A Runtime System for Logical-Space Programming

Eloi Pereira*
Systems Engineering
UC Berkeley, CA, USA
eloi@berkeley.edu

Clemens Krainer
Dept. of Computer Sciences
Univ. of Salzburg, Austria
clemens.krainer@cs.uni-salzburg.at

Pedro Marques da Silva
Research Center
Air Force Academy, Portugal
posilva@academiafa.edu.pt

Christoph M. Kirsch
Dept. of Computer Sciences
Univ. of Salzburg, Austria
ck@cs.uni-salzburg.at

Raja Sengupta
Systems Engineering
UC Berkeley, CA, USA
sengupta@ce.berkeley.edu

ABSTRACT

In this paper we introduce *logical-space programming*, a spatial computing paradigm where programs have access to a logical space model, i.e., names and explicit relations over such names, while the runtime system is in charge of manipulating the physical space. Mobile devices such as autonomous vehicles are equipped with sensors and actuators that provide means for computation to react upon spatial information and produce effects over the environment. The spatial behavior of these systems is commonly specified at the physical level, e.g., GPS coordinates. This puts the responsibility for the correct specification of spatial behaviors in the hands of the programmer. We propose a new paradigm named logical-space programming, where the programmer specifies the spatial behavior at a logical level while the runtime system is in charge of managing the physical behaviors. We provide a brief explanation of the logical-space computing semantics and describe a logical-space runtime system using bigraphs as logical models and bigActors as computing entities. The physical entities are modeled as polygons in a geometrical space. We demonstrate the use of logical-space programming for specifying and controlling the spatial behaviors of vehicles and sensors performing an environmental monitoring mission. The field test consisted of an Unmanned Aerial Vehicle and GPS drifters used to survey an area supposedly affected by illegal bilge dumping.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*Relations between models, Self-modifying machines*

*The author is also with the Research Center of the Portuguese Air Force Academy.

SWEC '15, April 13, 2015, Seattle, Washington

Keywords

Spatial Computing, Mobile, Robotics

1. INTRODUCTION

Computation is becoming ubiquitously and spatially embedded in our environment. Mobile cyber-physical systems such as smartphones and robots are equipped with sensors and actuators that observe and manipulate their spatial environment. This kind of computation that exhibits a behavior in space is commonly known as *spatial computing* [6].

Spatial computing often involves defining the behavior of machines in a geometrical location model such as GPS coordinates or indoor local coordinates [7]. We call this *physical-space programming*.

Example 1. Consider the example of an Unmanned Aerial Vehicle (UAV) collecting imagery of an oil-spill due to a suspicious illegal bilge dumping activity by an oil tanker. The exact location of the oil-spill is unknown *a priori*, although, due to *Automatic Identification System* (AIS) information collected from the tanker, it is known to be within a given rectangular area parametrized by its North-East and South-West GPS locations, $(37.04, -8.59)$ and $(36.94, -8.79)$. The UAV operator performs the following steps: select an UAV to perform the mission; specify a searching pattern comprised by the sequence of GPS locations to be visited inside the suspected area; as soon as the operator gets the information of the oil-spill location by some source, specify a new location to be visited. The mission is specified as a sequence of waypoints using a given format such as the Waypoint File Format (WFF) of the Mavlink Waypoint Protocol (MWP) [2]. The bottom row of images in Figure 1 depicts the physical-space execution of this example.

Physical-space programming provides full control of the physical capabilities of the involved computing devices. This puts the responsibility for the correct specification of spatial behaviors in the hands of the programmer. For example, a mistake in the specification of the GPS coordinates of the waypoints can lead to an unexpected behavior. Moreover, physical-space models do not entail explicitly relational information of spaces. For example, it would be important for the UAV operator to know if the UAV is at the search area without the need to perform any further calculations.

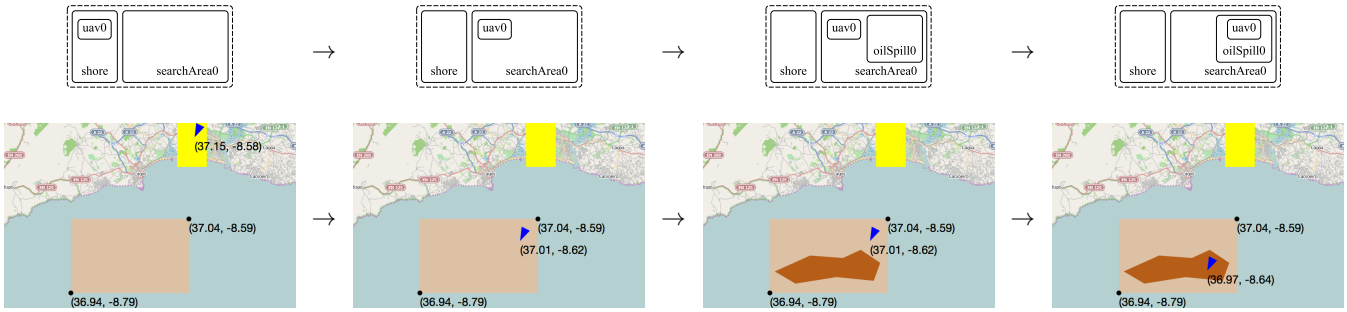


Figure 1: Symbolic-space execution (top row) and physical-space execution (bottom row) for Example 1.

The literature presents several programming models that approaches spatial computing from a physical level, such as Amorphous Computing [3], Spatial Programming [8], and the framework Gaia [13].

Another common approach on spatial computing is to model space symbolically, where locations are defined as symbols and explicit relations over those symbols [7]. An everyday example of a location model is the set of streets names, cities, and countries organized by their containment relation. We call it *symbolic-space programming* when computation is defined over symbolic-space models. One of the pioneer symbolic-space programming models is the Ambient Calculus by Luca Cardelli [9], where mobile processes can compute over bounded locations and are allowed to communicate if they share the same location. In Ambient Calculus locations have a tree-like structure. Inspired by Cardelli's model and by its own $\pi-$calculus, Robin Milner introduced the Bigraphical model [10] that combines a nested location model with a model of connectivity. A bigraph changes to another bigraph upon the application of Bigraph Reaction Rules (BRR). The top row of images in Figure 1 shows the bigraphical execution for Example 1.

Symbolic-space programming provides an abstract spatial model with explicit relations between locations. These models are in general convenient for specifying and formally verifying high-level spatial behaviors. However, they abstract away the physical behaviors of machines and their environment, which are necessary to operate the machines. For example, the application of a BRR that moves an UAV from its current position to another location is executed in the symbolic model as soon as it is requested. At the physical level, a control action does not execute instantaneously. It might not even execute at all due to some adversarial action of the environment. A programmer must be able to write programs that can cope with this asynchrony and react upon inadvertent behaviors.

In this paper we introduce *logical-space computing*. In logical space computing, the programmer manipulates a symbolic abstraction of the world, named the *logical-space model*, while the runtime system is in charge of manipulating a physical abstraction of the world, named the *physical-space model*. Both abstractions are loosely coupled by the logical-space semantics, which provides an asynchronous semantics for the execution of control actions and for the observation of the structure.

In this paper we present informally the semantics of logical-space computing. A complete formal treatment is introduced in [12]. We show how the BigActor model [11] can be used for logical-space programming and describe a runtime system for programming mobile robots using bigActors. We conclude this paper with experimental results, where this paradigm is used to control an UAV and sensors performing an oil-spill monitoring mission.

2. LOGICAL-SPACE COMPUTING

In logical-space computing the programmer handles a symbolic spatial abstraction with a well-defined semantics of mobility, while the runtime-system handles the physical execution. Logical-space computing provides a semantics to bridge these two spatial models.

2.1 Semantics

The logical-space computing semantics is modeled as a transition system over spatial-computing configurations. A spatial-computing configuration is denoted as $\langle \alpha \mid S \mid \eta \rangle$, where α is a set of spatial agents, S is a spatial structure, and η is a set of pending requests.

A spatial agent is a computing entity with local state that can perform three commands: `observe(q)`, `react(x)`, and `control(r)`. Command `observe(q)` requests an observation of the logical model specified by a query `q`. Command `react(x)` assigns to a local variable `x` the value of a requested observation. Command `control(r)` requests the execution of a control action over the logical model specified by the reaction rule `r`.

We define a logical model as $L = (dom(L), \sigma_L)$ where $dom(L)$ denotes the set of locations of L, i.e., the set of symbols, and σ_L is a set of relations over $dom(L)$. Likewise, a physical model P has a set of physical locations $dom(P)$ and a set of relations σ_P over $dom(P)$.

A spatial structure binds these two abstractions together. A spatial structure S is a tuple (L, P, β, γ) where L is a logical model, P is a physical model, $\beta : dom(L) \to dom(P)$ is the physical interpretation function that maps logical locations from L into physical locations in P, and $\gamma : \sigma_L \to \sigma_P$ provides an interpretation of the relations in L into relations in P.

We say that a structure (L, P, β, γ) is *consistent* if the interpretation of locations from L to P preserves the relations in L, e.g., in Figure 2 the parenting of the nodes in the bigraph is consistent with the containment relation over polygons. A structure (L, P, β, γ) is locally consistent with respect to L' if L' is contained by L and (L', P, β, γ). Locally consistency is an important property for correctness

of logical-space executions. This topic is discussed in depth in [12].

The logical-space computing semantics is modeled abstractly in order to fit different logical and physical models. Figure 2 shows an example of a bigActor as a spatial agent that operates over a bigraphical model of the world. The physical world is modeled as polygons defined using GPS coordinates. The bigActor specified in Figure 2 uses a query

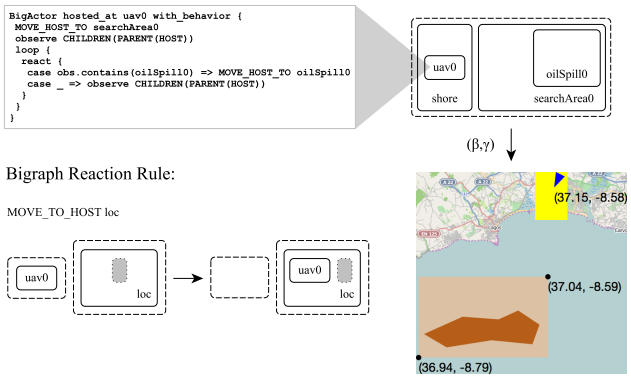


Figure 2: Logical-space program.

language to observe the logical space for oil-spills and BRRs to move the UAV from its current location to a new one. β maps each bigraph node to a polygon. γ maps the bigraph parenting relation to a containment relation over polygons.

The semantics is written in an operational style, largely influenced by [4, 5, 11]. It is formalized as a transition system over the space of spatial-computing configurations, specified by seven inference rules.

2.1.1 Computation

The rule denoted as $\langle \texttt{fun} : a \rangle$ models an internal computation performed by agent a, i.e., the change of the local state of a specified by the semantics of a host programming language.

2.1.2 Observation

There are three rules for modeling observations. Rule $\langle \texttt{req\_obs} : a, \texttt{observe(q)} \rangle$ models an agent a requesting an observation defined by query $\texttt{q}$ of the logic-space model. The request is defined as $\texttt{OBS}(a, \texttt{q})$ and it is stored in the set of pending requests η. Rule $\langle \texttt{sense} : \texttt{OBS}(a, \texttt{q}) \rangle$ models the runtime system taking an observation request $\texttt{OBS}(a, \texttt{q})$ from the set of pending requests, interpreting the query over the physical structure, and generating a new logical abstraction L_q. The result is stored in the set of pending requests as $\texttt{READY}(a, L_q)$. This rule is responsible for the keeping the logical model and the physical model locally consistent with respect to the observed space, i.e., if two observed physical locations are related, then their logical counterparts are also related. Rule $\langle \texttt{rcv\_obs} : a, \texttt{react(x)} \rangle$ delivers an observation $\texttt{READY}(a, L_q)$ to a by assigning L_q to the local variable $\texttt{x}$. Note that observation is asynchronous, i.e., an agent first requests an observation, the runtime system gets the necessary data from sensors, and delivers the result as soon as possible.

2.1.3 Control

There are two rules for modeling control actions from spatial agents and one from environmental sources. Rule $\langle \texttt{req\_ctr} : a, \texttt{control}(R \Rightarrow R') \rangle$ models an agent a requesting a control action over the the logical structure specified by the reaction rule $R \Rightarrow R'$, where R specifies the part of the logic model to be changed and R' specifies how it is intended to be changed. The rule generates a request $\texttt{CTR}(a, R \Rightarrow R')$ in the set of pending requests. Rule $\langle \texttt{actuate} : \texttt{CTR}(a, R \Rightarrow R') \rangle$ models the runtime system taking a request $\texttt{CTR}(a, R \Rightarrow R')$, checking if it can be applied over the logical and physical space models, and executing the rule over both models. The rule requests the spatial structure to be locally consistent with respect to R and keeps the structure locally consistent with respect to R'. Note that if a single agent observes first the space that is willing to control, locally consistency is ensured and control action can be successfully executed. Nonetheless, in the presence of concurrency, one must ensure that the space being controlled is by agents is free of race-conditions. In [11] we provide sufficient conditions to cope with these concurrency issues.

The effects of the environment are modeled by rule $\langle \texttt{env} : P' \rangle$, which changes the physical model to P'.

The semantics of the logical-space program of Figure 2, which logical-space execution is depicted in Figure 1, is the following sequence of configurations:

$$c_0 \xrightarrow{\langle \texttt{req\_ctr}:a, r=\texttt{MOVE\_HOST\_TO searchArea0} \rangle} c_1 \xrightarrow{\langle \texttt{actuate:CTR}(a,r) \rangle} c_2$$
$$\xrightarrow{\langle \texttt{req\_obs}:a, \texttt{observe}(q=\texttt{CHILDREN(PARENT(HOST))}) \rangle} c_3 \xrightarrow{\langle \texttt{sense:OBS}(a,q) \rangle} c_4$$
$$\xrightarrow{\langle \texttt{rcv\_obs}:a, \texttt{react}(L_q) \rangle} c_5 \xrightarrow{\langle \texttt{fun}:a \rangle} c_6 \xrightarrow{\langle \texttt{req\_ctr}:a, r'=\texttt{MOVE\_HOST\_TO oilSpill0} \rangle} c_7$$
$$\xrightarrow{\langle \texttt{actuate:CTR}(a,r) \rangle} c_8$$

where $c_i = \langle \alpha_i \mid S_i \mid \eta_i \rangle$. The analysis of the execution trace shows the asynchronous nature of both observation and control.

3. RUNTIME SYSTEM

Next we present a runtime system for programming in logical-space, where spatial agents are specified as bigActors, logical spaces are modeled as bigraphs, and physical spaces are modeled as polygons defined by GPS coordinates.

Figure 3 shows the overall runtime system. The left-hand side of Figure 3 depicts bigActor instances running over the BigActor Runtime System (BARS). BARS provides means for symbolic-space programming with bigActors. Our former implementation used BARS over a model checker responsible to simulate the bigraphical execution. The right-hand side of Figure 3 depicts the Logical-Space Execution Engine (LSEE) that extends BARS with a logical-space computing semantics. In this paper we present an implementation of the LSEE for programming UAVs and sensors used in an oil-spill monitoring scenario. Nonetheless, the implementation can easily extended with plugins to address other kind of sensors and actuators.

3.1 BigActor Runtime System

BigActors [11] are mobile agents that are embedded in bigraphical [10] models of space. The concurrency is modeled as per Hewitt and Agha's actor model [4], i.e., they have local state and communicate by asynchronous message-passing. Location and mobility of bigActors are modeled

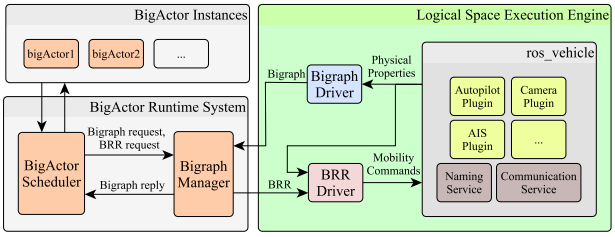


Figure 3: BigActor Runtime System for Logical-Space Programming.

using the bigraphical formalism. A bigActor is able to asynchronously observe locally the bigraph and control by requesting the execution of BRRs.

The BigActor Language is implemented as a Scala embedded Domain Specific Language (DSL) [1]. It is an extension of the Scala Actor library with BigActor commands plus implicit conversions for achieving a domain-specific syntax. We use Scala for two reasons: (1) The concurrency model of Scala is the Actor Model. (2) The type system together with high-order functions and implicit conversions makes Scala a powerful language to implement DSLs. Scala Actor library became recently deprecated in favor of the Akka Actor library. In order to cope with these changes in the Scala ecosystem, we are currently migrating the implementation to Akka Actors.

BigActor instances are Scala actors. The instances send requests to the BigActor Scheduler, which schedules them according to a *First-Come First-Served* policy, and manages their execution. Instances can send communication, observation, control, and migration requests.

The interactions between the BigActor Scheduler and the bigraphical model of space is mediated by the Bigraph Manager. The Bigraph Manager is responsible for delivering fresh bigraphical observations and for executing control actions specified as BRRs.

3.2 Logical-Space Execution Engine

The LSEE has three roles: serve as a middleware for sensors and actuators, generate bigraphs out of physical properties, and interpret BRRs into control commands that can be executed by a given actuator. Next we present the components that are responsible for these tasks.

3.2.1 Middleware

The middleware component named `ros_vehicle` contains software drivers named *plugins*. Plugins are responsible to interact with components that provide or consume spatial information. These components can be for example a GPS device, an autopilot, a computer vision system, or a cloud-based location service accessed over the internet. A plugin has a well-defined interface. It can subscribe to *mobility commands*, e.g., a waypoint command for an autopilot, and publish *physical properties*. A physical property may contain static information, such as a polygon describing the border of a city, or dynamic information, such as the location and connectivity of an UAV. Plugins are implemented over the Robot Operating System (ROS), which provides a *publish-subscribe* communication mechanism. For the oil-spill scenario we implemented five plugins. The *Autopilot Plugin* handles the execution of GPS waypoints over the autopilot and fetches the UAV state information, like GPS location, velocity, and control authority. The *AIS Plugin* receives, decodes, and filters AIS messages from an onboard AIS receiver. The *Camera Plugin* uses a video camera driver to capture and process video frames.

The `ros_vehicle` is also equipped with a *Naming Service* and a *Communication Service*. The Naming Service is responsible for assigning unique names to physical properties and can be implemented using different naming conventions. For example, the Naming Service implemented for the oil-spill scenario uses the autopilot serial number to identify the location of the UAV and the AIS Maritime Mobile Service Identity (MMSI) to identify the locations of the drifters. The Communication Service is responsible for sharing local observations between `ros_vehicle`s. For the oil-spill scenario, the Communication Service is implemented using UDP as the transport protocol over a 3G network. The service is used to share physical properties between different `ros_vehicle`s at different ground stations.

3.2.2 Generation of bigraphs

The Bigraph Driver subscribes physical properties from the `ros_vehicle` and generates bigraphical abstractions. The parenting of a bigraph node b is calculated by finding the smallest polygon that totally contains $\beta(b)$, i.e., its physical interpretation. In order to cope with Milner's bigraph definition we must enforce that the resulting parenting relation forms a tree. As such, a polygon can not be partially contained on another polygon, otherwise, the resulting parenting relation may form a cycle. This limitation can be removed by using Sevegnani's Bigraphs with Sharing [14].

Figure 4 depicts an example of the generation of a bigraph from physical properties produced by a network of vehicles and sensors.

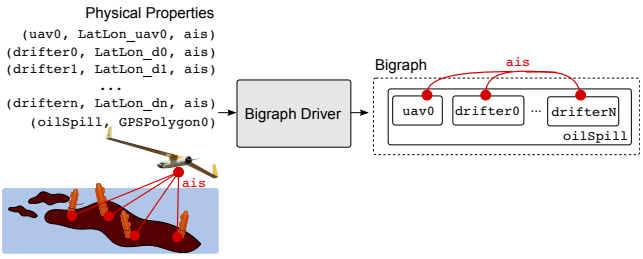


Figure 4: Bigraph Driver example.

The Bigraph Driver is stateful, i.e., it keeps track of the physical properties that it observes in order to generate bigraphs with increasing spatial information over time. If the Bigraph Driver receives a physical property that was already observed, it disregards the old one and re-generates the bigraphical abstraction.

The Bigraph Driver uses the Communication Service to exchange observed physical properties with other robots. Thus, the Bigraph Driver hosted by each robot generates a bigraphical estimate of the global bigraphical abstraction. In order to solve ambiguities between observations of the same physical space by different robots we augment each physical property with a GPS time-stamp. Example 2 shows the Communication Service being used by UAV operators during a handover of control authority.

Example 2. Consider that `uav0` that starts a mission under the control of `gcs0` and, at a given point, hands control over to `gcs1` on-board of a navy vessel. Figure 5 depicts this situation. Each operator has a local and limited observation of the world. The operator on the vessel does not know where the UAV is located until the handover has been successfully completed. The use of the Communication Service allows the operators to have access to an extended bigraphical abstraction. With this information, both operators have access to the location of the UAV before and after hand-over.

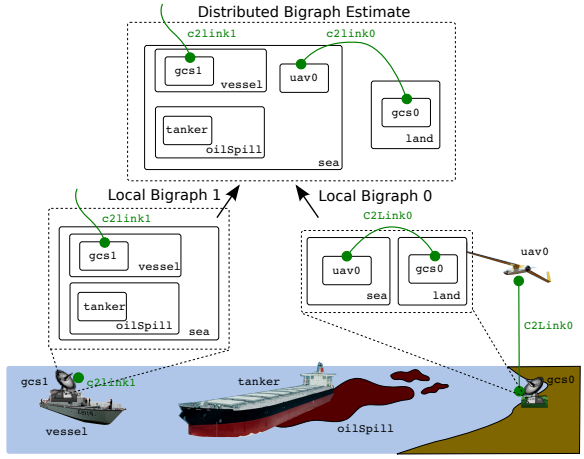


Figure 5: Distributed bigraph example

Bigraph observations "flood" over the network of robots and will eventually converge to a distributed bigraph estimate.

3.2.3 *Generation of control commands*

The BRR Driver translates BRRs to mobility commands that can be executed by devices that are interfaced by `ros_vehicle` plugins. In order to synthesize commands, the BRR Driver needs the physical interpretations of the nodes in the BRR. To derive the physical interpretation, the BRR Driver subscribes physical properties from `ros_vehicle`. For example, the generation of a waypoint command to move an UAV to a given destination needs the GPS location of the UAV and the destination.

Figure 6 exemplifies the execution of a BRR for moving an UAV to the oil-spill location. The BRR Driver generates a mobility command that specifies a GPS waypoint command to the centroid of the polygon that defines the the oil-spill. The mobility command is subscribed by the Autopilot Plugin from the `ros_vehicle` instance. The Autopilot Plugin is responsible for managing the execution of the waypoint.

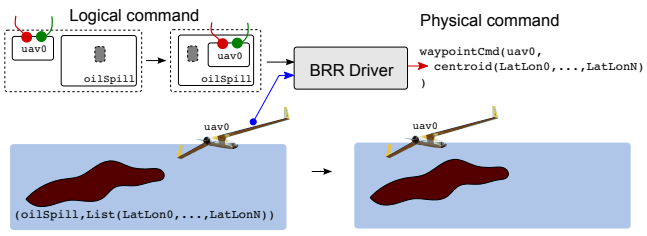
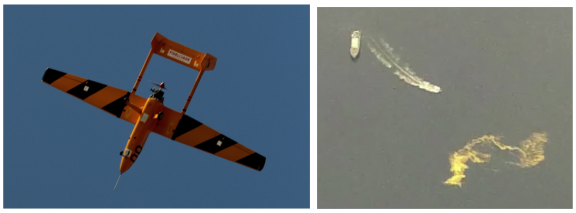


Figure 6: BRR Driver example.

4. OIL-SPILL CASE STUDY

The field test uses an UAV with a camera and drifters with AIS modems and GPS to monitor an oil-spill. The oil-spill is emulated by a Navy vessel dropping 100 kg of pop-corn 6 km south of the shore of Portimão, Portugal. This is a small spill of the kind that might be created by a large ship flushing its oil tanks, also known as bilge dumping. Bilge dumping is a major problem for small countries like Portugal with large maritime zones. Bilge dumping evidence is currently collected using satellite images correlated with AIS information from proximate vessels [15]. The field test aimed to assess the role of unmanned vehicles and sensors as complements to satellites for the collection of evidence.

We used two kinds of UAVs developed at the Portuguese Air Force Academy under the PITVANT project, the Alfa and the Alfa-Extended (Figure 7(a)). The Alfa-Extended



(a) Extended UAV (b) Onboard image

Figure 7: Alfa Extended UAV, onboard UAV picture of the oil-spill (popcorn) and Navy vessel.

is a gas-powered UAV with 3 m wingspan, equipped with a Piccolo autopilot for stable low-level control, and a PC-104 computing board for high-level control and vision processing. Each UAV is equipped with a gimballed optic camera and an AIS receiver.

The Unmanned Aerial System included three ground control stations (GCS) denoted as `gcs0`, `gcs1`, and `gcs2`. `gcs0` was situated in an air field and was responsible for take-off and landing maneuvers, `gcs1` was located at the shore and took control authority of the UAVs during emergencies, and `gcs2` was located at the shore and was responsible for the UAV mission. In one particular scenario, `gcs2` was located on a Navy vessel to extend the operational range of the UAV.

The drifters used in this demonstration were AIS beacons commonly used for locating fishing nets. They were equipped with GPS and transmitted their position up to a range of 10 miles. Drifters were identified by unique MMSI numbers.

The communication infrastructure included wireless communication links to connect the UAVs and the GCSs: 3G internet access to connect all GCSs, UHF communication radios for inland communication, and VHF radios for maritime communication. The European Maritime Safety Agency (EMSA) participated in the scenario, tasking a satellite to take a high-resolution optical picture of the oil-spill.

Next we present the lessons learned from programming the oil-spill mission in logical-space.

Recall the bigActor defined in Figure 2. The bigActor requests to move the UAV to logical locations, e.g., `oilSpill0`. Since the oil-spill moves over time, each execution of the same instruction at the logical level maps to a different instruction at the physical level. In other words, a new waypoint command must be generated each time the log-

ical location moves its physical location. Prior to the use of logical-space programming, the operator had to manually specify these new waypoints in physical-space, which was inconvenient and easy to make mistakes. With logical-space programming, the command `MOVE_HOST_TO oilSpill0` is the only one needed. The logical-space execution engine ensures that waypoints have always the correct physical coordinates.

Consider another bigActor defined in Figure 8. The bigActor observes the bigraph with a query `LINKED_TO_HOST` and displays the result. The bigActor also has alternatives for matching a `handover` message, which results in a BRR handing over the control authority for `uav0` to ground station `gcs0`. In our field test these messages were sent by another bigActor implementing a graphical user interface.

```
0 BigActor hosted_at gcs2 with_behavior {
1   observe(LINKED_TO_HOST)
2   loop {
3     react {
4       case obs: Observation => display(obs)
5                                observe(LINKED_TO_HOST)
6       case "handover" => control(HAND uav0 TO gcs0)
7     }
8   }
9 }
```

Figure 8: Code for bigActor `handover`.

Our operators were able to watch bigraphs evolve as the field test progressed. The logical abstraction proved particularly useful for UAV handovers, since it provided the operators with means to be constantly aware of the UAV location and connectivity regardless of which ground station had control authority. This was provided by our distributed bigraph estimation protocol executing over the Internet, which allowed synchronizing the bigraphs at both ground stations. The prior practice was to watch the control screen provided by the autopilot vendor that would only display any information if the UAV was under the control authority of the respective ground station. Correct termination used to be ensured by radio communication between operators. This communication was discontinued as the operators came to understand and trust the displayed bigraph.

5. CONCLUSION

In this paper we introduce a new paradigm for spatial computing named logical-space programming. In logical-space programming, programmers manipulate a logical-space model, while the runtime system is responsible to mediate this abstraction with the physical space. We introduce the logical-space computing semantics informally and describe the BigActor Runtime System for logical-space programming. The runtime system uses bigActors as spatial agents that operate over a bigraphical space model. The physical space model consists of polygons defined using geometrical coordinates. We demonstrated the use of the logical-space programming in a case study where vehicles and sensors performed an environmental monitoring mission.

Acknowledgment

This work has been supported by the National Science Foundation (CNS1136141), by the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), by the Fundação para a Ciência e Tecnologia (SFRH/BD/43596/2008), and by the Portuguese MoD - PITVANT. The authors want to thank the Portuguese Air Force, the Portuguese Navy, the European Maritime Safety Agency, and the Portimão Airfield.

6. REFERENCES

[1] Bigactor language repository. `https://bitbucket.org/eloipereira/bigactors`. Accessed: 2014-12-16.

[2] Mavlink waypoint protocol. `http://qgroundcontrol.org/mavlink/waypoint_protocol`. Accessed: 2014-12-09.

[3] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman, and R. Weiss. Amorphous Computing. *Communications of the ACM*, 43(5):74–82, 2000.

[4] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[6] J. Beal, S. Dulman, and K. Usbeck. Organizing the aggregate: Languages for spatial computing. In *Formal and Practical Aspects of Domain-Specific Languages*, pages 1–60. Information Science Reference, 2012.

[7] B. Becker, D. Beyer, H. Giese, F. Klein, and D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. *Proceeding of the 28th international conference on Software engineering - ICSE '06*, page 72, 2006.

[8] C. Borcea, C. Intanagonwiwat, P. Kang, U. Kremer, and L. Iftode. Spatial programming using smart messages: design and implementation. In *24th International Conference on Distributed Computing Systems. Proceedings.*, pages 690–699. Ieee, 2004.

[9] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures*, pages 140–155. Springer, 1998.

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

[11] E. Pereira, C. Kirsch, R. Sengupta, and J. Borges de Sousa. Bigactors - a model for structure-aware computation. In *4th International Conference on Cyber-Physical Systems*. ACM/IEEE, April 2013.

[12] E. Pereira, P. Silva, C. Krainer, C. Kirsch, and R. Sengupta. Logical space computing. Technical report, December 2014. `http://cpcc.berkeley.edu/papers/logicalSpaceComputingWorkingPaper.pdf`.

[13] M. Roman and R. Campbell. Gaia: enabling active spaces. *Proceedings of the 9th workshop on ACM*, 2000.

[14] M. Sevegnani. *Bigraphs with sharing and applications in wireless networks*. PhD thesis, Univ. Glasgow, 2012.

[15] SkyTruth. Bilge dumping? busted using satellite images and ais data. `http://blog.skytruth.org/2012/06/bilge-dumping-busted-using-satellite.html`, 2012.