BigraphTalk: Verified Design of IoT Applications
Blair Archibald®, Min-Zheng Shieh®, Yu-Hsuan Hu, Michele Sevegnani®, and Yi-Bing Lin, Fellow, IEEE

Abstract—Graphical Internet of Things (IoT) device management platforms, such as IoTtalk, make it easy to describe interactions between IoT devices. Applications are defined by dragging-and-dropping devices and specifying how they are connected, e.g., a door sensor controlling a light. While this allows simple and rapid development, it remains possible to specify unwanted device configurations, such as using the same device to drive a motor up and down simultaneously, risking damaging the motor. We propose BigraphTalk, a verification framework for IoTtalk that utilizes formal techniques, based on bigraphs, to statically guarantee that unwanted configurations do not arise. In particular, we check for invalid connections between devices, as well as type errors, e.g., passing a float to a Boolean switch. To the best of our knowledge, BigraphTalk is the first platform to support the graphical specification of correct-by-design IoT applications. BigraphTalk provides fully automated verification and feedback without end-users ever needing to specify a bigraph. This means that any application, specifiable in IoTtalk, is guaranteed, so long as verification succeeds, not to violate the given configuration constraints when deployed; with no extra cost to the user.

Index Terms—Application platform, bigraphs, device management, model verification.

I. INTRODUCTION

THE Internet of Things (IoT) combines sensors, actuators, and heterogeneous computing systems with the existing Internet infrastructure [1], [2]. Unfortunately, creating IoT applications can be difficult, often relying on detailed knowledge of low-level communication protocols [3]. Device integration and management systems [4]–[6] abstract over low-level protocols and are essential to allow both novice and advanced users to benefit from the increasing availability of IoT hardware. Several IoT solutions have been used to implement smart applications for a range of domains, including home automation [7], agriculture [8], aquarium management [9], smart campuses [10], entertainment [11], art [12], and more. While the IoT approaches in [7]–[12] allow complicated applications to be developed, they provide limited guarantees on application correctness.

Graphical IoT development provides an intuitive method for application developers to describe the links, e.g., the dataflow between IoT devices. For example, the IoTtalk [6] graphical user interface (GUI) describes the relationship between sensors and actuators graphically, allowing simple data transfer and transforms to occur. This approach is similar to other model-driven engineering methods that allow structural aspects of applications to be described [13]. Here, we focus on one existing tool—IoTtalk—that is specialized to IoT applications.

At the heart of IoTtalk is a Web-based GUI, shown in Fig. 1(a), that allows the users to drag-and-drop devices, e.g., smartphone and curtain, each containing a set of input and output device features (ODFs), e.g., acceleration, into a workspace. Device features (DFs) can then be graphically linked via joins—that implement data transformation and decision logic—to create an application. Other GUIs for IoT, e.g., WuKong [14] and Node-RED [15], describe IoT applications using a similar network-based representation.

While IoTtalk allows the development of a wide range of applications, it often allows too much flexibility; making it possible to connect two devices that should never have been connected, while providing limited guarantees of their behavior at deployment. For example, in Fig. 1(a), we try to simultaneously run the curtain up, down, and stop it. As each actuator receives the same value, they will attempt to drive the motor in different directions potentially leading to a hardware damage. We call such errors a forbidden configuration. Forbidden configurations have been observed in practice—usually due to a lack of domain knowledge about specific devices—and can cause incorrect or inefficient applications, as well as potential hardware damage.

Another common error that has been observed in practice is badly typed joins. For example, in Fig. 1(a), for Join 1 to be valid, it must convert the floating-point accelerometer values to a Boolean for use in the curtain motor switches. If the conversion is not performed, then we have a typechecking error. By removing the typechecking errors, we avoid undefined behavior at deployment.

To stop users creating invalid configurations of devices, we propose a formal verification approach for IoTtalk that guarantees the correctness, i.e., the absence of forbidden configurations and typechecking errors, of application deployments. While these two errors are some of the most commonly seen errors, in practice, we aim for an extensible approach
that allows additional errors to be verified in the future (see Section VII). In particular, the theory of bigraphs [16] makes use of the graphical placement of objects; giving it an almost one-to-one correspondence with IoTtalk as highlighted by Fig. 1. This allows interdisciplinary dialog to take place between the experts in formal methods and those in IoT. We choose bigraphs due to this almost one-to-one correspondence with the user interface, and the graphical nature of bigraphs allowing them to be easily understood by the users (for debugging etc.) without requiring, for example, knowledge of first-order logic or other notation heavy mathematical techniques. By performing an automatic translation between an IoTtalk application and corresponding formal model, end-users benefit from improved confidence in their applications without extra cost. As far as we are aware, this is the first coupling of formal methods and graphical device management frameworks for IoT.

Bigraphs are a universal computational model, defined by Milner [16], for modeling interacting systems that evolve in time and space and have been applied to model a wide range of systems, e.g., IoT/edge systems [17]–[19], MixedReality systems [20], context-aware systems [21], networking [22], [23], and security of cyber-physical systems [24], [25]. Relationships between entities, e.g., devices, are specified using both the spatial arrangement of nodes and (hyper-)links between them. Although existing tools, e.g., those based on UML [26], have basic support to, for example, express the safe connection of components, bigraphs are an expressive computational model open to extension, e.g., to express both forbidden configurations and typechecking errors in a single model, and provide an intuitive graphical notation.

We formulate forbidden configurations as static predicate checks based on bigraph patterns, which were introduced in [20] and implemented in BigraphER [27], a suite of open-source tools that provide support for specification and verification of bigraphs. Bigraph models directly reflect the IoTtalk GUI, while allowing the detection of both forbidden configurations, e.g., Fig. 1(a), and typechecking errors.

While we show how to apply bigraphs specifically to the IoTtalk platform, similar bigraph models could be applied to other graphical IoT platforms, such as Node-RED [15], and to the wider field of user interface modeling and HCI [28].

We make the following research contributions.

1) We describe the first application of formal methods to IoT graphical device management platforms.

2) We extend IoTtalk to allow specification of forbidden configurations between DFs, as well as type information for both DFs and joins.

3) We develop a bigraph model for IoTtalk applications, allowing the presence of forbidden configurations and typechecking errors to be detected statically.

4) We describe, implement, and evaluate BigraphTalk, a tool that automatically translates an IoTtalk application to the equivalent bigraph model and checks it. This allows users without knowledge of formal methods to specify correct-by-design IoT applications.

The article is organized as follows. Section II describes the IoTtalk platform for building IoT systems by linking a series of input and output devices. Section III gives an overview of bigraphs for formally verifying systems. Section IV details the conversion from an IoTtalk application to an equivalent bigraph model. We show how the two main safety properties—finding forbidden configurations and type safety—are encoded as bigraph predicates. Section V focuses on the BigraphTalk implementation, describing how we go from a user requesting verification to a result being displayed. Section VI evaluates the performance of BigraphTalk on both synthetic and real-world IoT applications. Section VII discusses the approach and suggests possible extensions to BigraphTalk, and we conclude this article in Section VIII.

II. IoTtalk

IoTtalk [6] is an application-layer IoT device management platform that provides connectivity between various devices, including a broad range of environmental sensors, home appliances, vehicle trackers, mobile phones, etc. IoTtalk allows the users to configure data interaction among devices to define new applications quickly and without knowledge of low-level network protocols, such as Bluetooth or ZigBee, etc.
Devices connect to the IoTtalk platform using a software, known as a device application (DA), that are typically installed either on an IoT gateway or integrated within the device. Many DAs are readily available [29], [30], and the developers may implement new DAs for their own devices.

An example of IoTtalk smart home application is shown in Fig. 2. Here, an air conditioner and curtain within the home are controlled based on both the value of a temperature sensor (positioned within the air conditioner) and mobile phone data. DAs [see Fig. 2(4), (5), and (6)] are connected to the IoTtalk engine [see Fig. 2(2)] that is a part of the IoTtalk server [see Fig. 2(1)]. The IoTtalk server consists of a GUI [see Fig. 2(3)] for specifying how devices should be connected, while the IoTtalk engine performs the data shepherding between devices.

In the IoTtalk platform, a device is a particular instance of a device model, e.g., representing a smartphone. A device model consists of one or more DFs, where a DF specifies a particular input or output capability of a device. We call them input DFs (IDFs) and ODFs, respectively. For example, in the smart home system, the AirCondr DA [see Fig. 2(4)] has two IDFs—IndrTemp and OutdrTemp—the indoor and the outdoor temperature sensors, and three ODFs—Temp, Speed, and Switch—that control the temperature, the speed, and the ON/OFF switches.

To make a new application, a user uses the GUI to connect IDFs to ODFs at joins. Then they select a predefined function or define a new function, in Python, for each join. This allows, for instance, input values to be averaged. An example of using the GUI to create an application is shown in Fig. 3. In the GUI, IDFs of the same device are grouped on the left, while ODFs are grouped on the right. That is, there is a single air conditioner in this application with input and outputs separated. The application reads indoor temperature, outdoor temperature, and user’s location and uses this to calculate the actuation parameters for the air conditioner. Join 3 determines when to turn on/off the air conditioner, e.g., when the user is nearby, join 2 computes the required fan speed based on the current temperature (a control loop), and join 1 sets the required temperature. In each case, the joins hide a specific join function that implements the decision-making logic. Creating applications in this manner is both quick and flexible and has been used to successfully implement many IoT applications [31], [32].

Upon receiving new values from IDFs, the IoTtalk engine computes the new values for the ODFs that are connected through the same join. This allows the corresponding DA to update the actual device with the new ODF values. When the IoTtalk engine receives new values of IDFs, it computes new values for ODFs connected to the same join. Each ODF connected to the same join receives the same value. The corresponding DA updates the values of ODFs on the device.

Many applications have sets of ODFs that should not be set to the same values or configuration simultaneously. We call these as forbidden configurations. For example, the application shown in Fig. 1(a) should be forbidden as we cannot wind the curtain up, down, and stop it at the same time.

ODFs involved in a forbidden configuration need not be part of a single device and some forbidden configurations may involve multiple devices. For example, in the same room, it is unreasonable to turn on the heater and the cooler at the same time, and we should therefore also ban the configuration in Fig. 4.

Devices are numbered from 1 to n and ODFs in a forbidden configuration are indicated by a triple consisting of the device number, the type of the device, and the ODF identifier. For instance, all the ODFs in the forbidden configuration in Fig. 5(a) have the same device number, since they are contained in the same device (e.g., curtain), while in the forbidden configuration of Fig. 4, the ODFs have different device numbers (e.g., cooler and heater).

As forbidden configurations occur based on the linking between devices, they can be statically detected before execution, e.g., down and up motor controls in Fig. 5(a). We must also check the multipath constraints, for example, in Fig. 5(b), as Join 1 and Join 2 may generate the same output at some point. For example, if Join 1 and Join 2 use the same join function, Fig. 5(b) becomes equivalent to Fig. 5(a). However, in general, multipath constraints are not necessarily erroneous as the join logic might preclude incorrect configurations. The DFs in Fig. 5(c) are in two distinct curtains and, unlike in Fig. 5(a), should not be forbidden.

III. BIGRAPHs

Bigraphs are a universal mathematical model, introduced by Milner [16], for representing the spatial configuration of...
physical or virtual entities and their interactions. Spatial relationships are specified by nesting one entity within or beside another, while nonspatial interactions are specified as hyperlinks between entities. Each entity is assigned a type which determines its arity, i.e., number of links, and whether it is atomic, meaning it cannot contain other nodes.

Bigraphs can be described algebraically or as an equivalent graphical representation. We focus on the graphical representation here due to the strong relationship with the IoTtalk graphical interface. An example of bigraph is given in Fig. 6(a). Entities are drawn as labeled shapes—e.g., A, B, and C—and represent different components of the system. In the diagrams, we often use shapes and colors to denote entity types in order to reduce the number of textual type labels shown.

Relationships between entities can be described spatially by placing an entity within or beside another. A region is indicated by a dashed rectangle and represents a logical partition of space. In Section IV, we show how regions can be used to allow a separation of modeling concerns between detecting forbidden configurations and checking types. Gray rectangles, such as in Fig. 6(b), denote sites that indicate parts of the model that have been abstracted away, i.e., a nonspecified bigraph may occur there, including the empty bigraph.

Connectivity is specified by green hyperlinks. Links may only be partially specified, in which case they connect a name, e.g., x—usually drawn above the bigraph—or are closed and not connected to anything, e.g., the link of the right-most B entity. Similar entities always have the same number of links, e.g., both A and B have one link in all cases. As all links are hyperlinks, a single link may connect multiple entities such as all the A entities in Fig. 6(a).

To check the correctness, domain-specific predicates are specified as bigraphs [20]. A matching routine then discovers if a predicate exists within another bigraph. That is, we specify what an invalid configuration of entities looks like and check these against a given input model.

An example predicate is shown in Fig. 6(b). This predicate looks for any link involving at least two A entities, possibly located in two distinct spatial regions of the system, while allowing arbitrary bigraphs to be nested within the two A entities.

The result of checking this predicate against Fig. 6(a) is in Fig. 6(c). Notice that due to the site, the left-most figure still matches regardless of the C nested within the A and that the two right-most A entities also match even though they share a top-level region. This comes from how bigraphs compose. Intuitively, we could remove both A entities from the rightmost B and treat it as having two sites. The two regions of the predicate could then replace both sites, allowing the match to occur.

Specifying predicates as bigraphs allows them to be easily understood by the end-users, who require no knowledge for formal logics, and is sufficient for our analysis. We note that more extensive logical properties of bigraphs can be expressed in the full-blown spatial logic BiLog [33].

IV. MODELING IOTTALK WITH BIGRAPHS

We define an encoding of IoTtalk applications using bigraphs. The encoding details high-level entities, such as devices, as well as predicates describing invalid user-specified applications. Such applications are automatically translated to a specific instantiation of the bigraph model, allowing them to be checked for correctness.

While there are many properties we may want to reason about, we focus on the following two high-level properties.

1) Forbidden Configurations: Are we allowed to connect two (or more) DFs together? For example, we should disallow Fig. 1(a) as simultaneously attempting to run the motor in three directions risks burning out the motor.

2) Typechecking Errors: DFs (and join functions) have specific input/output type (e.g., float, Boolean, etc.) Connections should be checked to ensure they make sense with respect to the types, for example, it is unclear what it means to (directly) connect an accelerometer outputting (x, y, z) to a Boolean switch.

The encoding mimics this separation of concerns using bigraph regions to split the model into a connection perspective for checking forbidden configurations and a typechecking perspective. Such a multiperspective approach has proved useful elsewhere [17], [20] to increase the readability and ease of extension of models, as well as highlight the potential design issues in the systems themselves.

The encoding is designed to be extensible, enabling additional properties to be added. Possible extensions are discussed in Section VII.

We begin with an informal discussion of the mapping between IoTtalk components and bigraphs, before detailing specific predicates to be checked.
A. Mapping IoTtalk to Bigraphs

The entities specified by the encoding are presented in Table I. Each entity has a fixed arity that defines the number of links, the entities it can link with, and a contained by relation that defines the placement of the entity.\(^1\) We do not explicitly check for well-formed input models, e.g., ensuring device does not contain another device, as such configurations are not specifiable in the IoTtalk user interface.

Using these entities, we show the two perspectives corresponding to Fig. 1(a) in Fig 1(b) and 7. The full bigraph model is built by joining like-names on the open links.

The bigraph model—particularly, the Connection_Perspective—closely resembles the original IoTtalk GUI, allowing it to be understood by both IoTtalk and formal method experts.

1) Connection Perspective: Each IoTtalk device corresponds to a Device entity with the Device_Model describing the type of device, e.g., a smartphone. We model specific instances of devices, i.e., one Device entity per-physical device in the system. For this purpose, devices always contain a unique Device_Id. In practice, this is often the MAC address of the device.

Devices contain a set of either input (IDF) or output (ODF) DFs. As with the devices, each DF is assigned an identifier (DF_Id). This identifier must be unique within a single device, e.g., we disallow two DFs called switch. This allows the pair (Device_Id, DF_Id) to uniquely determine a specific DF.

DFs link to their equivalent representation in the Typechecking_Perspective—closely resembles the original IoTtalk GUI, allowing it to be understood by both IoTtalk and formal method experts.

Ports contain type information that allows basic type checking to be performed (Section IV-C). Types may include additional information such as valid ranges for Num types. Port_ids ensure correct mapping of DFs with multiple outputs to join functions expecting tuple inputs. A port may be connected to multiple other ports using different Port_pt entities.

B. Detecting Forbidden Configurations

Forbidden configurations occur when two or more output features, that should be driven independently, are connected to the same IDF. This can occur directly through a single join [Fig. 1(a)] or through multiple (join) paths [see Fig. 5(b)]. As the functionality of a join can be varied by implementing a different join function, two outputs being connected to a single input through multiple joins is not necessarily an error; but it

\(^2\)This method is often used to overcome the fixed arity (number of links) of bigraph entities (e.g., in [17]) without using additional entity types—one per arity required.
TABLE I
BIGRAPH TALK ENTITIES

<table>
<thead>
<tr>
<th>Entity</th>
<th>Arity</th>
<th>Contained By</th>
<th>Linked Entities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>0</td>
<td>Connection_Perspective</td>
<td>–</td>
<td>IoTTalk Device</td>
</tr>
<tr>
<td>Device_Id</td>
<td>0</td>
<td>Device</td>
<td>–</td>
<td>Unique device id, e.g. MAC Address</td>
</tr>
<tr>
<td>Device_Model</td>
<td>0</td>
<td>Device</td>
<td>–</td>
<td>Device model name, e.g. smartphone</td>
</tr>
<tr>
<td>IDF</td>
<td>2</td>
<td>Device</td>
<td>Join_Od</td>
<td>Input device feature</td>
</tr>
<tr>
<td>ODF</td>
<td>2</td>
<td>Device</td>
<td>Join_Od</td>
<td>Output device feature</td>
</tr>
<tr>
<td>Ind</td>
<td>1</td>
<td>ODF</td>
<td>Ind</td>
<td>Independence link end</td>
</tr>
<tr>
<td>DF_Id</td>
<td>0</td>
<td>IDF/ODF</td>
<td>–</td>
<td>Device feature name, e.g. accelerometer</td>
</tr>
<tr>
<td>Join</td>
<td>1</td>
<td>Connection_Perspective</td>
<td>Join_Od</td>
<td>IoTTalk Join</td>
</tr>
<tr>
<td>Join_Id</td>
<td>0</td>
<td>Join</td>
<td>–</td>
<td>Unique join identifier</td>
</tr>
<tr>
<td>Join_pl</td>
<td>1</td>
<td>Join</td>
<td>ODF/ODF</td>
<td>Join connection point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Typechecking_Perspective</th>
<th>Arity</th>
<th>Contained By</th>
<th>Linked Entities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF</td>
<td>1</td>
<td>Typechecking_Perspective</td>
<td>ODF/ODF</td>
<td>Arbitrary device feature</td>
</tr>
<tr>
<td>Join_Fn</td>
<td>1</td>
<td>Typechecking_Perspective</td>
<td>Join</td>
<td>Join function wrapper</td>
</tr>
<tr>
<td>Join_Fn_In</td>
<td>0</td>
<td>Join_Fn</td>
<td>–</td>
<td>Function input block</td>
</tr>
<tr>
<td>Join_Fn_Out</td>
<td>0</td>
<td>Join_Fn</td>
<td>–</td>
<td>Function output block</td>
</tr>
<tr>
<td>Port</td>
<td>0</td>
<td>Join_Fn/Join_Fn_In/Join_Fn_Out</td>
<td>–</td>
<td>Typed connection wrapper</td>
</tr>
<tr>
<td>Port_Id</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Port number</td>
</tr>
<tr>
<td>Port_pl</td>
<td>1</td>
<td>Port</td>
<td>Port_pl</td>
<td>Port connection point</td>
</tr>
<tr>
<td>Bool</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Boolean type</td>
</tr>
<tr>
<td>Num</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Int/Float type</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Min val for Num type</td>
</tr>
<tr>
<td>Max</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Max val for Num type</td>
</tr>
<tr>
<td>String</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>String type</td>
</tr>
<tr>
<td>JSON</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>JSON type</td>
</tr>
<tr>
<td>Any</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Type wildcard</td>
</tr>
<tr>
<td>Missing_Args</td>
<td>0</td>
<td>Port</td>
<td>–</td>
<td>Missing argument connection point</td>
</tr>
</tbody>
</table>

Fig. 7. Typechecking perspective bigraph corresponding to Fig. 1(a).

has the potential to be one depending on the join functions. For example, the join functions of Fig. 5(b) may ensure only one motor is driven at a time if, say, join 1 always returns False.

To allow these checks to be made, IoTTalk has been extended to allow an administrator/domain expert, who is creating the device description to specify DFs that should be driven independently. Device independence is represented as interconnected Ind entities within all DFs that occur in a forbidden configuration. Independent links may occur between any DFs, even if they do not share the same device. This allows cases, such as in Fig. 4, where the cooler and heater devices should not be connected to be modeled.

Predicates for forbidden configurations are in Figs. 8 and 9. Due to the closed Ind link, these predicates match the case where exactly two DFs are independent. Similar predicates are required to check n independent devices.

To determine a forbidden configuration through a single join (Fig. 8), we match any instance where two ODFs, that should be independent, are both connected to a single join. This predicate matches regardless of the input device they are connected to. By placing the ODF’s and join in different bigraph regions, this predicate matches regardless if the DFs are within a single device or spread across multiple devices.

The predicate in Fig. 9 handles the case of where there is more than one path from a single input to two ODFs. This is similar to the predicate for a single join, however, now relies on checking if the same IDF is connected through two joins to two independent ODFs.
C. Type Safety

Each DF has a type that determines the format, and possibly the range, of input/output values from/to the DF. To ensure correctness, we check the validity of input/output type pairs as data moves through joins. This amounts to encoding a simple typing system in bigraphs. Importantly, we only check the DF interfaces against the join function interfaces. The code of the join functions is not verified.

IoTtalk supports five main types of data: 1) Booleans; 2) integers; 3) floats; 4) strings; and 5) JSON. In the encoding, we combine integers and floating point into a single Num type which reflects the implicit conversions possible in join functions. The JSON format represents an arbitrary JSON objects. When performing typechecking, we treat all data labeled JSON as the same static type JSON, without considering the runtime values. While an implicit conversion of numerical types to Booleans is possible (i.e., val = 0 implies False), we maintain Booleans as a separate type as it more accurately reflects devices such as switches.

IoTtalk has been extended to support assigning types to join functions. As join functions are written in Python, which does not feature static typing, function types are declared using a decorator as in Listing 1. The first argument of the join function decorator as in Listing 1. The first argument of the

Listing 1. Python decorator to enforce the types and ranges of function return value and arguments.

```python
@enforce_types((int, 1, 5), {float, 0.0, 1.0})
def fan_speed(\(x\)):
    return \(1 + \text{int}(4 \ast x)\)
```

To match the cases where connections are missing, we assign the port a common link name—uncon. As it is difficult to match on the absence of something within a bigraphs, i.e., that a link does not exist, we introduce an additional entity—Missing_Args—that connects to any uncon links. With this in place, finding a missing argument corresponds to matching the predicate shown in Fig. 12.
V. IMPLEMENTATION

Model verification can heavily consume computation resources and degrade the performance of other subsystems on the same machine. Rather than adding verification directly to IoTtalk, we implement the model verification as a separate tool—BigraphTalk—which interfaces to IoTtalk through a JSON API over TCP/IP (see Fig. 13). This allows us to deploy the IoTtalk server and the BigraphTalk system on different machines to provide more flexible deployment. BigraphTalk source code and a selection of examples are available online [34].

To integrate IoTtalk with BigraphTalk, we add a forbidden configuration and verification module to the IoTtalk engine. The former manages the forbidden configurations. The latter communicates with BigraphTalk directly. It composes the messages for verification requests and interprets the results from BigraphTalk. We extend the original IoTtalk GUI to provide new functions and create new tables in the database to store forbidden configurations.

To specify forbidden configurations, we add a new page to the IoTtalk GUI that allows a domain expert to create new forbidden configurations by choosing a number of devices and picking DFs involved (see Fig. 14). Based on the inputs of Fig. 14, the forbidden configuration module creates or modifies the required rows of the corresponding tables when the user saves in the IoTtalk GUI. Assume that a new forbidden configuration involves \( n \) devices and \( k \) toggled DFs.

The forbidden configuration module inserts a new row consisting of its ID number \( f \), name, and description into the ForbiddenConfiguration table. Then, it inserts \( k \) rows into the ForbiddenFeature table. Each of these rows consists of four columns as follows.

1) \( ff_id \): The primary index.
2) \( fc_id \): This DF appears in the forbidden configuration of ID \( fc_id \).
3) \( mf_id \): The ID number for retrieving the information of the DF and the associated device model.
4) \( d_idx \): The feature belongs to the \( d_idx \)-th one of the \( n \) devices involved.

This allows us to retrieve the forbidden configuration of ID number \( f \) by selecting all rows whose \( fc_id \) is \( f \) from the ForbiddenFeature table.

When a forbidden configuration is created or modified, it automatically applies to all projects using the devices. This benefits every user with improved guarantees of correctness without requiring expert device knowledge.

A user requests verification by selecting a new verify option in the IoTtalk user interface. The verification module then encodes the details of devices, joins, and connections—including any type information and forbidden configurations into a JSON message. This message is passed to the BigraphTalk system for verification. The message protocol is given in Appendix A.

BigraphTalk uses a model generator to construct an instance of the bigraph model corresponding to the IoTtalk application. The encoding includes predefined entities and predicates as described in Section IV. The conversion is fully automated and requires no input from the user. We then validate the model by matching it against the predicates. For working with bigraphs, we use the BigraphER tool [27] that provides both a textual language, based on the algebra of bigraphs, and a simulation/verification environment. There exist several good tools for bigraphs, e.g., BiGmTE [35]. We choose BigraphER as it is open source, actively maintained, and provides a library of matching routines to build upon. Crucially, it is the only tool that supports features, such as parameterized entities [e.g.,
allowing Device_Model ("Smartphone"), that are essential for our implementation. The architecture of the BigraphTalk system is in Fig. 15.

To aid debugging of IoTtalk models, BigraphTalk extracts the Device_Id, DF_Id, etc., of all devices and joins involved in an error and returns these to IoTtalk as JSON. Then, the verification module pushes the information to the GUI for display. In practice, we diverged slightly from the predicates in Section IV to make it easier to extract debugging information. The JSON message format is described in Appendix B.

The GUI reports invalid applications to the user as follows. If the verification indicates the existence of a forbidden configuration or typechecking error, then the erroneous DFs and joins are colored in red as in Fig. 16(a). For potential errors, IoTtalk warns the user by coloring the corresponding joins and DFs yellow [see Fig. 16(b)]. When multiple errors are detected, they are returned to IoTtalk in a single message. Joins or DFs appearing in both errors and warnings are colored red, e.g., Join2 in Fig. 16(c).

Importantly, the network application is verified with respect to the model generator output and we assume that the model generator constructs a correct model.

VI. EVALUATION

We evaluate BigraphTalk on a real-world application from the ArgiTalk [8] project, as well as a set of synthetic applications designed to provide worst-case analysis of the BigraphTalk system. All experiments are run on a machine with an Intel Core-i7 960 (3.2 GHz), 16-GB RAM, and Ubuntu 18.04 installed. Rather than specifying these applications in the GUI, we use a testing program that communicates directly with the IoTtalk verification module (see Fig. 13). In each case, we record the time spent in BigraphTalk, i.e., performing the bigraph encoding/matching, and the full time required to validate, i.e., including marshaling of the IoTtalk application and errors to/from JSON. In each test case, BigraphTalk performs well and accurately, which successfully detects all the errors and warnings.

Fig. 17 shows a real-world network application deployed as part of the ArgiTalk project [8]. AgriTalk provides the inexpensive smart agriculture solutions to precision soil farming. In this application, the soil sensor device SoilSensor contains sensors for electric conductivity (EC-I), moisture (Moisture-I), and pH (pH-I). The sensors in the weather station device WeatherSTA are for humidity (Humidity-I), temperature (Temperature-I), insect trap (InsectTrapper-I), rain gauge (RainGauge-I), and ultraviolet (UV-I). These sensors control three actuator devices. The Irrigation device determines the amounts of dripping (Drip-O), nitrogen ingredient (Nitrogen-O), phosphorus ingredient (Phosphorus-O), and potassium ingredient (Potassium-O). The PestSpray device
includes a switch to control spraying of biopesticides (Switch-O). The RepellentBulb device includes a switch to control repellent bulbs (Switch-O). This application consists of a similar number of devices/DFs as is common in many applications with large-scale deployments often being replicas of a base design, e.g., a field may consist of several smaller sites all running the application of Fig. 17.

The application in Fig. 17 contains one forbidden configuration error and one typechecking error. Connecting PestSpray and RepellentBulb is forbidden, as simultaneously turning on the sprayers and the repellent bulbs both reduces the effect of the biopesticides and wastes electricity, i.e., we should only run one form of repellent at a time. Using a function with three inputs for Join 1 causes a typechecking error as EC-I and pH-I provide a single float measurement each. BigraphTalk detects both errors correctly. The mean verification time over 100 runs is 2.53 s, of which BigraphTalk takes 0.86 s. Verification time is similar to that of source code compilation, and these results show that the responsiveness of verification is adequate to enable online feedback during application development.

To evaluate the scalability of BigraphTalk, we use two sets of synthetic applications designed to stress-test the verification, and these should be considered as the worst-case scenarios. The tests take a similar form to Fig. 18 with a single input device connected to two output devices that should be independent.

In general, while bigraph matching is an NP-complete problem, due to the efficiency of modern solvers, e.g., to prune large sections of the search space, we expect the time required to find an error to be (primarily) a function on the size and number of predicates (errors) we are trying to match. That is, we expect the verification time to increase primarily with the number of errors, not the size of the bigraph we are matching on.

In Fig. 19, we show the performance of BigraphTalk as we increase the number of joins (copies of join 1) in Fig. 18 from 1 to 15, where each join connects to all DFs. By connecting each join to all DFs we get an additional forbidden configuration, plus several multipath forbidden configuration errors, for each join added. As expected, the time required to verify the application increases with the number of joins and, hence, errors with even the largest test case featuring 15 joins taking only 18 s to verify. In practice, it is unlikely that there will be more than a few joins between the same DFs, giving us confidence in the scalability of verification to handle even extreme cases.

To show the effect of increasing the number of DFs, in Fig. 20, we show the performance of BigraphTalk using an application with one smartphone, five coolers, five heaters, and ten joins. In each case, we increase the number of ODFs connected to each join. When we connect all DFs to every join, there are 2500 errors of which 250 are single path forbidden configurations, and 2250 are multipath forbidden configurations. Again, BigraphTalk performs well, spending only 0.134 s per error in the worst case. In practice, we expect the number of total errors to be less than 500, allowing BigraphTalk to verify even complex applications in less than 1 min.

As expected, formal verification can consume a lot of computing resources often running with 100% CPU utilization. This is not an issue in practice given the ubiquity of multicore machines, and the ability to run BigraphTalk on a second host due to the TCP interface. Verification is only performed when requested by the user at design time and deployed IoTtalk applications are not affected by the increased CPU usage.

VII. DISCUSSION

While BigraphTalk allows fully automated verification from the perspective of an end-user, DF constraints must still be specified by an experienced user ahead of time. This is unavoidable, as it requires device domain knowledge that cannot be automatically inferred. However, like creating an IoTtalk device model itself, this is a one-off overhead per device and the device constraints may be reused in many
projects. This allows nondevice experts to conform to device constraints they may not be aware of. For smart home applications [36], we have found that DF constraints are relatively easy to specify. We are also working on the aforementioned smart agriculture applications [8] and are collecting feedback from end-users on any difficulties experienced when specifying DF constraints.

By maintaining BigraphTalk as a separate package, we may incrementally add additional features with limited disruptions to the main IoTtalk development. There are numerous ways to extend this article in the future.

1) Additional Property Analysis: Bigraphs are more flexible than, say, a type checker only approach, in that we can encode additional properties. One such property would be modeling device location, for example, allow a phone to communicate with a device only if they are in the same location, e.g., disallow turning on a fire if no one is around to supervise. Location-based properties could be handled in an additional modeling perspective such that a device linked under a location control is said to be in the location. This is shown in Fig. 21. Here, one smartphone is in Dorm 1 and another simply in the Guest_House_1. Using bigraphs with sharing [37] allows devices to be in multiple locations simultaneously, for example, to model wireless router ranges; further increasing the expressivity of the model.

Additional device information could allow, for example, device ownership and privacy properties, e.g., data within a dorm should only be accessible to the dorm resident. Their verification could follow an approach based on a bigraph encoding similar to that used for forbidden configurations and typechecking.

2) Runtime Monitoring: So far we have only utilized bigraphs to check static properties of IoTtalk applications. However, bigraphs also admit a reactive theory that describes how they evolve over time. This can be used to allow runtime monitoring [17], where a single model is maintained by BigraphTalk and events, e.g., GUI updates, are passed from IoTtalk to trigger model updates. For example, a user may add a new device, triggering a new device event that causes the device to be added into the model. At some later time, the user may then remove the device, again causing a model update.

Runtime monitoring allows a wider range of properties to be specified. Reliability of connections could be encoded by having the system send the model an event each time a connection is used. Timestamp information can then be added to the joins, and predicates can match on timestamps that have not been used in the last $t$ seconds; potentially indicating an error.

Maintaining a model at runtime not only allows checking of a wider range of properties but also by allowing changes in the model to be reflected back into the system (not just from the system to the model), we gain the ability to perform model-based adaption of the deployed system. Such self-adaptation is common in the models@runtime approach [38].

3) Model Driven User Interface Control: Currently, applications are modeled in IoTtalk and then sent to BigraphTalk for verification purposes, with BigraphTalk providing either success or a particular error with the model. Given a reactive bigraph model, such as that needed for the runtime monitoring above, we can generate a transition system that details the events that might occur in the future, e.g., a user adds a join. By checking error predicates on future paths, we can determine which operations in the GUI should be disallowed to avoid forming a bad model. That is, instead of only checking an existing IoTtalk application for correctness, stop the user being able to create invalid models.

VIII. Conclusion

IoT integration and management platforms, such as IoTTalk, are essential to allow nonexpert users to design coherent and usable Internet-enabled systems. Although these tools have proved useful in practice, they often have limited capabilities for providing guarantees on the correctness of the designed application. We have shown how bigraphs have been successfully applied to specify correct-by-design IoT applications in the IoTTalk platform so that they are free from forbidden configurations and typechecking errors. The fully automatic translation from IoTtalk applications to bigraphs has been implemented and evaluated in a new tool—BigraphTalk—giving users improved confidence in the correctness of their applications without requiring knowledge of formal methods.

APPENDIX A

FORMAT OF MESSAGES TO BIGRAPH TALK

The JSON message sent to BigraphTalk is an JSON object specifying the network application and forbidden configuration to be verified. It has three attributes: 1) Device object; 2) Join_list; and 3) Forbidden_Configuration (see Listing 2).

The attribute Device_List is an array of devices which build the network application. We encode each device into a JSON using the format in Listing 3. Device_ID is a unique identifier for the device in the IoTTalk system. Device_Model is a string representing the name of the model of the device. DF_list is an array of the features of the device used in the network application.

We describe the DFs with the following format. If the df_parameter does not have maximum or minimum value, we just omit the corresponding attribute.

The attribute Join_list in Listing 6 is an array of joins which connect the devices in the network application. The
information of joins are encoded into the format in Listing 4. Each join has a unique Join_Id and a function described by a JSON object. A join function has a unique identifier and the description for its input and output. We use arrays of DFs to denote the connection of the join.

The attribute Forbidden_Configuration in Listing 6 is an array of forbidden configurations which only involve the device models in the network application. We use the format in Listing 5 to describe them. Each element in the array Data designates a particular DF. If a forbidden configuration involves the same DF of several distinct devices, we assign different Device_Id's to them. If a forbidden configuration involves several DFs of the same device, then we assign the same Device_Id to the corresponding DFs.

### APPENDIX B

**FORMAT OF MESSAGES TO IOTTALK**

BigraphTalk sends the errors as an array like Listing 7. If there is no error, then IoTtalk will receive an empty array. Each error is encoded as a JSON object. Currently, error_type is one of “Forbidden Configuration,” “Possible Forbidden Configuration,” “Type mismatch,” and “Missing argument.” The detailed text information is given by error_msg. Each element of the array data indicates the problematic connection with the corresponding join and DF.

### REFERENCES


