Block Diagrams as a Syntactic Extension to Haskell

Ben Denckla[1] and Pieter J. Mosterman[2]

[1] Denckla Consulting, 1607 S. Holt Ave., Los Angeles, CA 90035, USA
bdenckla@alum.mit.edu
[2] The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, USA
pieter.mosterman@mathworks.com

Abstract. Often, the semantics of languages are defined by the products that support their usage. The semantics are then determined by the source code of those products, which often is a general-purpose programming language. This may lead to complications in defining a clean semantics, for example because imperative notions slip into a declarative language. It is illustrated how block diagrams can be translated into Haskell to define the semantics of a graphical language in terms of a textual programming language. This also allows the use of block diagrams as a syntactic extension to Haskell and the use of Haskell as an action language in block diagrams. Imperative notions can then be included from the declarative perspective of Haskell, which is more constrained and less prone to resulting in complicated semantics of interaction and combination of the imperative and declarative.

1 Introduction

The increasing application of computational power in engineered systems such as automobiles, consumer electronics, and aircraft has resulted in a staggering complexity that has proven difficult to negotiate with conventional design approaches. For example, high-end automobiles may now employ up to 80 microprocessors that largely interact with each other through the automobile networks such as the controller area network, CAN [1]. The discrete nature of the software that is running on these microprocessors abandons the notions of continuity that are inherent in the more traditional physics design problems [20] and this has put forward the need for different design approaches.

To successfully tackle the computational complexity of modern engineered systems, Model-Based Design [3] is increasingly being adopted, which addresses the software complexity at a computational model level. This allows circumventing the practice of designing functionality directly at the level of computation, such as in assembly or programming languages. The power of those languages often leads to complexity in a design that is difficult to comprehend, and, therefore, results in errors. To curtail this problem, typically programming styles are mandated and code structure of too high complexity (e.g., cyclomatic complexity) is disallowed.

Though styles have been successful to mitigate the error-prone nature of design at a programming language level, it has not addressed the design problem satisfactorily. For example, the design of the F/A-22 has still not been completed in spite of significant budget increases, which has large been caused by software producibility problems [23].

Rather than designing the structure of computation in a programming language, Model-Based Design allows the use of high-level and domain-specific languages. These languages can be tailored to capture the semantic notions of the domain in which the design problem needs to be solved. This allows the design engineers to work with a language that is intuitive and close to their understanding of the problem, and, therefore, is more efficient and less error-prone.

There are a number of essential aspects to this approach:

- The design of tailored languages needs to be an efficient process itself.
- The evolution of a given language needs to be supported.
- The domain-specific language needs to be automatically transformed into a structure of computation.

The field of Computer Automated Multiparadigm Modeling (CAMPaM) [19] aims to address these aspects by establishing a framework to reason about models of systems at multiple levels of abstraction, transforming between models in different languages, and providing and evolving modeling languages. This paper concentrates on the language aspect and shows how a graphical model can be designed as syntactic extension to a textual model. It furthermore touches upon the differences between imperative and declarative semantics that is orthogonal to the language modality and it discusses the interaction between the two.

Section 2 provides a brief introduction to CAMPaM to establish a common vocabulary and show where this work fits into the overall framework. Section 3 discusses the graphical and textual modalities and their characteristics. Section 4 concentrates on block diagrams specifically and presents the block diagram syntactic extensions to Haskell [15], a functional and declarative language. Section 5 presents the conclusions of this work.

2 Computer Automated Multiparadigm Modeling

The basic CAMPaM aspects are introduced and important notions that it entails are defined.

2.1 Aspects of CAMPaM

Previous work [19] has established CAMPaM as the field that provides a framework and computational methods to relate and combine models. This requires handling the different levels of abstraction that are being used for the models as well as the different formalisms that are being employed. Note that the levels of abstraction and different formalisms are orthogonal, although often a different formalism is employed for a different level of abstraction. This makes formalisms

a first class element of study, and establishes the modeling of formalisms as an essential activity, where a formalism can be thought of as a having a syntax and semantics [13].

All in all, this puts forward the following aspects of CAMPaM:

- multiple levels of abstraction,
- multiple formalisms,
- formalism modeling, which includes
  - modeling the syntax, and
  - modeling the semantics.

This paper concentrates on formalisms, in particular on combining and relating formalisms and the elements of a formalism.

2.2 Definitions

To anchor the work presented in this paper, a set of definitions are given that are derived from discussions during a number of workshops on Computer Automated Multiparadigm Modeling[3] [2]. A detailed description of this framework and a set of agreed upon definitions, which may differ from the following, is forthcoming.

The Ogden/Richards semiotic triangle[4] can be thought of as establishing a relationship between concrete syntax, abstract syntax, and semantics. The concrete syntax is the actual referent. The abstract syntax is a symbol representing the referent. The semantics is the thought associated with the referent. This work restricts the referent to be a sentence.

Definition 1 (Sentence) *A presentation of information.*

A sentence in a given language is presented in its concrete syntax.

Definition 2 (Concrete Syntax) *The presentation of a sentence is in the concrete syntax of a language.*

A sentence can be represented as an element of an abstract syntax.

Definition 3 (Abstract Syntax) *The abstract syntax contains representations of sentences.*

In general, an abstract syntax consists of a set of symbols and their possible combinations. This then requires the abstract syntax to be comprised of entities, the symbols, and relations, the combinations of symbols.

A sentence relates to information by invoking a thought when it is interpreted. This thought is the semantics of the sentence.

Definition 4 (Semantics) *The semantics of a sentence is the thought associated with that sentence.*

[3] http://moncs.cs.mcgill.ca/people/mosterman/campam/
[4] http://en.wikipedia.org/wiki/Semiotic_triangle

A sentence is an element of a set of sentences that is called a language.

Definition 5 (Language) *A language is a set of sentences.*

The sentences in an abstract syntax relate to a set of semantics that is called the semantic domain of the language that represents the set of sentences.

Definition 6 (Semantic Domain) *The semantic domain of an abstract syntax is the set of thoughts associated with the elements in the abstract syntax.*

The semantic domain then consists of the intuitive notions that can be represented by the elements of the abstract syntax.

To associate a thought with a sentence is to give it a meaning. This corresponds to a mapping of the sentence to a semantics.

Definition 7 (Meaning) *The meaning of a sentence in the abstract syntax is a projection into the semantic domain of the language.*

The combination of an abstract syntax, its semantic domain, and a meaning for each of the elements in the abstract syntax is called a formalism.

Definition 8 (Formalism) *A formalism consists of an abstract syntax, a semantic domain, and a meaning.*

A semantic domain can be shared completely or partially by many formalisms.

A sentence can be translated by changing its syntax.

Definition 9 (Translation) *The translation of a sentence is another sentence in another formalism.*

An abstract syntax can be modeled by another or the same abstract syntax, where the model represents a set of sentences, or language. The language model is called a metamodel and its meaning is the abstract syntax of the language that is modeled.

Definition 10 (Metamodel) *The meaning of a sentence in a metamodel language is an abstract syntax.*

This metamodel can be a grammar or an enumeration and may be generative.

The metamodel does not model a formalism, as it does not capture the meaning of the abstract syntax. Because the meaning is a relation between abstract syntax and thought, no explicit model of meaning can be provided. Instead, the meaning is modeled implicitly by a relation between two abstract syntaxes (that may be the same), an interpretation, where the range abstract syntax is more intuitive than the domain abstract syntax.

3 Textual and Graphical Languages

The concrete syntax of a language is the presentation of the abstract syntax. This presentation is often classified as either graphical or textual. At the abstract syntax level, there is no principled distinction between the two.

3.1 Textual Versus Graphical Languages

Whether textual or graphical syntax is preferred depends much on the application. Textual syntaxes of many written natural languages, programming languages such as FORTRAN and C, and modeling languages such as Modelica™ [11] have proven to be useful and successful. Similarly, graphical syntaxes of modeling languages such as bond graphs [17], Petri nets [4], and block diagrams [6] have been very successful in their respective usages as well.

At the core, the difference between graphical languages and textual languages is that the former often reference expressions by lines (directed or undirected) whereas the latter reference expressions by symbols. The use of lines allows direct display of referencing relations between expressions, which helps in quickly building an understanding of the model. The drawback of this explicitness is that it becomes costly in terms of visual clutter when many such references are present. Though the use of symbols for referencing prevents the graphical clutter, it may lead to clutter of the namespace with named references.

Considering graphical models that represent programs, one could say the value of graphical models is that they allow the graphical structure of a program to be expressed in a more direct form. Textual syntaxes have to represent the program (term graph) mostly as a tree, and further, have to represent that tree as a sequence of lexemes. They represent a graph mostly as a tree by adding some context-sensitive constraints to a context-free grammar. They represent a tree as a sequence through mechanisms such as parentheses and precedence. This is not to say that the more direct, graphical form of expression offered by graphical models is always preferable, on the contrary. Most textual programs can be seen as compact, elegant representations of what would be a very convoluted graph. On the other hand, there are times when it would be clearer to see the graph directly, and this is what graphical models offer.

It is also possible to imagine a language that has a single notion of reference with two different ways to view it: one by arrow and one by symbol. So, rather than arguing a purified approach, a mixture of these approaches is advocated and typically provided in industrially successful products. For example, the GoTo and From blocks that Simulink® [22] provides allow reference by symbol in addition to the graphical referencing inherent in block diagrams. Another example of this mixture of referencing is Subtext [7] and, in a more limited way, the many integrated development environments (IDE) that allow the user to, e.g., jump from a site where a symbol is used to the symbol definition.

In general, graphical languages are never full languages; they are always combined with some other language that defines what individual entities and relations mean. For example, the meaning of a block in a block diagram is typically defined by a name inside the block (e.g. "H"), possibly combined with the use of a shape other than a rectangle for a block, (e.g., the use of a triangle instead of a rectangle, to indicate a gain, with the gain factor indicated by a numeral such as "-1" inside the block).

3.2 Related Work

Ellner and Taha [9, 10] provide a good introduction to the unneeded gap between block diagram and textual languages. They also provide a promising way to formally bridge this gap with a visual multi-stage calculus called PreVIEW. The work presented in this paper seeks to bridge this gap in a different but complementary way. It does not intend to provide a visual language equivalent to a textual one. Rather, a textual language (Haskell) is only extended with visual features. Additionally, the work presented in this paper does not address multi-stage programming and it concerns a full language because of the use of Haskell. This is in contrast to the use of a minimal calculus that is suitable for theoretical work, as presented by Ellner and Taha.

The Subtext language [7] is notable for the way it bridges the textual/graphical gap. Names (identifiers) are present in Subtext but are more like comments upon its fundamental concepts of sources and references connected by links, which are represented explicitly. The HOPS language is notable [16] for its implementation of a Haskell-like language using a term graph instead of a textual representation. The Vital language [12] is notable for its implementation of a Haskell environment in which data structures can be viewed and manipulated as diagrams.

The particular idea of implementing a block diagram language as syntactic extensions to Haskell is discussed and partially implemented by Reekie [21]. This is closely related to the approach of implementing a domain-specific language by embedding it in a general-purpose language [14]. However, in this case, the block diagram language needs new syntax, so it cannot, in the strictest sense, be called embedded. Implementing a new language by extending syntax or by embedding can be seen as part of Landin's 40-year-old program to avoid reinvention of general-purpose language capabilities when inventing a new language [18]. Of course this is not always possible and desirable, but the work presented in this paper intends to illustrate that it is in the case of block diagrams.

Again, the point made is that it is not imperative to make a choice: the approaches can be mixed by extending a textual language with block diagrams.

3.3 Semantics of Block Diagrams

Block diagrams have become very popular among control system engineers to support their design efforts. In particular, the support for computational simulation by block diagram based products such as Simulink has been an important enabler for this success. Furthermore, the constraints that are enforced by block diagrams address the specific needs of the control system discipline. The corresponding limitations in expressiveness, as compared to general purpose programming languages, prevent the user from making mistakes that can happen in a general language of computation. For example, Simulink supports only a very stylized form of recursion which cannot cause infinite loops or stack overflows.

On the other hand, the semantics of block diagram languages are often informal, which suffices for many users who infer the semantics through trial and error or by example, and do not rely on a documented semantics.

Because a formal semantics is not required for most successful applications of a language and to achieve expedient and convenient implementation of interpreters, often a programming language such as C is used as the language for defining the semantics. For example, the semantics of Modelica is defined by the source code of its supporting products. It is evident that semantic discrepancies between products that support the same language are inevitable.

Furthermore, the freedom of expression that programming languages such as C allow can lead to semantic inconsistency when adding language features. For example, in a pure interpretation, block diagrams are a declarative language, but the use of an imperative language to define the semantics of new language elements has allowed introducing imperative notions in the form of the Merge block in Simulink. This block produces as output the last computed input, and, therefore, the outcome becomes dependent upon the execution order. In case of the Merge block, the use of imperative notions has lead to a very powerful language construct that greatly aids the modeling task. In other words: "One user's feature is another user's bug."

Another implication of the use of an imperative language to define the semantics of a declarative language is that it becomes difficult to integrate imperative notions with the declarative language in a consistent manner. In the implementation, there is no distinction between the two, and, therefore, maintaining a consistent separation and interaction becomes problematic.

History has shown, though, that such limitations constitute no impediment to the success of a language (natural languages included). However, in applications where categorical evaluations are required, a clear definition of semantics is often required. This paper aims to illustrate that such a definition can be achieved for block diagrams. In particular, the use of a declarative language such as Haskell to define the semantics prevents many of the issues discussed. Furthermore, because block diagrams rely on some other language to provide the symbolic environment in which blocks are defined, it is a short leap to make this environment a scope in Haskell.

4 Block Diagrams and Haskell

Building on previous work [5], BdHas is presented as Haskell [15] plus syntactic extensions for block diagrams. A block diagram can be used inside a Haskell expression, and a Haskell expression can be used inside a block. The interpretation of a BdHas program is the Haskell code that results from translating all of its block diagrams to pure Haskell. Before describing this translation, an introduction to BdHas is provided by means of an example.

4.1 Introduction to BdHas

The example used to introduce BdHas is that of a simplified token ring bus [8]. In each clock cycle, the stations on the bus "decide amongst themselves" which station (if any) will use the bus. The decision is made in the following manner:

- A station has permission to use the bus if it owns the token or has been passed permission to use the bus.
- A station with permission to use the bus grants itself the bus if it has requested the bus.
- Otherwise it passes permission to use the bus to the clockwise-next station.
- At the end of each clock cycle, token ownership is transferred to the clockwise-next station.

Figure 1 shows the arbiter as a hierarchical block diagram in BdHas. The functions *sdelay*, *sand*, *sor*, and *snot* are assumed to be defined elsewhere to be a unit delay and stream versions of the boolean operators "and," "or," and "not," respectively. The symbol *r* is short for "bus requested," *g* for "bus granted," *pi*/*po* for "permission in/out," *ti*/*to* for "token ownership in/out," and *iot* for "initially owns token."

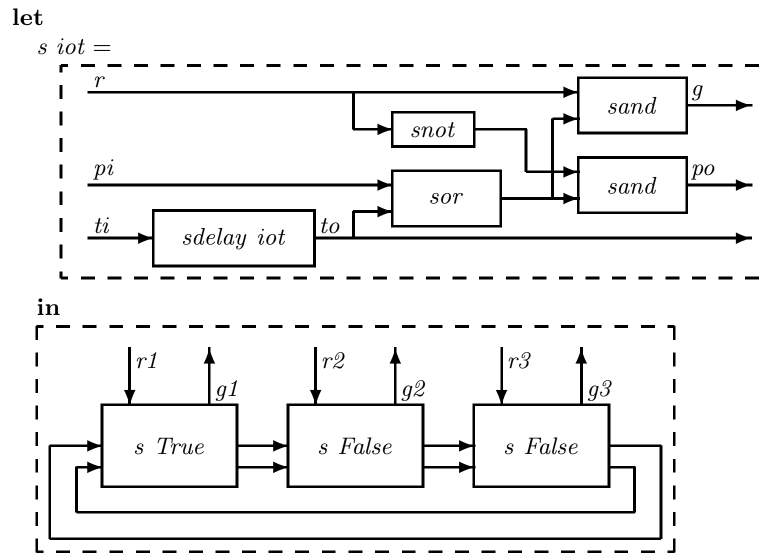

Fig. 1. Token ring arbiter in BdHas

The symbols labeling arrow tails (*r*,*g*,*pi*, etc.) are comments to aid understanding; they have no semantic significance as they would in a textual program.

To execute this BdHas program, it is passed to a program that translates the syntactic block diagram extensions into pure Haskell, yielding the code in Fig. 2. Note that some automatically generated identifiers have been renamed to make them more meaningful.

Although hand-written code may be quite different, the automatically generated code is a reasonable implementation and thus conveys a notion of how advantageous the use of a block diagram can be over textual code.

```
let
  s iot =
      λ(r, pi, ti) →
        let
          g = sand (r, o)
          po = sand (n, o)
          to = sdelay iot ti
          o = sor (pi, to)
          n = snot r
          in
          (g, po, to)
  in
  λ(r1, r2, r3) →
    let
      (g1, po1, to1) = s True (r1, po3, to3)
      (g2, po2, to2) = s False (r2, po1, to1)
      (g3, po3, to3) = s False (r3, po2, to2)
      in
      (g1, g2, g3)
```

Fig. 2. Token ring arbiter translated from BdHas to Haskell

4.2 Translating Block Diagrams to Haskell

The translation from BdHas to Haskell proceeds roughly as follows. Give a unique identifier (ID) to each arrow tail junction. An arrow tail junction (ATJ) is a point where one or more arrow tails coincide. Translate each block to a declaration of the form $y = f\ u$ where

- y is a tuple of the IDs of the ATJs on the block,
- u is a (possibly empty) tuple of the IDs of the ATJs for each arrow whose head is on the block, and
- f is the translation of the BdHas expression inside the block.

Translate the diagram to

- a λ expression binding a (possibly empty) tuple of the IDs of the ATJs that are not on any block, where the body of this λ expression is
  - a **let** expression containing the (possibly empty) list of declarations from the block translations, where the body of this **let** expression is
    - a tuple of the IDs of the ATJs for each arrow whose head is not on any block.

So, in general, block diagrams translate to expressions of the form shown in Fig. 3, where i is a meta-variable ranging over the IDs of the ATJs and e is a meta-variable ranging over the expressions that result from translating the contents of the blocks.

$\lambda(i, i, ...) \rightarrow$
let
$(i, i, ...) = e\ (i, i, ...)$
$(i, i, ...) = e\ (i, i, ...)$
...
in
$(i, i, ...)$

Fig. 3. General form of block diagrams translated to Haskell

4.3 Integrating Imperative Notions

The desire to integrate imperative notions in block diagrams has been discussed in Section 3.3. In Haskell, it is straightforward to create a pseudo-imperative domain-specific embedded language in which integers can be added and an "instruction count" of the tally of additions at a point in the instruction sequence can be retrieved.

For example, with a few lines of helper code not shown, the following function

$addSeq = \textbf{do}$
$x \leftarrow add\ 1\ 2$
$c \leftarrow get$
$y \leftarrow add\ x\ 3$
$z \leftarrow add\ y\ c$
$return\ z$

returns 7. But, if the line $c \leftarrow get$ is moved to one line later, i.e.,

$addSeq = \textbf{do}$
$x \leftarrow add\ 1\ 2$
$y \leftarrow add\ x\ 3$
$c \leftarrow get$
$z \leftarrow add\ y\ c$
$return\ z$

it will return 8.

5 Conclusions

Domain specific languages help efficient and effective design of systems with a significant computational component. A classification into textual and graphical languages can be made. Whereas textual languages often lack a graphical component, graphical languages typically include textual extensions.

The definition of semantics for a language has been discussed as a syntactic translation. In many languages, the semantics are defined by the source code of a product that supports each language. This often leads to complications because of the implementation freedom that general-purpose programming languages such as C provide.

This paper discussed the use of Haskell to provide semantics for block diagrams instead. The declarative nature of Haskell renders it well-suited for this

purpose. Furthermore, this provided the basis for including block diagrams into Haskell as a syntactic extension. The translation of block diagrams into Haskell results in a uniform pure Haskell representation. Additionally, this facilitates the use of a full programming language as an action language for block diagrams.

This paper has further briefly illustrated how imperative notions may be included in Haskell. Rather than using imperative programming languages to define declarative semantics, this may result in more rigorous and consistent definition of mixed imperative/declarative languages.

A working implementation based on a textual input of the block diagrams was presented while a visual editor is not yet available.

6 Acknowledgment

The authors wish to acknowledge the other attendees of the 2004 through 2006 editions of the *International Workshop on Computer Automated Multi-Paradigm Modeling*:[5] Jean-Sébastien Bolduc, Peter Bunus, Gary Godding, David Hill, Stephen Neuendorfer, Hans Vangheluwe, Mamadou Traore, Thomas Kühne, Hessam Sarjoughian, Vasco Miguel Moreira do Amaral, Adam Cataldo, Jerome Delatour, Jean-Marie Favre, Holger Giese, Anneke Kleppe, Juan de Lara, Tihamér Levendovszky, Jie Liu, Alexandre Muzy, and Ernesto Posse for their help in developing the CAMPaM framework.

References

1. CAN specification. Technical Report, 1991. Robert Bosch GmbH.
2. CAMPaM '06 Attendees. CAMPaM 2006 workshop position statements. Technical Report SOCS-TR-2006.2, McGill School of Computer Science, 2006.
3. Paul Barnard. Graphical techniques for aircraft dynamic model development. In *American Institute of Aeronautics and Astronautics (AIAA) Modeling and Simulation Technologies Conference and Exhibit*. Providence, Rhode Island, August 2004. CD-ROM, paper number AIAA-2004-4808.
4. René David and Hassane Alla. *Petri Nets & Grafcet*. Prentice Hall Inc., Englewood Cliffs, NJ, 1992. ISBN 0-13-327537-X.
5. Ben Denckla, Pieter J. Mosterman, and Hans Vangheluwe. Towards an executable denotational semantics for causal block diagrams. In *Proceedings of The 5th OOPSLA Workshop on Domain-Specific Modeling*, San Diego, CA, October 2005. ISBN 951-39-2202-2.
6. Richard C. Dorf. *Modern Control Systems*. Addison Wesley Publishing Co., Reading, MA, 1987.
7. Jonathan Edwards. Subtext: uncovering the simplicity of programming. In *Proceedings of OOPSLA '05*, pages 505–518. ACM Press, 2005.
8. Stephen A. Edwards and Edward A. Lee. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program.*, 48(1):21–42, 2003.
9. Stephan Ellner. PreVIEW: An untyped graphical calculus for resource-aware programming. Master's thesis, Rice U., 2004.

[5] http://moncs.cs.mcgill.ca/people/mosterman/campam/

10. Stephan Ellner and Walid Taha. The semantics of graphical languages. In *Informal Proceedings of the Workshop on Designing Correct Circuits*, 2006.
11. Hilding Elmqvist *et al.* Modelica^{TM}–A unified object-oriented langauge for physical systems modeling: Language specification, December 1999. version 1.3, http://www.modelica.org/.
12. Keith Hanna. http://www.cs.kent.ac.uk/projects/vital/index.html.
13. David Harel and Bernhard Rumpe. Modeling languages: Syntax, semantics and all that stuff. Technical Report MCS00-16, The Weizmann Institute of Science, 2000.
14. Paul Hudak. Modular domain specific languages and tools. In *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.
15. Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge U. Press, April 2003.
16. Wolfram Kahl. The term graph programming system HOPS. In *Tool Support for System Specification, Development and Verification*, pages 136–149, March 1999.
17. D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley and Sons, New York, 2 edition, 1990.
18. P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.
19. Pieter J. Mosterman and Hans Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 80(9):433–450, September 2004.
20. Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, Massachusetts, 1961.
21. H. J. Reekie. *Realtime Signal Processing: Dataflow, Visual and Functional Programming*. PhD thesis, U. of Technology at Sydney, Australia, 1995.
22. Simulink®. *Using Simulink®*. The MathWorks, Inc., Natick, MA, March 2006.
23. Michael Sullivan. TACTICAL AIRCRAFT–F/A-22 and JSF acquisition plans and implications for tactical aircraft modernization. Technical Report GAO-05-519T, United States Government Accountability Office, April 2005.