BigraphTalk: Verified Design of IoT Applications

Blair Archibald, Min-Zheng Shieh, Yu-Hsuan Hu, Michele Sevegnani, Yi-Bing Lin *Fellow, IEEE*

***Abstract*—Graphical 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.**

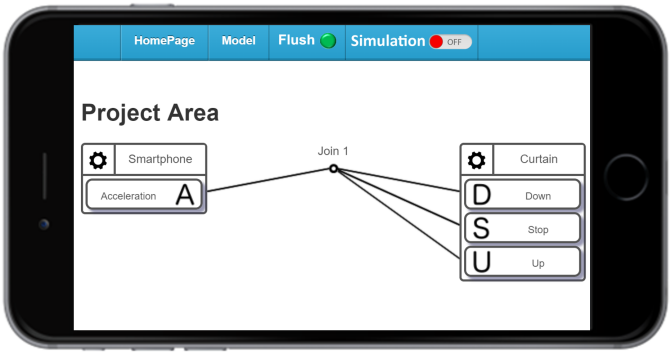
We propose BigraphTalk, a verification framework for IoTtalk that utilizes formal techniques, based on bigraphs, to statically *guarantee* that unwanted configurations do not arise. In particular, we check for invalid connections between devices, as well as type errors, *e.g.* passing a `float` to a `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 *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.

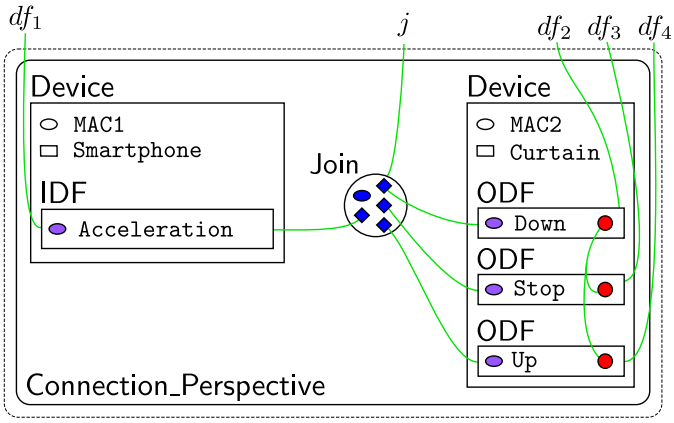
***Index Terms*—device management, application platform, bigraphs, model verification**

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.



(a) IoTtalk application with two devices, *Smartphone* and *Curtain*, and four device features, *Acceleration*, *Down*, *Up*, and *Stop*. Data flows left-to-right through *Join 1* and each device feature receives the same value.



(b) Equivalent bigraph. The link connecting the three red circles represents a forbidden configuration involving the corresponding device features. Open links at the top of the diagram are used to type check the join. Details will be elaborated in Section IV.

Fig. 1: Relationship between IoTtalk applications and bigraph model.

Graphical IoT development provides an intuitive method for application developers to describe the links, *e.g.* the data flow 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. 1a, that allows users to drag-and-drop devices, *e.g.* *Smartphone* and *Curtain*, each containing a set of input and output *device features*, *e.g.* *Acceleration*, into a workspace.

B. Archibald and M. Sevegnani are in the School of Computing Science, University of Glasgow, UK. E-mail: {blair.archibald, michele.sevegnani}@glasgow.ac.uk

M.-Z. Shieh is in the Information Technology Service Center, National Chiao Tung University, Hsinchu, Taiwan. E-mail: mzshieh@nctu.edu.tw

Y.-H. Hu and Y.-B. Lin are in the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. E-mail: {yuxuan.cs07g,liny}@nctu.edu.tw

This work was supported in part by the RSE International Exchange Programme: RSE MOST Joint Project (MOST106-2911-I-009-508), by the Engineering and Physical Sciences Research Council grant S4: Science of Sensor Systems Software (EP/N007565/1), by the Center for Open Intelligent Connectivity from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education in Taiwan, by Ministry of Science and Technology 108-2221-E-009-047 and by Ministry of Economic Affairs 107-EC-17-A-02-S5-007.

Device features 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 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. 1a, we try to simultaneously run the curtain up, down and stop it. As each actuator receives the same value, they will attempt drive the motor in different directions potentially leading to 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. 1a, 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 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 in practice, we aim for an extensible approach that allows additional errors to be verified in 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 dialogue 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 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. 1a, 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 device features, as well as type information for both device features 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 paper 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 performace of BigraphTalk with both synthetic and real-world IoT applications. Section VII discusses the approach and suggests possible extensions to BigraphTalk, and we conclude this work 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 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 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 developers may implement new DAs for their own devices.

An example IoTtalk smart home application is shown in Fig. 2. Here, an air conditioner and curtain within the home

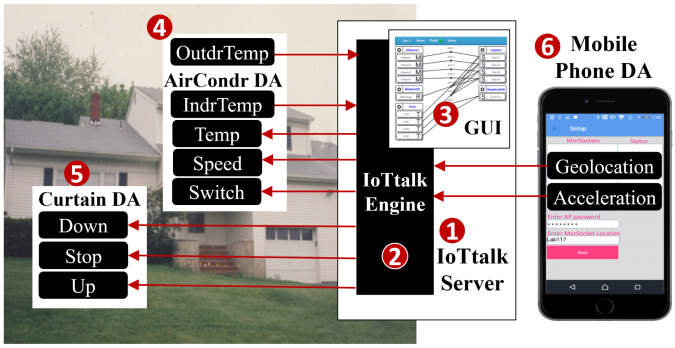


Fig. 2: The IoTtalk architecture for a smart home application.

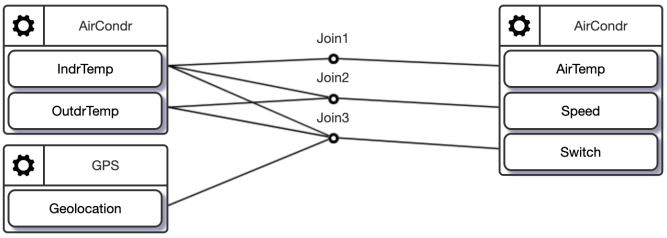


Fig. 3: An example application in the IoTTalk GUI.

are controlled based on both the value of a temperature sensor (positioned within the air conditioner) and mobile phone data.

Device applications (Fig. 2 (4), (5), (6)) are connected to the IoTtalk engine (Fig. 2 (2)) that is part of the IoTtalk server (Fig. 2 (1)). The IoTtalk server consists of a GUI (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 *device features* (DFs), where a DF specifies a particular input or output capability of a device. We call them *input device features* (IDFs) and *output device features* (ODFs) respectively. For example, in the smart home system, the AirCondr DA (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*. They then 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 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

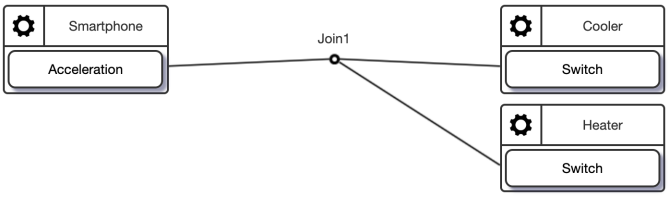


Fig. 4: A forbidden configuration involving multiple devices.

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 *forbidden configurations*. For example, the application shown in Fig. 1a should be forbidden as we cannot wind the curtain up, down, and stop it at the same time.

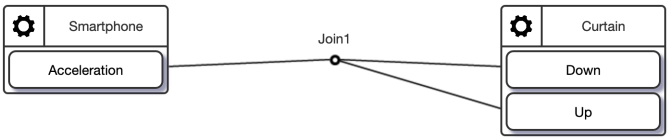
ODFs involved in a forbidden configuration need not be part of a single device, and some forbidden configurations may involve multiple devices. For example, in the same room, it is unreasonable to turn on the heater and the cooler at the same time, and we should therefore also ban the configuration in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

Devices are numbered from 1 to n and output device features in a forbidden configuration are indicated by a triple consisting of the device number, the type of the device, and the output device feature identifier. For instance, all the output device features in the forbidden configuration in Fig. 5a 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 output device features have different device numbers *e.g.* Cooler and Heater).

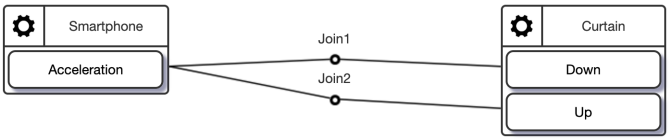
As forbidden configurations occur based on the linking between devices, they can be statically detected before execution, *e.g.* down and up motor controls in Fig. 5a. We must also check multi-path constraints, for example in Fig. 5b, as Join 1 and Join 2 *may* generate the same output at some point.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

III. Bigraphs

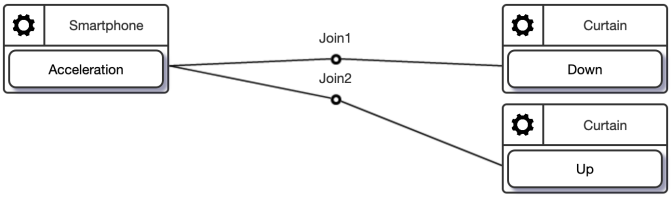
Bigraphs are a universal mathematical model, introduced by Milner [16], for representing the spatial configuration of physical or virtual entities and their interactions. Spatial relationships are specified by nesting one entity within or



(a) A forbidden configuration.



(b) An application *potentially* featuring a forbidden configuration depending on the join functions.



(c) An application not featuring any forbidden configuration as there are two different curtain devices.

Fig. 5: Possible configurations driving smart curtains.

beside another, while non-spatial interactions are specified as hyper-links 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 bigraph is in Fig. 6a. Entities are drawn as labeled shapes – *e.g.* A, B, 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. Grey rectangles, such as in Fig. 6b denote *sites* that indicate parts of the model that have been abstracted away, *i.e.* a non-specified bigraph may occur there, including the empty bigraph.

Connectivity is specified by green *hyper-links*. Links may be only 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 hyper-links, a single link may connect multiple entities such as all the A entities in Fig. 6a.

To check 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. 6b. 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. 6a is in Fig. 6c. Notice, 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 right-most 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 end-users, who require no knowledge for formal logics, and is sufficient for our analysis. We note more extensive logical properties of bigraphs can be expressed in the full-blown spatial logic BiLog [33].

IV. Modelling 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.

- **Forbidden configurations:** Are we allowed to connect two (or more) device features together? For example, we should disallow Fig. 1a as simultaneously attempting to run the motor in three directions risks burning out the motor.
- **Typechecking errors:** Device features (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 $\langle x, y, z \rangle$ 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 multi-perspective approach has proved useful elsewhere [17], [20] to increase the readability and ease of extension of models, as well as highlight 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.

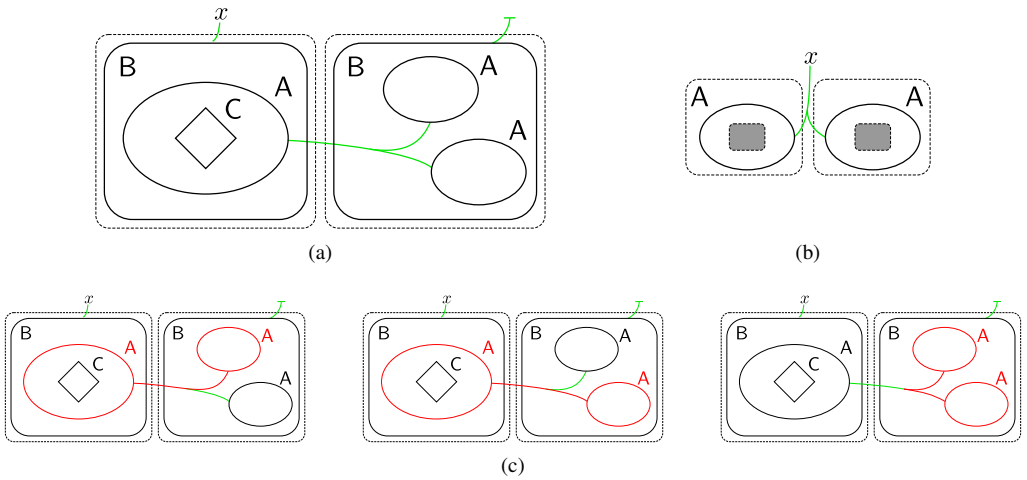


Fig. 6: (a) Example bigraph with entities A, B, and C; (b) example predicate; (c) occurrences of the predicate (highlighted in red) in the example bigraph.

TABLE I: BigraphTalk entities.

Entity		Arity	Contained By	Linked Entities	Description
			Connection_Perspective		
Device		0	Connection_Perspective	–	IoTtalk Device
Device_Id(id)	⬭	0	Device	–	Unique device id, *e.g.* MAC Address
Device_Model(mdl)	▭	0	Device	–	Device model name, *e.g.* smartphone
IDF		2	Device	Join_pt, DF	Input device feature
ODF		2	Device	Join_pt, DF	Output device feature
Ind	●	1	ODF	Ind	Independence link end
DF_Id(id)	⬭	0	IDF/ODF	–	Device feature name, *e.g.* accelerometer
Join		1	Connection_Perspective	Join_Fn	IoTtalk Join
Join_Id(id)	⬭	0	Join	–	Unique join identifier
Join_pt	◆	1	Join	ODF/IDF	Join connection point
			Typechecking_Perspective		
DF		1	Typechecking_Perspective	ODF/IDF	Arbitrary device feature
Join_Fn		1	Typechecking_Perspective	Join	Join function wrapper
Join_Fn_Ins		0	Join_Fn	–	Function input block
Join_Fn_Outs		0	Join_Fn	–	Function output block
Port		0	Join_Fn/Join_Fun_Ins/Join_Fun_Outs	–	Typed connection wrapper
Port_Id(num)	⬭	0	Port	–	Port number
Port_pt	◆	1	Port	Port_pt	Port connection point
Bool	▬	0	Port	–	Boolean type
Num	▬	0	Port	–	Int/Float type
Min(x)	◀	0	Port	–	Min val for Num type
Max(x)	▶	0	Port	–	Max val for Num type
String		0	Port	–	String type
JSON		0	Port	–	JSON type
Any		0	Port	–	Type wildcard
Missing_Args	☆	0	Port	–	Missing argument connection point

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. 1a in Fig. 1b and Fig. 7. The full bigraph model is built by joining like-names on the open links.

[1]This defines a *sorting* scheme for the bigraphs (see [16, C.6]).

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) device features. As with devices, each device feature is assigned an identifier (DF_Id). This identifier must be unique

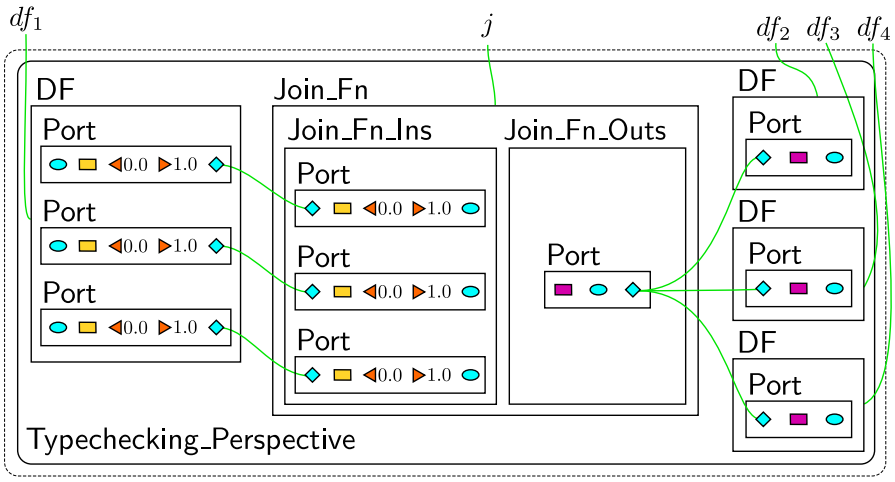


Fig. 7: Typechecking perspective bigraph corresponding to Fig. 1a.

within a single device, *e.g.* we disallow two device features called switch. This allows the pair ⟨Device_Id, DF_Id⟩ to uniquely determine a specific device feature.

Device features link to their equivalent representation in the Typechecking_Perspective as well as to any joins. As bigraphs support hyper-links, a single input/output may link to multiple joins, *e.g.* in Fig. 5b.

To detect forbidden configurations, ODFs that must not share an input (are independent) contain an additional entity Ind for each forbidden configuration they are part of. Forbidden configurations are then described by connecting Ind entities for each ODF in the configuration together (*e.g.* the red circles in Fig. 1b). The use of Ind, rather than linking directly to ODF entities, allows connections between any number of ODFs[2]. ODFs that feature in multiple forbidden configurations have additional Ind entities for each configuration. As bigraphs support hyper-links, a forbidden configuration can be specified between any number of ODFs.

IoTtalk joins are converted to Join entities that store a unique Join_Id alongside connections to device features. As with the Ind entities, the use of Join_pts instead of direct linking allows an IDF/ODF to be connected to any number of Joins.

2) Typechecking Perspective: The Typechecking_Perspective presents lower-level details of device features (DFs) and join functions (Join_Fns). In particular, it models the typing and input/output arity information that is not present in the Connection_Perspective. Device/DF/Join identifiers are not required in this perspective as these can be recovered from the Connection_Perspective by following the cross-perspective links.

Device features are connected to join functions using a Port entity. 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 device features with multiple outputs to join functions expecting tuple inputs. A port may be connected to multiple other ports using different Port_pt entities.

[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.

B. Detecting Forbidden Configurations

Forbidden configurations occur when two or more output features, that should be driven independently, are connected to the same input device feature. This can occur directly, through a single join (Fig. 1a), or through multiple (join) paths (Fig. 5b). 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 has the potential to be one depending on the join functions. For example, the join functions of Fig. 5b 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 device features that should be driven *independently*. Device independence is represented as interconnected Ind entities within all device features that occur in a forbidden configuration. Independence links may occur between *any* device features, 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 Fig. 8 and Fig. 9. Due to the closed Ind link, these predicates match the case where *exactly two* device features 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 device features 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

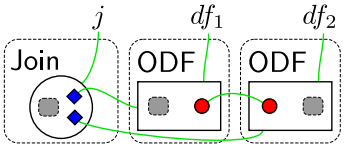


Fig. 8: Forbidden configuration predicate: two independent (link between the red circles) device features, regardless of location, share a join. Hyper-links on ODF inputs are open but omitted for clarity.

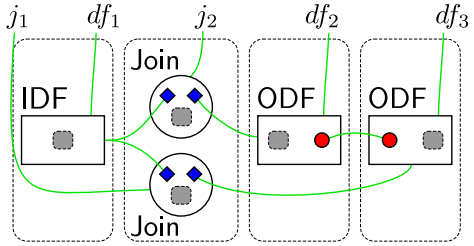


Fig. 9: Potential forbidden configuration predicate: two independent (link between the red circles) devices, regardless of location, share an input device feature. Hyper-links on IDF/ODF output/input are open but omitted for clarity.

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 device feature has a type that determines the format, and possibly the range, of input/output values from/to the device feature. 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 device feature interfaces against the join function interfaces. The *code* of the join functions is not verified.

IoTtalk supports 5 main types of data: `booleans`, `integers`, `floats`, `strings` and `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 run-time 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 function decorator as in Listing 1. The first argument of the decorator determines the type and the range of the return value, while the other arguments indicate the types and the ranges of any inputs.

IoTtalk supports composite data types to describe device features with multiple outputs, and we use tuples to represent

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

```
@enforce_types((int, 1, 5),(float, 0.0, 1.0))
def fan_speed(x):
    return 1 + int(4 * x)
```

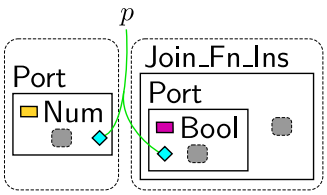


Fig. 10: Typechecking predicate: cannot connect Num to Bool.

them in the decorator. For example, the device feature `GeoLocation` outputs a pair of `floats` representing longitude and latitude. We describe this using a type such as `((float,-90.0,90.0),(float,-180.0,180.0))` in a join function decorator. Additional predicates (omitted) determine errors in correctly mapping between tuple inputs/outputs by matching when Port_Id's do not align.

In the case `None` is used to describe the data type, or when no decorator is provided, we assign ports a special type Any that never fails to typecheck. The use of Any increases the range of applications that can be modeled at the cost of reduced correctness guarantees.

Typechecking uses predicates such as in Fig. 10. This predicate looks for a mismatch in the types of two (or more) connected ports and reports this as an error. We encode data direction using the fact all connections move through a join function with separate input and output blocks. Similar predicates exist for Join_Fn_Outs. We require a predicate for all combinations of the 4 types in both input and output positions: 24 in total. No predicates are added for Any types, ensuring they pass all typechecking.

Although the encoding defines entities to model the range of numeric types, due to limited support in BigraphER for parameter comparisons, we currently do not check range correctness. This could be added in future through a predicate such as in Fig. 11. As IoTtalk allows range values to be determined using machine learning [6], this could be used to dynamically update the range parameters of the model at runtime.

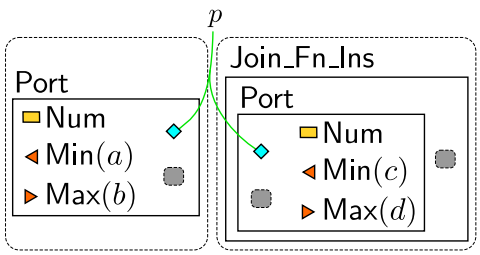


Fig. 11: Range check predicate: invalid range mapping where $[a, b] \not\subseteq [c, d]$.

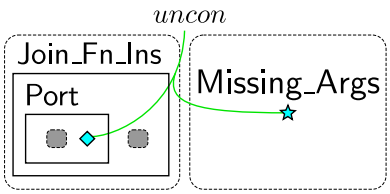


Fig. 12: Missing parameter predicate: join function argument not connected.

To ensure join functions are being used correctly, we not only check the types of the inputs/outputs, but also that no inputs/outputs are missing, *e.g.* passing only two `Num` types to a function that expects three. These checks only occur for join functions as, for example, we may wish to only connect the x component of an accelerometer to a join.

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 add 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 device features 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 a new forbidden configuration involves n devices and k toggled device features. 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.

- `ff_id`: The primary index.

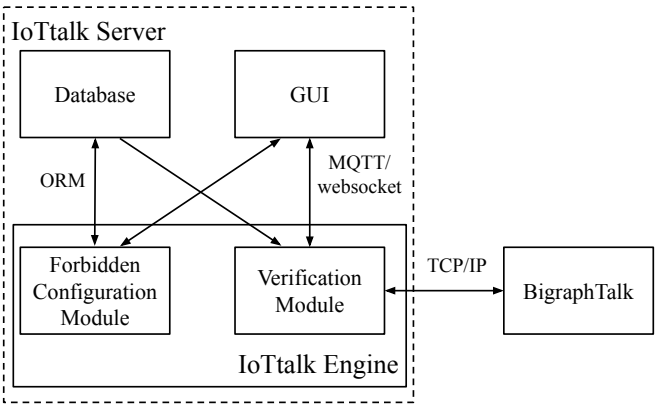
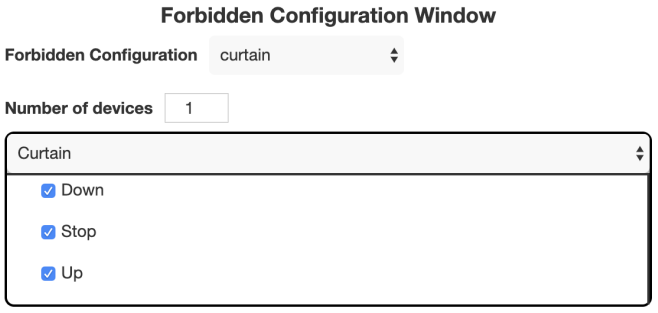
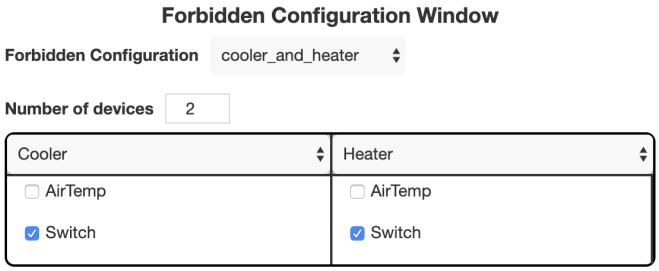


Fig. 13: Implementation overview.



(a) Forbidden configuration for Fig. 1a.



(b) Forbidden configuration for Fig. 4.

Fig. 14: Interface for specifying forbidden configurations.

- `fc_id`: This device feature appears in the forbidden configuration of ID `fc_id`.
- `mf_id`: The ID number for retrieving the information of the device feature and the associated device model.
- `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

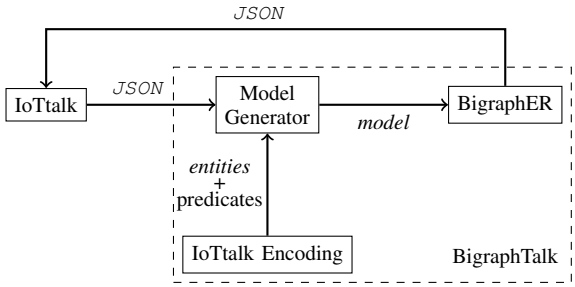


Fig. 15: BigraphTalk architecture.

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.

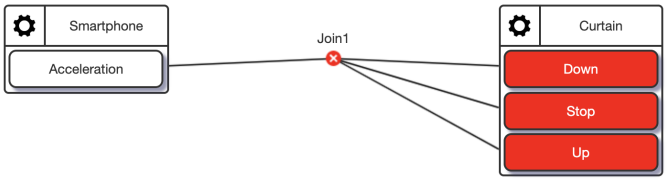
To aid debugging of IoTtalk models, BigraphTalk extracts the Device_Id, DF_Id, *etc.* of all devices and joins involved in an error, and returns these to IoTtalk as JSON. Then the verification module pushes the information to the GUI for display. In practice, we diverged slightly from the predicates 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 device features and joins are colored in red as in Fig. 16a. For potential errors, IoTtalk warns the user by coloring the corresponding joins and device features yellow (see Fig. 16b). When multiple errors are detected they are returned to IoTtalk in a single message. Joins or device features appearing in both errors and warnings are colored red, *e.g.* Join2 in Fig. 16c.

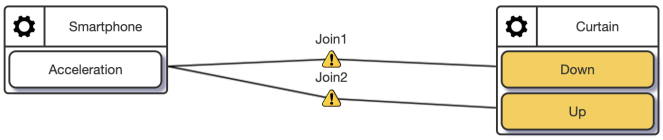
Importantly, the network application is verified with respect to the model generator output and we assume that the model generator constructs a correct model.

VI. Evaluation

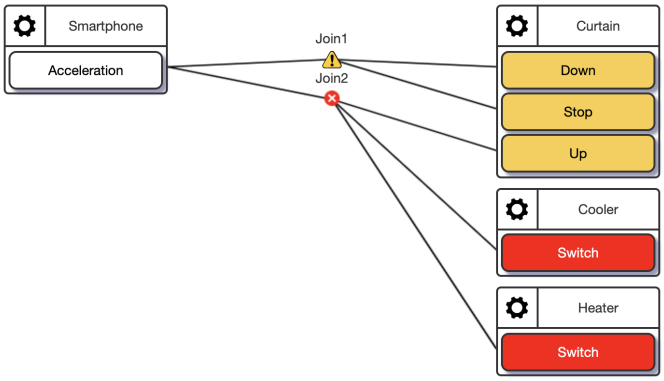
We evaluate BigraphTalk on a real-world application from the ArgiTalk [8] project, as well as a set of synthetic applications designed to provide worst-case analysis of the BigraphTalk system. All experiments are run on a machine with an Intel core-i7 960 (3.2 GHz), 16 GB RAM, and Ubuntu



(a) Forbidden configuration highlighted in red.



(b) Potential forbidden configuration highlighted in yellow.



(c) Display errors with higher priority.

Fig. 16: User interface feedback from verification results.

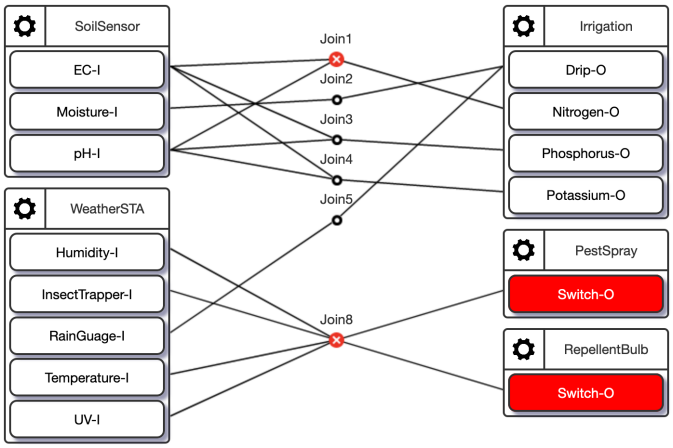


Fig. 17: The verification result of the AgriTalk example

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 successfully detects all errors and warnings.

Figure 17 shows a real-world network application deployed

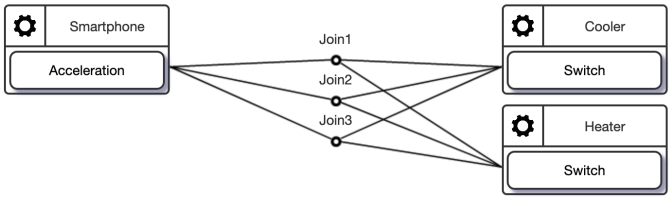


Fig. 18: A synthetic IoTtalk application for testing

as part of the ArgiTalk project [8]. AgriTalk provides inexpensive smart agriculture solutions to precision soil farming. In this application, the soil sensor device SoilSensor and the weather station device WeatherSTA control irrigation, pest control and the repellent bulbs. The soil sensor device (SoilSensor) includes the 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 trapper that reports the number of trapped bugs (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/device features 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 seconds, of which BigraphTalk takes 0.86 seconds. Verification time is similar to that of source code compilation, and these results show 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 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.

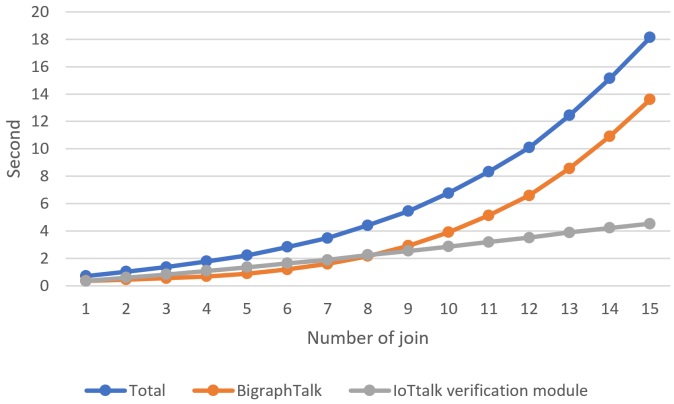


Fig. 19: Verification time as the number of joins are increased

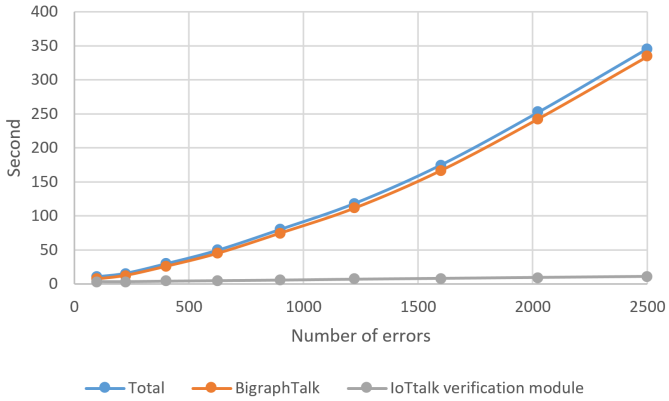


Fig. 20: Verification time for increasing numbers of errors

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 device features. By connecting each join to all device features we get an additional forbidden configuration, plus several multi-path 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 seconds to verify. In practice, it is unlikely that there will be more than a few joins between the same device features, giving us confidence in the scalability of verification to handle even extreme cases.

To show the effect of increasing the number of device features, 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 output device features connected to each join. When we connect all device features to every join, there are 2500 errors of which 250 are single path forbidden configurations, and 2250 are multi-path forbidden configurations. Again, BigraphTalk performs well, spending only 0.134 seconds 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 minute.

As expected, formal verification can consume a lot of

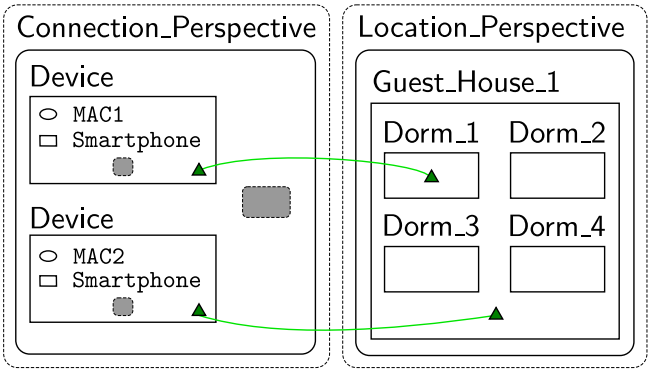


Fig. 21: Example model extension to represent locations.

computing resources often running with 100% CPU utilization. This is not an issue in practice given the ubiquity of multi-core 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, device feature 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 re-used in many projects. This allows non-device experts to conform to device constraints they may not be aware of. For smart home applications [36] we have found device feature 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 device feature 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 work in the future, including:

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, 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 non-expert 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 BigraphTalk

The JSON message sent to BigraphTalk is an JSON object specifying the network application and forbidden configuration to be verified. It has three attributes Device_List, Join_List and Forbidden_Configuration, see Listing 2.

Listing 2: JSON object sent to BigraphTalk.

```
{
    "Device_List": [device1, ...],
    "Join_List": [join1, ...],
    "Forbidden_Configuration": [fc1, ...]
}
```

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.

Listing 3: A device.

```
{
    "Device_Id": device_id,
    "Device_Model": device_model,
    "DF_List": [df1, ...]
}
```

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

Listing 4: A device feature.

```
{
    "df_name": name,
    "df_type": input_or_output,
    "df_parameter": [
        {
            "param_i": index,
            "param_type": data_type,
            "max": max_value,
            "min": min_value
        }
    ]
}
```

The attribute Join_List in Listing 2 is an array of joins which connect the devices in the network application. The information of joins are encoded into the format in Listing 5. 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 device features to denote the connection of the join.

Listing 5: A join.

```
{
    "Join_Id": join_id,
    "Join_Fn": {
        "Fn_Id": fn_id,
        "Fn_Input": [(type1,min1,max1), ...],
        "Fn_Output": (type, min, max)
    },
    "Devices": {
        "input": [idf1, ...],
        "output": [odf1, ...]]
    }
}
```

The attribute Forbidden_Configuration in Listing 2 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 device feature. If a forbidden configuration involves the same device feature of several distinct devices, we assign different Device_Id's to them. If a forbidden configuration involves several device features of the same device, then we assign the same Device_Id to the corresponding device features.

Listing 6: A forbidden configuration.

```
{
    "FC_Name": name,
    "Data": [
        {
            "Device_Id": device_id,
            "Device_Model": device_model,
            "Device_Feature": device_feature
        },
    ...]
}
```

B. Format of messages to IoTtalk

BigraphTalk sends the errors as an array like Listing 7. If there is no error, then IoTtalk will receives 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 device feature.

Listing 7: JSON array sent back to IoTtalk.

```
[
    {
        "error_type": error_type,
        "error_msg": error_msg,
        "data": [
            {
                "join_id": join_id,
                "device":{
                    "device_id": device_id,
                    "df_name": df_name
                }
            },
        ...]
    },
...]
```

REFERENCES

[1] S. Verma, Y. Kawamoto, Z. M. Fadlullah, H. Nishiyama, and N. Kato, "A survey on network methodologies for real-time analytics of massive IoT data and open research issues," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 3, pp. 1457–1477, 2017. [Online]. Available: https://doi.org/10.1109/COMST.2017.2694469

[2] T.-Y. Chan, Y. Ren, Y.-C. Tseng, and J.-C. Chen, "eHint: An efficient protocol for uploading small-size IoT data," in *2017 IEEE Wireless Communications and Networking Conference, WCNC 2017, San Francisco, CA, USA, March 19-22, 2017*, 2017, pp. 1–6.

[3] F. Corno, L. D. Russis, and J. P. Senz, "On the challenges novice programmers experience in developing IoT systems: A survey," *Journal of Systems and Software*, vol. 157, p. 110389, 2019.

[4] "Allseen alliance," accessed 2016. [Online]. Available: https://allseenalliance.org/

[5] "Standards for M2M and the Internet of Things," accessed: 2019-04-08. [Online]. Available: http://www.onem2m.org/

[6] Y.-B. Lin, Y.-W. Lin, C.-M. Huang, C.-Y. Chih, and P. Lin, "IoTtalk: A management platform for reconfigurable sensor devices," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1552–1562, Oct 2017.

[7] C.-S. Shih, J.-J. Chou, N. Reijers, and T.-W. Kuo, "Designing CPS/IoT applications for smart buildings and cities," *IET Cyber-Physical Systems: Theory Applications*, vol. 1, no. 1, pp. 3–12, 2016.

[8] W.-L. Chen, Y.-B. Lin, Y.-W. Lin, R. Chen, J.-K. Liao, F.-L. Ng, Y.-Y. Chan, Y.-C. Liu, C.-C. Wang, C.-H. Chiu, and T.-H. Yen, "AgriTalk: IoT for precision soil farming of turmeric cultivation," *IEEE Internet of Things Journal*, pp. 5209–5223, 2019.

[9] C. Dupont, P. Cousin, and S. Dupont, "IoT for aquaculture 4.0 smart and easy-to-deploy real-time water monitoring with IoT," in *2018 Global Internet of Things Summit (GIoTS)*, June 2018, pp. 1–5.

[10] X. Li, R. Lu, X. Liang, X. Shen, J. Chen, and X. Lin, "Smart community: an internet of things application," *IEEE Communications Magazine*, vol. 49, no. 11, pp. 68–75, November 2011.

[11] Y. Sulema, "Mulsemedia vs. Multimedia: State of the art and future trends," in *2016 International Conference on Systems, Signals and Image Processing (IWSSIP)*, May 2016, pp. 1–5.

[12] X. Xiao, P. Puentes, E. Ackermann, and H. Ishii, "Andantino: Teaching children piano with projected animated characters," in *Proceedings of the The 15th International Conference on Interaction Design and Children*, ser. IDC '16. New York, NY, USA: ACM, 2016, pp. 37–45.

[13] A. R. da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015. [Online]. Available: https://doi.org/10.1016/j.cl.2015.06.001

[14] C.-S. Shih, J.-J. Chou, and K.-J. Lin, "WuKong: Secure run-time environment and data-driven IoT applications for smart cities and smart buildings," *Journal of Internet Services and Information Security (JISIS)*, vol. 8, no. 2, pp. 1–17, May 2018.

[15] "Node-RED framework," accessed: 2019-04-26. [Online]. Available: https://nodered.org/

[16] R. Milner, *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[17] M. Sevegnani, M. Kabác, M. Calder, and J. A. McCann, "Modelling and verification of large-scale sensor network infrastructures," in *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, 2018*. IEEE Computer Society, 2018, pp. 71–81.

[18] H. Sahli, T. Ledoux, and É. Rutten, "Modeling Self-Adaptive Fog Systems Using Bigraphs," in *17th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems Proceedings*, Oslo, Norway, Sep. 2019.

[19] S. Marir, F. Belala, and N. Hameurlain, "A formal model for interaction specification and analysis in iot applications," in *Model and Data Engineering - 8th International Conference, MEDI 2018, Marrakesh, Morocco, October 24-26, 2018, Proceedings*, 2018, pp. 371–384. [Online]. Available: https://doi.org/10.1007/978-3-030-00856-7_25

[20] S. Benford, M. Calder, T. Rodden, and M. Sevegnani, "On lions, impala, and bigraphs: Modelling interactions in physical/virtual spaces," *ACM Transactions on Computer-Human Interaction*, vol. 23, no. 2, pp. 9:1–9:56, 2016.

[21] T. A. Cherfia, K. Barkaoui, and F. Belala, "A brs-based modeling approach for context-aware systems: A case study of smart car system," in *12th IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2014, Milano, Italy, August 26-28, 2014*, 2014, pp. 310–314. [Online]. Available: https://doi.org/10.1109/EUC.2014.53

[22] M. Calder, A. Koliousis, M. Sevegnani, and J. S. Sventek, "Real-time verification of wireless home networks using bigraphs with sharing," *Science of Computer Programming*, vol. 80, pp. 288–310, 2014.

[23] M. Calder and M. Sevegnani, "Modelling IEEE 802.11 CSMA/CA RTS/CTS with stochastic bigraphs with sharing," *Formal Aspects of Computing*, vol. 26, no. 3, 2014.

[24] C. Tsigkanos, L. Pasquale, C. Ghezzi, and B. Nuseibeh, "On the interplay between cyber and physical spaces for adaptive security," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 3, pp. 466–480, May 2018.

[25] F. Alrimawi, L. Pasquale, and B. Nuseibeh, "On the automated management of security incidents in smart spaces," *IEEE Access*, vol. 7, pp. 111 513–111 527, 2019.

[26] W. J. Thong and M. A. Ameedeen, "A survey of UML tools," in *Proceedings of the Second International Conference on Advanced Data and Information Engineering, DaEng 2015, Bali, Indonesia, April 25-26, 2015*, 2015, pp. 61–70. [Online]. Available: https://doi.org/10.1007/978-981-13-1799-6_7

[27] M. Sevegnani and M. Calder, "BigraphER: Rewriting and analysis engine for bigraphs," in *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, 2016, pp. 494–501.

[28] B. Weyers, J. Bowen, A. Dix, and P. A. Palanque, Eds., *The Handbook of Formal Methods in Human-Computer Interaction*. Springer International Publishing, 2017.

[29] Y.-W. Lin, Y.-B. Lin, M.-T. Yang, and J.-H. Lin, "ArduTalk: An Arduino network application development platform based on IoTtalk," *IEEE Systems Journal*, vol. 13, no. 1, pp. 468–476, March 2019.

[30] Y.-B. Lin, H.-C. Tseng, Y.-W. Lin, and L.-J. Chen, "NB-IoTtalk: A service platform for fast development of NB-IoT applications," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 928–939, Feb 2019.

[31] Y.-B. Lin, M.-Z. Shieh, Y.-W. Lin, and H.-Y. Chen, "MapTalk: mosaicking physical objects into the cyber world," *Cyber-Physical Systems*, vol. 4, no. 3, pp. 156–174, 2018.

[32] Y.-B. Lin, L.-K. Chen, M.-Z. Shieh, Y.-W. Lin, and T.-H. Yen, "CampusTalk: IoT devices and their interesting features on campus applications," *IEEE Access*, vol. 6, pp. 26 036–26 046, 2018.

[33] G. Conforti, D. Macedonio, and V. Sassone, "Static BiLog: a unifying language for spatial structures," *Fundamenta Informaticae*, vol. 80, no. 1-3, pp. 91–110, 2007.

[34] B. Archibald and M. Sevegnani, "BigraphTalk source code," May 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3240171

[35] A. Gassara, I. B. Rodriguez, M. Jmaiel, and K. Drira, "Executing bigraphical reactive systems," *Discrete Applied Mathematics*, vol. 253, pp. 73–92, 2019, 14th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW 2016).

[36] Y.-W. Lin, Y.-B. Lin, C.-Y. Hsiao, and Y.-Y. Wang, "IoTtalk-RC: Sensors as universal remote control for aftermarket home appliances," *IEEE Internet of Things Journal*, vol. 4, no. 4, pp. 1104–1112, Aug 2017.

[37] M. Sevegnani and M. Calder, "Bigraphs with sharing," *Theor. Comput. Sci.*, vol. 577, pp. 43–73, 2015.

[38] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Software and Systems Modeling*, vol. 18, no. 5, pp. 3049–3082, 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-00712-x

Blair Archibald is a Research Associate at the University of Glasgow, where he previously obtained his Ph.D. His research interests include systems modelling with bigraphs, parallelising large combinatorial search problems, and sensor networks.

Min-Zheng Shieh received the B.S. and M.S. degrees in Computer Science and Information Engineering and the Ph.D. degree in Computer Science and Engineering, all from National Chiao Tung University, Taiwan, in 2003, 2004 and 2011, respectively. From 2012 to 2016, he served as an assistant research fellow of the Information and Communication Technology Laboratories, National Chiao Tung University. Since 2016, he has been an assistant professor of Information Technology Service Center at National Chiao Tung University. His main research interests include computational complexity, algorithms, coding theory and Internet of Things.

Yu-Hsuan Hu received the B.S. degree from the Department of Computer Science and Information Engineering, National Taipei University, Taipei, Taiwan, in 2014. She is currently pursuing the M.S. degree at the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan. Her research interests include operating systems and Internet of Things.

Michele Sevegnani is Lecturer in Computing Science at the University of Glasgow. He received a PhD in Computing Science from the University of Glasgow, Scotland in 2012 and an MSc in Bioinformatics jointly from the universities of Edinburgh (Scotland) and Trento (Italy) in 2008. He is one of the leading researchers in bigraph theory and applications, especially reasoning about safety, reliability and predictability of location-aware, event-based, systems that are deployed and were not designed with reasoning in mind.

Yi-Bing Lin (M'96-SM'96-F'03) is Winbond Chair Professor of National Chiao Tung University (NCTU). He received his Bachelor's degree from National Cheng Kung University, Taiwan, in 1983, and his Ph.D. from the University of Washington, USA, in 1990. From 1990 to 1995 he was a Research Scientist with Bellcore (Telcordia). He then joined NCTU in Taiwan, where he remains. In 2010, Lin became a lifetime Chair Professor of NCTU, and in 2011, the Vice President of NCTU. During 2014 – 2016, Lin was Deputy Minister, Ministry of Science and Technology, Taiwan. Since 2016, Lin has been appointed as Vice Chancellor, University System of Taiwan (for NCTU, NTHU, NCU, and NYM). Lin is AAAS Fellow, ACM Fellow, IEEE Fellow, and IET Fellow.