Integrating a Computational Model and a Run Time System for Image Processing on a UAV

Per Andersson and Krzysztof Kuchcinski
Lunds University
P.O. Box 118
SE-221 00 Lund
Sweden
{Per.Andersson, Krzysztof.Kuchcinski}@cs.lth.se

Klas Nordberg and Patrick Doherty
Linköping University
S-581 83 LINKÖPING
Sweden
klas@isy.liu.se and patdo@ida.liu.se

Abstract

Recently substantial research has been devoted to Unmanned Aerial Vehicles (UAVs). One of a UAV's most demanding subsystem is vision. The vision subsystem must dynamically combine different algorithms as the UAV's goal and surrounding chang. To fully utilize the available hardware, a run time system must be able to vary the quality and the size of regions the algorithms are applied to, as the number of image processing tasks changes. To allow this the run time system and the underlying computational model must be integrated. In this paper we present a computational model suitable for integration with a run time system. The computational model is called Image Processing Data Flow Graph (IP-DFG). IP-DFG has been developed for modeling of complex image processing algorithms. IP-DFG is based on data ßow graphs, but has been extended with hierarchy and new rules for token consumption, which makes the computational model more ß exible and more suitable for human interaction. In this paper we also show that IP-DFGs are suitable for modelling expressions, including data dependent decisions and iterations, which are common in complex image processing algorithms.

1. Introduction

Substantial research has recently been devoted to development of Unmanned Aerial Vehicles (UAVs) [1, 2, 3, 4]. A UAV is a complex and challenging system to develop. It operates autonomously in unknown and dynamically changing environment. This requires different types of subsystems to cooperate. For example, the subsystem responsible for planning and plan execution base its decisions on information derived in the vision subsystem. The vision system, on the other hand, decides which image pro-

This work is part of the WITAS UAV project and is Þnanced by the Wallenberg Foundation.

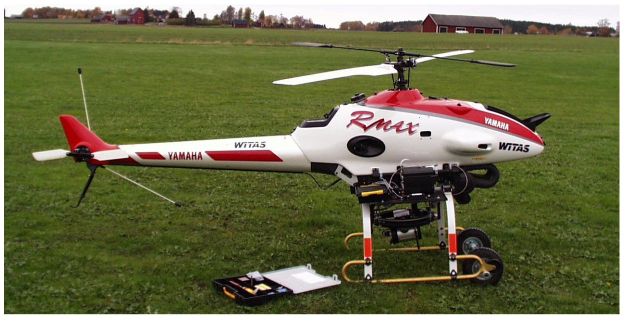
Figure 1. The mini-helicopter used for experiments in the WITAS UAV project.

cessing algorithms to run based on expectations of the surrounding, information which the planning and plan execution subsystem derives. To simplify this cooperation it is important that the subsystems are ß exible.

We currently develop a UAV in the WITAS UAV project [1]. This is a long term research project covering many areas relevant for UAV development. Within the project both basic research and applied research is done. As part of the project a prototype UAV is developed. The UAV platform is a mini helicopter and we are looking at scenarios involving trafÞc supervision and surveillance missions. The helicopter currently used in the project for experimentation is depicted in Þgure 1. On the side of the helicopter the onboard computer system is mounted and beneath the helicopter the camera gimbal is mounted. The project is a cooperation between several universities in Europe, the USA, and South America. The research groups participating in the WITAS UAV project are actively researching topics including, but not limited to, knowledge representation, planning, reasoning, computer vision, sensor fusion, helicopter modeling, helicopter control, human interaction by dialog. See [1] for a more complete description of the activities within the WITAS UAV project.

One of the most important information sources for the WITAS UAV is vision. In this subsystem symbolic information is derived from a stream of images. There exists a large set of different image processing algorithms that are suited for different purposes and different conditions. It is desirable for a UAV to be able to combine different algorithms as its surrounding and goal changes. To allow this a ß exible model for the image processing algorithms together with a run time system that manages the combination and execution of the algorithms is needed. Further more a run time system needs to represent image processing algorithms in a model with a clear semantic deÞ nition. This would make it possible to do both static (of-line) and dynamic (run-time) optimizations. Optimizations makes it possible to utilize the available processing power optimally by varying the quality and the size of the regions an algorithm is applied to, as the number of image processing tasks varies.

One such model commonly used for image processing algorithms is Synchronous Data Flow Graphs (Synchronous DFGs). However, this model can only represent static behavior. Another computational model is boolean DFG. Boolean DFGs can model dynamic behavior, but are hard to use. To overcome the mentioned problems we introduce in this paper a new model of computation aimed at high level descriptions of image processing algorithms.

2. Related work

Several systems for modeling signal processing or systems dedicated for image processing algorithms have been developed over the years [11, 12, 13]. Early work laid the foundation by developing data ßow models of computation. Based on the token ßow model several systems for modeling or designing signal processing in general or image processing is particular have been developed.

Data ßow graphs are a special case of a Kahn process networks [5]. A DFG is a directed graph where nodes represent processes and arcs represent communication channels. The communication channels are directed unbounded FIFO queues, where reads are blocking and writes are non-blocking. A DFG is executed by repeated Þring of actors (nodes in the graph). The Þrings are data driven. Each actor has a set of Þring rules, each specifying a pattern of tokens on the input channels. A pattern specify the number of tokens needed on each channel for the actor to Þre. The tokens are removed from the input channels as the actor Þres. A pattern can also specify a value for a speciÞc token. For example an actor with three inputs can have two Þring rules with the patterns {[true], [*], $\perp$} and {[false], $\perp$, [*]}. $\perp$ is called bottom and represent the empty sequence of tokens, hence is always matched. * is a wildcard and is matched by any token. This actor will consume one token on either its second or third input, depending on the value of the token on its Þrst input. The actor is a deterministic merge.

Synchronous DFGs are a special case of DFGs, where an actor consumes and produces a Þ xed number of tokens each time it Þres, i.e. it has one Þring rule. To avoid token accumulation all tokens produced must be consumed. This leads to linear system of equations called the balancing equation. From a solution of the balancing equation a static schedule and bound on the buffer size needed for the communication channels can easily be derived. See [5, 6] for a detailed description of synchronous DFGs.

Synchronous DFGs can only model static behavior. To relax this while still preserving some analyzability boolean data ßow graphs was introduced [7]. A boolean DFG actor has one input called the control input. This input receives boolean tokens. When a boolean DFG actor Þres the number of tokens consumed and produces is a function of the value of the control token. This is equivalent to two Þ ring rules, one with the pattern *true* and one with the pattern *false* for the control input and a Þ xed number of wildcards on the other inputs. A systematic approach for consistency analysis of boolean DFGs is presented in [8].

DFGs are powerful and can be used to model complex image processing algorithms. However their structure are complex and working directly with DFGs is error prone. Synchronous DFGs are much nicer to work with, but they can not model dynamic behavior which is needed in more complex image processing algorithms. Boolean DFGs extends synchronous DFGs with dynamic behavior, but they are cumbersome to use for representing recursive algorithms. Hence they are not suitable for modeling of algorithms containing iterative approximations.

3. Integrating a computational model and a run time system

By integrating the run time system and the underlying computational model it is possible to consider different aspects at different levels of granularity. Consider the general orientation algorithm shown in Þgure 2. For an algorithm like this the application typically implies a timing requirement for the whole algorithm, hence it is preferable to specify timing constraints at the algorithm level of granularity. Other aspects are better to consider at a Þner level of granularity. If the algorithm is to be executed in a parallel architecture the distribution should be done at node level and the distribution algorithm should balance the amount of computation on each processing unit and the communication between them. Similarly a scheduler should schedule the nodes individually which would make it possible to minimize the amount of live data. This could reduce the memory requirements considerably [14, 15]. The information needed for this type of optimization could be derived automatically if the computational model was carefully designed and integrated with the run time system.

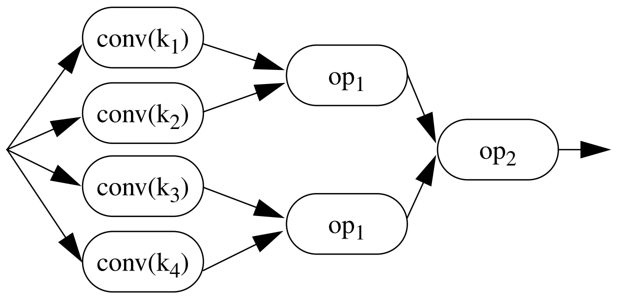

Figure 2. Orientation, a simple image processing algorithm

This approach would be more ßexible and offer more control over execution than using a traditional thread model and implement algorithm using concurrent threads.

When deciding on a computational model it is not sufÞcient only to consider the interaction with the run time system. One also have to consider what expression power is needed for the intended applications. For the vision subsystem of the WITAS UAV project the applications are image processing. Many image processing algorithms are simple. They are composed of well known operations applied to the whole image. The output of one operation is the input to another. To illustrate consider the orientation algorithm shown in Þgure 2. The input image is convolved with four different kernels resulting in four temporary images. The features extracted in the temporary images are then combined on pixel level using a binary tree structure.

All algorithms are not as simple as the orientation algorithms described above. Consider a camera control algorithm for a UAV. The camera it to be pointed at an area of interest. The camera should point to this area independent of the helicopter movement. This can be done by tracking a number of Þxpoints in the image [16]. A Þxpoint is a point that is easy to track. If one Þxpoint is out of sight, i.e. hidden, the camera control algorithm should Þnd another Þxpoint. To describe this behavior a model of computation that allows data dependent decisions is needed.

Another type of image processing algorithms which need to be considered are recursive algorithms, commonly iterative approximation algorithms. These algorithms iteratively improve their result until some conditions are satisÞed. The conditions could be that the result is sufÞciently accurate, i.e. a value exceeds a threshold, or that a maximum number of iterations has been done.

4. IPAPI - a run time system

Early in the WITAS UAV project it was clear that the planning and plan execution subsystem would need to access and guide the image processing subsystem. For this purpose an Image Processing Application Program Interface (IPAPI) was deÞned. IPAPI has evolved from a simple run time system with static behavior to a run time system with dynamic behavior, where the planning and plan execution subsystem conÞgures the set of algorithms to be executed based on what symbolic information it needs to extract from its surrounding.

IPAPI has a library with implementations of different algorithms. An algorithm is executed as long as its results are needed. The creation, execution and removal of algorithms is managed by IPAPI. IPAPI also updates the Visual Object data Base (VOB) with the result of the algorithms. The VOB contains one entry for each dynamic object the UAV keeps track of in its surrounding, The VOB is the UAV's view of the world. Other subsystem access image processing result from the VOB.

Internally the image processing algorithms are represented using a DFG based model. This representation is described later in this paper. IPAPI has functionality for dynamic creation and removal of graphs. It dynamically manages the size of temporary data in the algorithms i.e. buffers for images. The planning and plan execution subsystem sets the size of the region of either the input or the output of a graph. IPAPI then propagates the size through the graph and allocates memory for buffers needed during execution. When the size of the regions propagates through the graph operations that affect the size, i.e. the result is the intersection of the input regions, are accounted for. IPAPI also manages the execution of the graphs.

IPAPI has a type system. For each input and output of an actor a type is speciÞed. The types are checked when inputs are connected to outputs during the creation of graphs. The typing of inputs and outputs of actors makes it possible for the run time system to automatically add conversion between graphs, for example an image with pixels encoded as integers can be converted to an image with pixels encoded as ßoating points.

5. IP-DFG - a computational model

We had two goals when developing the computational model for IPAPI. First the model should be simple for humans to work with yet it should have sufÞcient expression power to allow seamless modeling of complex image processing algorithms. The second goal was to Þnd a model that is suitable for integration with a ßexible run time system such as IPAPI. The model should simplify dynamic combinations of different algorithms. It should allow

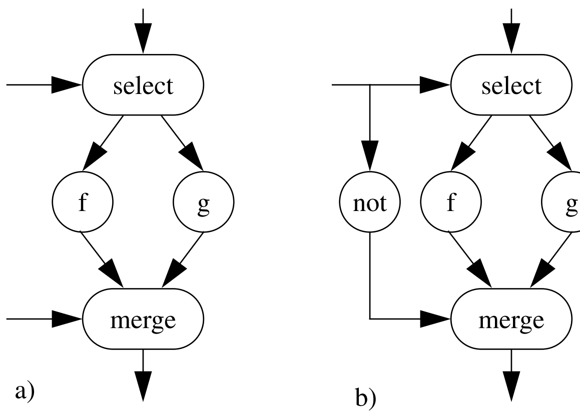

Figure 3. A DFG example

dynamic change of region sizes and the inherited parallelism of an algorithm should be preserved. Since there is no need to describe non determinism in image processing algorithms the model should only allow deterministic behavior.

DFGs are suitable for integrating with a run time system. Using data dependent Þring rules one can create complex control structures. However DFGs are cumbersome to use for these functions. There are two major obstacles, deadlock and token accumulation. The problem of deadlocks can be reduced by using a tagged token ßow model. In a tagged token ßow models tokens are tagged which makes it possible to Þre actors out of order while preserving the implicit synchronization of tokens [9, 10]. Token accumulation is illustrated in Þgure 3. If the streams of the control input (horizontal arc) of *select* and *merge* are identical the graph in Þgure 3a represents two mutually exclusive calculations (an *if then else* statement). If the control input to *merge* is the negation of the control input to *switch*, as in Þgure 3b, the behavior is quite different. Consider the input sequences for select to be {true, true, true, false} and {x_0, x_1, x_2, x_3}. First switch Þres and output x_0 on the *f* branch. The actor *f* Þres and outputs f(x_0). *Merge* receives false (the negation of *selects* input) on its control input. It can not Þre because the matching Þring rule needs one token on the *g* branch. This will not happen until *switch* has Þred four times. At this time x_3 will be sent on the *g* branch. The output sequence of the graph will be {g(x_3), f(x_0)} and several tokens are still remaining in the graph, but no Þ ring rules are matched.

Boolean DFGs can express data dependent conditions and iterations needed by complex algorithms. However it is cumbersome to express some complex algorithms using boolean DFGs. Therefore we have developed a new variant of DFGs targeted for complex image processing algorithms and more suitable for humans to work with. We call this computational model Image Processing Data Flow Graph (IP-DFG) and we have used it for integration with IPAPI.

IP-DFG is based on boolean DFG, but has been extended with hierarchy and new rules for token consumption when an actor Þres. This makes the model more ß exible. IP-DFG has the same concept, as boolean DFG, of one control input and two Þring rules for each actor. The value of the control token decides which Þring rule is used and the Þring rule determines the number of tokens consumed and produced.

As in DFGs tokens are stored on the arcs in FIFO order. However in IP-DFG an actor do not need to remove all tokens matching the Þring rule from the input. A Þring rule consists of a token pattern and a number for each input to the actor. The token pattern indicates which tokens need to be present for the actor to Þre and the number indicate how many tokens should be removed from the inputs when it Þres. If an actor has a pattern with *n* tokens and *m* tokens are removed when it Þres the actor is said to read *n* tokens and consume *m*. This simpliÞes the implementation of functions that use the same token in several Þrings, for example sliding average. Sliding average over *n* tokens reads *n* tokens but consume only 1. *n*-1 tokens remains in the FIFO-queue of the input so they can be read the next time the actor Þres. This behavior is common for image processing algorithms that extract dynamic features in an image sequence, for example optical ßow.

5.1. Hierarchy

An IP-DFG is composed of boolean DFGs in a hierarchical way. An actor can encapsulate an acyclic boolean DFG. Such actor is called a hierarchical actor. Since the internal graph of a hierarchal actor is acyclic it can not model iterations. Instead in IP-DFG iterations are explicitly deÞ ned. This makes the model more suitable for human interaction, see section 5.2. In an IP-DFG a hierarchical actor is no different from any other boolean DFG actor. It has two Þ ring rules and when it Þres it consumes and produces tokens according to the matching Þring rule. Internally the Þring is divided into three steps. The three steps perform *initialization*, *execution* and *result transmission*. For initialization and result transmission a *token mapping function* is used. A token mapping function maps a set of token sequences to another set of token sequences, $\{S_0, \ldots, S_n\} \rightarrow \{S_0, \ldots, S_m\}$. A token mapping function is usually simple, i.e. a result sequence is identical to a source sequence. However it is possible to perform more complex operations in a token mapping function, for example split one sequence or merge or concatenate two sequences. The token sequences in a token mapping function always have a Þnite length in contrast to DFGs which are working on inÞnite token sequences. Token mapping functions are described in section 5.3.

A hierarchical actor is stateless. To guarantee this the Þrst step of the Þring of a hierarchical actor is to generate the initial state of all internal arcs. The contents of the arcs are created from the tokens read by the hierarchical actor as it is Þred. This is done according to a token mapping function. When a hierarchical actor with *n* inputs and *m* internal arcs Þres a sequence of tokens, S_{i_j}, are read from each input, i_j. The token mapping function maps the set of all input sequences, $\{S_{i_1}, \ldots, S_{i_n}\}$, to a set of sequences $\{S_{a_1}, \ldots, S_{a_m}\}$. The sequence S_{a_l} is used as the initial state of the FIFO queue on arc a_l. A token mapping function can also create new tokens from constants. There is one token mapping function for each Þring rule.

The second step is the execution of the internal boolean DFG. This is done according to the Þring rules of the actors in the internal boolean DFG. When no internal Þring rules are matched the boolean DFG is blocked and the second step ends.

In the third and Þnal step the result tokens are mapped to the outputs of the hierarchical actor. This is done according to a token mapping function. When the internal boolean DFG is blocked there is a sequence of tokens, S_{a_l}, on each arc a_l. The token mapping function maps the set of all the sequences of the internal arcs, $\{S_{a_1}, \ldots, S_{a_m}\}$, to a set of sequences $\{S_{o_1}, \ldots, S_{o_m}\}$. Each sequence S_{o_l} is transmitted on the output O_l. After the token mapping function is performed all tokens remaining in the internal boolean DFG are dead. The initialization mapping guarantees that they will not be used in the next Þring of the hierarchical actor and the run time system can reclaim the memory used by these tokens. This also prevents token accumulation.

To illustrate consider the camera control algorithm outlined in section 3. The algorithm tracks a constant number of Þxpoints in the images and based on this aims the camera to the area of interest. If a Þxpoint is no longer visible it is replaced by a new visible Þxpoint. Figure 4 shows a sub-function in the algorithm. The sub-function is called *pointLocator* and Þnds one Þxpoint with a given signature in a given image, or if the Þxpoint is not found selects a new one. The result of *pointLocator* is an updated or new Þxpoint signature and corresponding parameters used in the camera control loop. *PointLocator* consume/produce one token on each input/output when it Þres. The internal actors has the following Þring rules (inputs are ordered counter clockwise starting with the the upper-most left-most one):

- *Þnd*: {[*], [*]}
- *select new:* {[*], [true]}, {⊥, [false]}
- *merge:* {[*], [true], ⊥}, {⊥, [false], [*]}

The Þring of *pointLocator* starts with the initialization step. There are two token sequences, $S_{Þxpoint}$ and S_{image}. Both contain one token. According to the token mapping

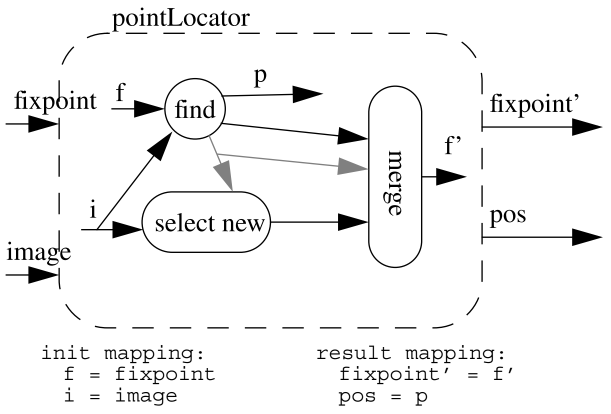

Figure 4. A hierarchical actor for tracking a Þxpoint. If the Þxpoint is not visible a new Þxpoint is selected.

function the sequence $S_{Þxpoint}$ is placed in the FIFO queue of the internal arc *f* and the sequence S_{image} is placed in the FIFO queue of *i*. After this the internal boolean DFG is executed. The Actor *Þnd* Þres Þrst, since it is the only actor with a matched Þring rule. It generates one token on each of its outputs. The gray arc is connected to the control input of *select new* and *merge*. The generated control token is *true* if the Þxpoint was found or else *false*. If the token is true both *select new* and *merge* can Þre. In this case *select new* do not generate any token and *merge* forwards the Þxpoint token from *Þnd*. If the Þxpoint was not found only *select new* can Þre. *Select new* now selects a new Þxpoint and send it to its output. *Merge* forwards this token to the *f'* arc. Now the boolean DFG is blocked and the result mapping starts. The tokens on the internal arcs *f'* and *p* are transferred to the outputs *Þxpoint'* and *pos*. In this example the actor *select new* Þred independent of whether a new Þxpoint needs to be selected. It is important to understand that it is only in one of these Þrings that the actor do any work. The purpose of the second Þring rule is only to remove tokens from the control input. If the run time system knows that the actor is stateless it can chose not to Þre an actor that do not produce any output. Instead the tokens matching the Þring rule are discarded. Also note that if the Þxpoint was not found the Þxpoint token from *Þnd* to *merge* will not be consumed. This is not a problem because the next time *pointLocator* Þres it will be removed during initialization. In fact a run time system should reclaim this token directly when *pointLocator* is blocked.

From this example it seems unnecessary to have a token mapping function for initialization and transferring the result to the outputs. It would be simpler to directly connect the internal arcs with the FIFO queues of the surrounding arcs. However this separation allows an actor to create constant tokens during the initialization step and it simpliÞes modeling of iterative functions as described below.

5.2. Iteration

There are two types of iterations of interest when dealing with image processing, iteration over data and tail recursion. In iteration over data the same function is applied on independent data. In a computational model based on the token ßow model this is achieved by placing several tokens, each containing one quantum of data, in the graph. The actors will Þre appropriate number of times. To use the *pointLocator* shown in Þgure 4 for tracking ten Þxpoints one simply place ten tokens containing the signature on the *Þxpoint* arc and ten images on the *image* arc, *pointLocator* will then Þre ten times. This implementation tracks different Þxpoints in different images. For tracking several Þxpoint in the same image a more suitable approach is to change the Þring rules. By setting the Þring rule of *pointLocator* to consume ten tokens on the *Þxpoint* input and one on the *image* input the proper tokens are placed on the *f* and *i* internal arcs during the initialization mapping. The Þring rule of the internal actors *Þnd* and *select new* must also be changed. They should read one token and consume zero tokens from the *i* arc. The image token on the internal arc *i* will be removed after the result mapping, so *pointLocator* will behave correct the next time it Þres.

Tail recursion is an important type of iteration commonly used for iterative approximations. In tail recursion the result of one invocation of a function is used as argument for the next invocation of the same function. Tail recursion is equivalent to iteration and in the rest of this paper we will use the term iteration. In IP-DFGs iteration is implemented using hierarchical actors. When a hierarchical actor fires the internal graph can be executed several times. For modeling of iteration we have extend hierarchical actors with a *termination arc* and an *iteration mapping function*. Such actors are called iterative actors.

When an iterative actor Þres it is initialized and the internal boolean DFG is executed once as a hierarchical actor. As part of the execution a termination condition is evaluated, resulting in a boolean token on the *terminate arc*. The *terminate arc* is a special arc in the internal boolean DFG of an iterative actor. If the last token on the *terminate arc* is false when the internal boolean DFG is blocked the internal boolean DFG is to be executed again. Before the repeated execution starts the internal boolean DFG must be transformed from its blocked state, where no Þring rules are satisÞed to the initial state of the next iteration. This is done according to the *iteration mapping function*. The *iteration mapping function* generates a token sequence for each arc in the internal graph from the token sequences in the blocked graph. This mapping is deÞned by a token mapping function. Tokens not explicitly declared to

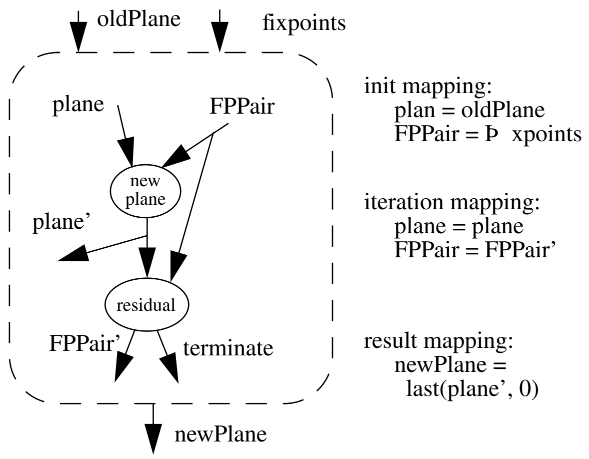

Figure 5. A recursive algorithm for estimating the ground plane relative to the camera.

remain in the boolean DFG will be removed. This is to avoid token accumulation. The internal boolean DFG is executed until the termination condition is satisÞed. When this happens result tokens are generated according to the output mapping as described in the section on hierarchical actors.

The separation of the iteration body and token mapping functions allows a cleaner implementation of iterative functions. Consider the camera control algorithm from section 3. and 5.1. The core of the algorithm is to Þnd the cameras position and heading in the 3D world. This can be done by using the Geographical Position System (GPS) and anchoring objects in an image from the camera with a geographical information system. This is very computational intensive, so it is preferable not to do this often. A less computational intensive approach is to track the cameras movement in the 3D world. This can be done by tracking a set of points in the 2D image from the camera [16]. The calculations assume that all points are located on the ground plane. However this is not the case for all points in the world, i.e. points at the top of a building. An algorithm based on this approach should ignore points not behaving as if they was on the ground plane. An iterative actor which do this is shown in Þgure 5. All actors consume/produce one token on each input/output, except the *plane* input of the *new plane* actor, which reads one token, but the token is not consumed. In each iteration one estimate of the ground plane is calculated. Also the distance between the estimated plane and the Þxpoints are calculated. If there is one Þxpoint with a distance larger than a threshold then the Þxpoint with the largest distance is removed and a new iteration is to be done. The actor *new plane* calculates a new estimate of the ground plane from an earlier ground plane estimate and a set of current and earlier Þxpoints positions. The set of Þxpoints is stored in one token. The actor *resid-*

ual Þnds the Þxpoint with the largest distance from the estimated ground plane. If this distance is larger than a threshold then this point is removed and a token with the remaining Þxpoints are sent to the *FPPair'* arc and a *false* token is sent to the *terminate* arc, otherwise *true* are sent to the *terminate* arc. The Þring of the iterative actor is simple to follow. First in the *initialization mapping function* an earlier ground plane and a set of tracked Þxpoints are mapped to the internal boolean DFG. Next the internal boolean DFG is executed. The actor *new plane* Þ res Þ rst followed by *residual*, resulting in a blocked boolean DFG. If the estimate of the ground plane was based on fixpoint not on the plane then the last token generated on the *terminate* arc is *false* and the set of Þxpoints now believed to be on the ground plane is in the token on the *FPPair'* arc. The internal boolean DFG is to be executed again and the *iteration mapping function* generates the initial state for the next iteration. The token on the *plane* arc is to stay on the arc and the token with the set of Þxpoints on the *PFPair'* arc is mapped to the *FPPair* arc. Now the internal boolean DFG is executed again. This repeats until all Þxpoints is sufÞciently close to the estimate of the ground plane, resulting in a *true* token on the *terminate* arc. The *result mapping function* then maps the last estimated plane to the *newPlane* output arc.

5.3. Token mapping functions

Token mapping functions are used in hierarchical and iterative actors to transfer tokens between the internal arcs and the surrounding. They are also used in iterative actors to change internal state between iterations. This can be seen as a mapping from one set of token sequences of Þ nal length to another set of token sequences of Þnite length, $\{S_0, \ldots, S_n\} \to \{S_0, \ldots, S_m\}$. Each sequence is associated with one arc and the token sequence is the contents of the arcs FIFO queue. In a token mapping function a result sequence is created from concatenations of sequences. The sequences can be original sequences, new sequences or an empty sequence ($\perp$). New sequences are created from individual tokens from the source sequences and new tokens created from constants.

In a *token mapping function* a new token can also be created by wrapping a token sequence. The original token sequence can be recreated by unwrapping token created by a wrapping. This makes it possible to treat some elements as a set, but still have the possibility to apply functions to the individual elements. Consider the plane estimate actor for the camera control algorithm in Þgure 5. In the iterative actor the Þxpoints are treated as a set, encapsulated in one token. However as part of the algorithm the distance for each Þxpoint to a plane is to be calculated. This is done by the hierarchical actor in Þgure 6. During the initialization

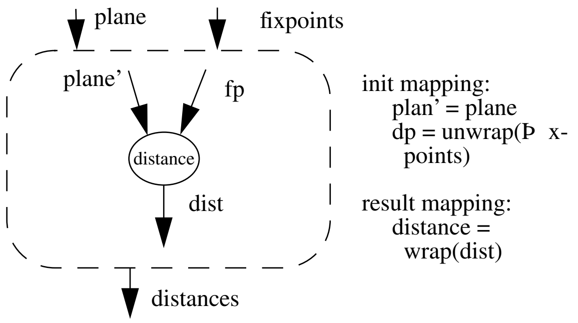

Figure 6. A hierarchical actor for calculation of the distance of all points in a set to one plane.

mapping the set of Þxpoints is unwrapped and placed as a sequence of tokens on the *fp* arc. The *distance* actor will Þ re once for each Þxpoint and the result mapping will later wrap the sequence of distances to one token.

To preserve properties of a graph a token mapping function is said to be *monotonic* if it preserves the relative precedence of all tokens. If token t_k proceeds token t_l in any if the input sequences t_k precedes t_l in all result sequences both tokens are present in.

6. Experimental platform

Originally IPAPI was implemented and tested in the simulator developed for the WITAS UAV project. The WITAS UAV simulator is composed of several components communicating using real-time CORBA. The components of the simulator correspond to the subsystems of the onboard system in addition to components for simulating the environment, the physical behavior of the helicopter and rendering of 3D images. The rendered images are used as input to the image processing algorithms. Currently this implementation is ported to the onboard hardware platform. During this porting the components are modiÞed to coupe with the limitations of the onboard hardware while meeting constraints on timing and stability.

IPAPI has been implemented in Java which make it easy to move between different hardware platforms. To achieve good performance all image processing operations (actors) have been implemented in native methods. The onboard system has one PowerPC processor dedicated for image processing. The native implementations of the image processing actors fully utilize the AltiVec unit of the PowerPC processor. The AltiVec unit is a vector processing unit with 128 bit internal data-paths. It allows SIMD instructions on 16, 8, or 4 parallel elements depending on the number of bits per element (short, int, long or ßoating point). Our experiments show that this division of Java and native C implementation give a negligible overhead compared to implementing the whole run time system in C. Java has the beneÞt of automatic memory management and stable exception handling while still meeting real time constraints for critical tasks [17].

7. Future work

In the WITAS UAV project we have encountered a need for higher order functions. A higher order function is a function that generates another function and in the context of DFGs an actor that generates DFGs. This is similar to a hierarchical actor, but more powerful. Currently we are using algorithms that is composed of calculations at different scales of an image and we need the number of scale levels to be parametric. Today we solve this by allow a hierarchical actor to create its subgraph as it is being created. The creation of a subgraph is deÞned by a Java function, which creates the internal data structure used by IPAPI where the hierarchical actor Þres. To allow people not familiar with the internal structure of IPAPI to use this functionality it is desirable to express higher order functions directly in IP-DFG.

Currently our onboard system for the WITAS helicopter has one processor dedicated for image processing. Thus IPAPI is implemented for single processor. We plan to extend the onboard system with more processors in the near future. For this purpose IPAPI will be extended with algorithms for distribution of image processing algorithms.

8. Acknowledgments

We would like to thank Johan Wiklund and the people in the the computer Vision Laboratory at Linköping University for their share of work with implementing IPAPI.

9. References

[1] P. Doherty et. al. "The WITAS Unmanned Aerial Vehicle Project", *Proceedings of the 14th European Conference on ArtiÞ cial Intelligence*, 2000

[2] AUVS - The Association od Unmanned Vehicle Systems. http://www.auvsi.org/

[3] G. Buskey, G. Wyeth and J. Roberts. "Autonomous helicopter hover using an artiÞcial neural network", *Proceedings of Robotics and Automation 2001 ICRA. IEEE International Conference on, vol. 2*, 2001

[4] J. Evans, G. Inalhan, Jung Soon Jang, R. Teo,C.J. Tomlin, "Dragonßy: a versatile UAV platform for the advancement of aircraft navigation and control", *Digital Avionics Systems, 2001. DASC. 20th Conference, Volume: 1*, 2001

[5] E. A. Lee and Thomas M. Parks. "Dataßow process networks", *Proceedings of the IEEE, vol. 83, no. 5, pp. 773-801,* May, 1995

[6] E..A. Lee and David G. Messerschmitt. "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE transactions on computers vol. C-36,no 1*, January 1987

[7] J.T. Buck. "Scheduling dynamic dataßow graphs with bounded memory using the token ßow model", *Ph.D. thesis from University of California*, 1993

[8] E. A. Lee. "Consistency in Dataßow Graphs", *IEEE transactions on parallel and distributed system, vol.2 no. 2* April 1991

[9] Arvind and K. P. Goestelow, "Some Relationships between Asynchronous Intepreters of a Dataßow Language", *Formal Description of Programming Languages, IFIP Working Group 2.2* 1977

[10] Arvind and K. P. Goestelow, "The U-Intepreter", *IEEE Computer 15(2)*, February 1982

[11] E. A. Lee, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M01/11, University of California, Berkeley*, March 6, 2001.

[12] C. S. Williams and J. R. Rasure, "A visual language for image processing", *In Proc. 1990 IEEE Workshop on Visual Languages, pages 86--91*, 1990.

[13] R. Lauwereins, M. Engels, M. Ade and J.A. Peperstraete, "Grape-II: a system-level prototyping environment for DSP applications", *IEEE Computer 28(2)*, February, 1995

[14] R. Szymanek, K. Kuchcinski, "Task Assignment and Scheduling under Memory Constraints", *Euromicro*, 2000

[15] P. K. Murthy and S. S Bhattacharyya, "Shared memory implementations of synchronous dataßow speciÞcations", *Proceedings of Design, Automation and Test in Europe Conference and Exhibition 2000*, 2000

[16] P. Forssén, "Updating Camera Location and Heading Using a Sparse Displacement Field", *technical report Linköping University LiTH-ISY-R-2318*, November 2000

[17] A. Nilsson and T. Ekman, "Deterministic Java in Tiny Embedded Systems", *Proceedings of the fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC),* 2001