpr ::= ID "." (invActExprStmt | readPropExpr)
expression ::= term (relationExpr | binOpExpr)? | "!"term
invActExprStmt ::= "invokeAction(" ID ("," term)? ")"
writePropStmt ::= "writeProperty(" ID "," term ")"
readPropExpr ::= "readProperty(" ID ")"
relationExpr ::= ("<" | ">" | "==" | "≤" | "≥" | "≠" | "&&" | "||") term
binOpExpr ::= ("+" | "−" | "×" | "÷") term
term ::= idOrField | INT | STRING | BOOL | parenExpr
idOrField ::= ID | ID "." ID
```

Listing 5.1. Simplified controller grammar based on the `ConsumedThing` interface for client interactions.

Each controller script represents the behavior of one controller. Listing 5.1 shows the formal grammar of the language for controller scripts in W3C EBNF[6]. The WoT Scripting API `ConsumedThing` interface is integrated by using expressions (**Expr*) for interaction affordances with output and by using statements (**Stmt*)

[6] https://www.w3.org/TR/2010/REC-xquery-20101214/#EBNFNotation

for affordances without output. Statements are nonterminals that do not produce any output and represent an instruction to do something, while expressions are evaluated and stand for a value that is returned. For easier translation while ensuring correct operational semantics, the production rules restrict interaction affordance expressions to only appear on the right-hand side of assignment statements and not, e.g., within an `if` statement's condition.

5.2 Thing Scripts

```
ThingScript ::= (handler)+
handler ::= ID "." (invActHandler | writePropHandler
            | readPropHandler)";"
invActHandler ::= "setActionHandler(" ID
            ", (" ID? ") => " (blockStmt | blockStmtWithRet) ")"
writePropHandler ::= "setPropertyWriteHandler(" ID
            ", (" ID ") => " blockStmt ")"
readPropHandler ::= "setPropertyReadHandler(" ID
            ", () => " blockStmtWithRet ")"
blockStmtWithRet ::= "{" statement* returnStmt "}"
returnStmt ::= "return" expression";"
```

Listing 5.2. Abbreviated Thing grammar based on the `ExposedThing` interface for server interactions.

Listing 5.2 shows the formal grammar of the language for Thing scripts. A handler program is a set of handlers for one or more Things. Handlers implement the `ExposedThing` interface. A handler can act as a controller because the handler bodies consist of *statements* from the controller script syntax (Listing 5.1), which enables composite Things. Like the controller grammar, the Thing grammar enforces that reading a property affordance returns a value but does not accept any argument, and vice versa for writing property affordances.

6 Implementation of the Compiler

We implement a proof-of-concept (PoC) of SaVeWoT in JavaScript using the Chevrotain parser library[7]. Our executable scripts use the WoT Scripting API implementation `node-wot`[8]. As a target model checker, we chose Spin [35] as Fu *et al.* [36] already showed that Spin can verify various Web Services based on behavior descriptions. The translation workflow from TDs and scripts to models can be seen in figure 2. The visitors iterate over the syntax trees and emit partial models, which are assembled to a Promela model by the linker. The visitors and the linker use the sets and relations used to formalize WoT systems (extracted from TDs using SPARQL) to link all behavior descriptions and variables in the model together.

[7] https://chevrotain.io
[8] https://github.com/eclipse-thingweb/node-wot

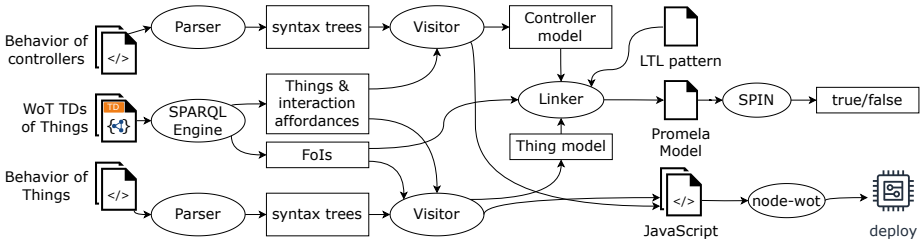

Fig. 2. Dataflow diagram showing the translation of TDs, scripts and correctness requirements to the Spin model checker.

The input language of Spin, Promela, supports channel and C-style control flow (`if` and `while`), which makes the translation from the SaVeWoT language to Promela straightforward. The PoC handles the differences in blocking semantics and variable scope between JavaScript and Promela. The translation to JavaScript avoids the event loop (JavaScript's built-in concurrency mechanism) because its operational semantics are more complicated [37] than the operational semantics of Promela [38]. We instead facilitate concurrency by creating separate scripts and using mutex locks during translation by the visitors.

7 Evaluation

To study the feasibility of our approach for large control systems with many components, we evaluate Promela models translated by our PoC on consumer hardware with an Intel i5-8295U CPU and 16 GB of RAM. We consider a series of conveyor belts orchestrated by a controller. Each belt has a motor and an ultrasonic sensor, which interact with the physical environment via a FoI. The controller implements a control loop for each belt. The belt's motor runs until a certain distance is observed, after which the motor stops. We evaluate the number of states, verification time, and verification memory consumption as measured by Spin for Promela models.

First we increase the number of belts from one to 58, which increases the number of Things (two per belt) from two to 116 respectively. Figure 3 (left) shows the results. The number of states is linear in the number of Things. The initial model with two Things has 76 states, which is increased to 4180 states for 116 Things. The memory consumption appears almost linear, starting at 128.73 MB for one belt and increasing to 134.39 MB for 116 Things. The verification time exhibits a step-wise pattern. Spin measures zero ms for up to 30 Things and 60 ms for 116 Things.

In a second step, we verify the system with 28 belts (the highest number of belts with a verification time of 10 ms) for increasingly large conjunctive LTL formulas on the FoIs of each belt. We generate the verifier for up to 18 belt motors in the LTL formula. For LTL formulas longer than 18 belts Spin takes multiple hours to generate a verifier, independently of the verification time.

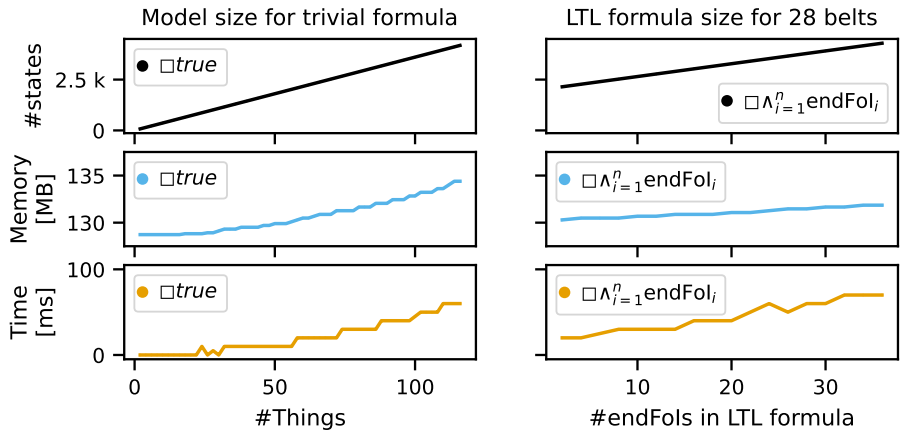

Fig. 3. Number of states, memory consumption and verification time for increasing model and LTL formula sizes.

Figure 3 (right) shows that the state space is linear in the number of belts (up to 4288 states), while the memory footprint increases linearly (up to 131.86 MB). The verification time is between 20 and 70 ms.

8 Conclusion and Future Work

We presented SaVeWoT, an approach to formally verify that WoT systems fulfill requirements that specify desired or undesired physical effects. The main components of SaVeWoT are a model of the physical world based on SOSA/SSN and WoT, which enables expressing requirements on physical effects in LTL, and a language that can be translated to models and deployments of WoT systems. We evaluate our approach based on the model checker Spin on consumer hardware. Our evaluation shows that SaVeWoT is feasible for verifying large WoT systems on consumer hardware. As a result, SaVeWoT can be used to check whether various subsystems, when integrated, exhibit the intended effects on the physical world.

Because we inherit various limitations from translating to Spin, we plan to explore different tools like Uppaal, nuXmv or LTL satisfiability checking tools like Black [39] in the future. We expect these tools to provide better performance because they rely on modern and performant satisfiability solvers. Further, we aim to integrate event affordances into SaVeWoT and want to incorporate graph-structured process representations [40] in the future.

Acknowledgments. This work was funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) through the Antrieb 4.0 project (Grant No. 13IK015B).

References

1. Thuluva, A.S., Anicic, D., Rudolph, S., Adikari, M.: Semantic Node-RED for rapid development of interoperable industrial IoT applications. Semantic Web **11**(6), 949–975 (2020), https://doi.org/10.3233/SW-200405
2. Ramanathan, G., Husmann, M., Mayer, S.: Interoperability vs. Tradition: Benefits and Challenges of Web of Things in Building Automation. In: IoT '21: 11th International Conference on the Internet of Things, November 8 - 12, 2021. pp. 57–63. ACM (2021), https://doi.org/10.1145/3494322.3494330
3. Lin, S., Manerkar, Y.A., Lohstroh, M., Polgreen, E., Yu, S.J., Jerad, C., Lee, E.A., Seshia, S.A.: Towards Building Verifiable CPS Using Lingua Franca. ACM Trans. Embed. Comput. Syst. **22**(5s) (sep 2023), https://doi.org/10.1145/3609134
4. Krishna, A., Le Pallec, M., Mateescu, R., Salaün, G.: Design and Deployment of Expressive and Correct Web of Things Applications. ACM Trans. Internet Things **3**(1) (oct 2021), https://doi.org/10.1145/3475964
5. Kis, Z., Peintner, D., Aguzzi, C., Hund, J., Nimura, K.: Web of Things (WoT) Scripting API. Working group note, W3C (Nov 2020), https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/
6. Cena, F., Haller, A., Lefrançois, M.: Semantics in the Edge: Sensors and actuators in the Web of Linked Data and Things. Semantic Web **11**(4), 571–580 (2020), https://doi.org/10.3233/SW-200379
7. Käbisch, S., Kamiya, T., McCool, M., Charpenay, V., Kovatsch, M.: Web of Things (WoT) Thing Description. Recommendation, W3C (Apr 2020), https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/
8. Haller, A., Janowicz, K., Cox, S.J., Lefrançois, M., Taylor, K., Le Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R., Stadler, C.: The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. Semantic Web **10**(1), 9–32 (2019)
9. Freund, M., Fries, J., Dorsch, R., Schiller, P., Harth, A.: Wot2pod: An architecture enabling an edge-to-cloud continuum. In: Proceedings of the 13th International Conference on the Internet of Things. p. 42–49. IoT '23, Association for Computing Machinery, New York, NY, USA (2024), https://doi.org/10.1145/3627050.3627063
10. Kovatsch, M., Matsukura, R., Lagally, M., Kawaguchi, T., Toumura, K., Kajimoto, K.: Web of Things (WoT) Architecture. Recommendation, W3C (Apr 2020), https://www.w3.org/TR/2020/REC-wot-architecture-20200409/
11. Giacomo, G.D., Masellis, R.D., Montali, M.: Reasoning on LTL on Finite Traces: Insensitivity to Infiniteness. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada. pp. 1027–1033. AAAI Press (2014), https://doi.org/10.1609/aaai.v28i1.8872
12. Korkan, E., Kaebisch, S., Kovatsch, M., Steinhorst, S.: Safe Interoperability for Web of Things Devices and Systems, Lecture Notes in Electrical Engineering, vol. 611, pp. 47–69. Springer (2020), https://doi.org/10.1007/978-3-030-31585-6_3
13. Kast, A., Korkan, E., Käbisch, S., Steinhorst, S.: Web of Things System Description for Representation of Mashups. In: IEEE International Conference on Omni-layer Intelligent Systems, COINS 2020, August 31 - September 2, 2020. pp. 1–8 (2020), https://doi.org/10.1109/COINS49042.2020.9191677
14. Salama, F., Korkan, E., Käbisch, S., Steinhorst, S.: Towards a Behavioral Description of Cyber-Physical Systems Using the Thing Description. In: Proceedings of the 2021 Workshop on Descriptive Approaches to IoT Security, Network, and Application Configuration. p. 6–9. DAI-SNAC '21, Association for Computing Machinery, USA (2021), https://doi.org/10.1145/3488661.3494030

15. Mena, M., Criado, J., Iribarne, L., Corral, A.: Defining Interactions of WoT Servients with Causality Relations. In: Proceedings of the 13th International Conference on Management of Digital EcoSystems. p. 112–119. MEDES '21, Association for Computing Machinery, USA (2021), https://doi.org/10.1145/3444757.3485102
16. Salama, F., Tsirkunenko, A., Korkan, E., Käbisch, S., Steinhorst, S.: WoT-Phyng-Sim: Integrating Physics Simulations with IoT Digital Twins using the Web of Things. In: IEEE International Conference on Omni-layer Intelligent Systems, COINS 2023, July 23 - 25, 2023. pp. 1–8 (2023), https://doi.org/10.1109/COINS57856.2023.10189326
17. Hackett, F., Hosseini, S., Costa, R., Do, M., Beschastnikh, I.: Compiling Distributed System Models with PGo. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, March 25-29, 2023. pp. 159–175. ACM (2023), https://doi.org/10.1145/3575693.3575695
18. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
19. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types. Proc. ACM Program. Lang. **7**(OOPSLA1) (apr 2023), https://doi.org/10.1145/3586037
20. Adiego, B.F., Darvas, D., Tournier, J., Viñuela, E.B., Suárez, V.M.G.: Bringing automated model checking to PLC program development - a CERN case study. In: 12th International Workshop on Discrete Event Systems, WODES 2014, May 14-16, 2014. pp. 394–399. International Federation of Automatic Control (2014), https://doi.org/10.3182/20140514-3-FR-4046.00051
21. Lesi, V., Jakovljevic, Z., Pajic, M.: Reliable industrial IoT-based distributed automation. In: Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019. pp. 94–105. ACM (2019), https://doi.org/10.1145/3302505.3310072
22. Vogel-Heuser, B., Folmer, J., Frey, G., Liu, L., Hermanns, H., Hartmanns, A.: Modeling of Networked Automation Systems for simulation and model checking of time behavior. In: International Multi-Conference on Systems, Signals & Devices, SSD 2012, March 20-23, 2012. pp. 1–5. IEEE (2012), https://doi.org/10.1109/SSD.2012.6197943
23. Lewerentz, C., Lindner, T.: Formal Development of Reactive Systems: Case Study Production Cell, Lecture Notes in Computer Science, vol. 891. Springer (1995), https://doi.org/10.1007/3-540-58867-1
24. Paun, D.O., Marsha, C., Biechele, B.: Production Cell Revisited. In: Proceedings of SPIN '98 (1998)
25. Zhang, L., He, W., Martinez, J.J., Brackenbury, N., Lu, S., Ur, B.: AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In: Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. pp. 281–291. IEEE / ACM (2019), https://doi.org/10.1109/ICSE.2019.00043
26. Trimananda, R., Aqajari, S.A.H., Chuang, J., Demsky, B., Xu, G.H., Lu, S.: Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1215–1227. ESEC/FSE 2020, Association for Computing Machinery, USA (2020), https://doi.org/10.1145/3368089.3409682

27. Yu, Y., Liu, J.: TAPInspector: Safety and Liveness Verification of Concurrent Trigger-Action IoT Systems. IEEE Trans. Inf. Forensics Secur. **17**, 3773–3788 (2022), https://doi.org/10.1109/TIFS.2022.3214084
28. Kashaf, A., Sekar, V., Agarwal, Y.: Protecting Smart Homes from Unintended Application Actions. In: 13th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2022, Milano, Italy, May 4-6, 2022. pp. 270–281 (2022), https://doi.org/10.1109/ICCPS54341.2022.00031
29. Alhanahnah, M., Stevens, C., Chen, B., Yan, Q., Bagheri, H.: IoTCom: Dissecting Interaction Threats in IoT Systems. IEEE Transactions on Software Engineering **49**(4), 1523–1539 (2023). https://doi.org/10.1109/TSE.2022.3179294
30. Charpenay, V., Käbisch, S.: On Modeling the Physical World as a Collection of Things: The W3C Thing Description Ontology. In: The Semantic Web - 17th International Conference, ESWC 2020, May 31-June 4, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12123, pp. 599–615. Springer (2020), https://doi.org/10.1007/978-3-030-49461-2_35
31. Brand, D., Zafiropulo, P.: On Communicating Finite-State Machines. Journal of the ACM **30**(2), 323–342 (apr 1983), https://doi.org/10.1145/322374.322380
32. Biere, A., Artho, C., Schuppan, V.: Liveness Checking as Safety Checking. Electronic Notes in Theoretical Computer Science **66**(2), 160–177 (2002), https://doi.org/10.1016/S1571-0661(04)80410-9, fMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems
33. Greenman, B., Saarinen, S., Nelson, T., Krishnamurthi, S.: Little tricky logic: Misconceptions in the understanding of LTL. Art Sci. Eng. Program. **7**(2) (2023), https://doi.org/10.22152/programming-journal.org/2023/7/7
34. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-State Verification. In: Proceedings of the 21st International Conference on Software Engineering. p. 411–420. ICSE '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/302405.302672, https://doi.org/10.1145/302405.302672
35. Holzmann, G.J.: The SPIN Model Checker - Primer and reference manual. Addison-Wesley (2004)
36. Fu, X., Bultan, T., Su, J.: Analysis of interacting BPEL web services. In: Proceedings of the 13th international conference on World Wide Web, WWW 2004, New York, NY, USA, May 17-20, 2004. pp. 621–630. ACM (2004), https://doi.org/10.1145/988672.988756
37. Loring, M.C., Marron, M., Leijen, D.: Semantics of Asynchronous JavaScript. In: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages. p. 51–62. DLS 2017, Association for Computing Machinery, USA (2017), https://doi.org/10.1145/3133841.3133846
38. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press (2008)
39. Geatti, L., Gigante, N., Montanari, A.: BLACK: A Fast, Flexible and Reliable LTL Satisfiability Checker. In: Proceedings of the 3rd Workshop on Artificial Intelligence and Formal Verification, Logic, Automata, and Synthesis hosted by the Twelfth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2021), Padua, Italy, September 22, 2021. CEUR Workshop Proceedings, vol. 2987, pp. 7–12. CEUR-WS.org (2021), https://ceur-ws.org/Vol-2987/paper2.pdf
40. Harth, A., Käfer, T., Rula, A., Calbimonte, J.P., Kamburjan, E., Giese, M.: Towards Representing Processes and Reasoning with Process Descriptions on the Web. Transactions on Graph Data and Knowledge **2**(1), 1:1–1:32 (2024), https://doi.org/10.4230/TGDK.2.1.1