2009 24th Annual IEEE Symposium on Logic In Computer Science

Applications of Game Semantics: From Program Analysis to Hardware Synthesis

Dan R. Ghica
School of Computer Science
University of Birmingham, UK

Abstract

After informally reviewing the main concepts from game semantics and placing the development of the field in a historical context we examine its main applications. We focus in particular on finite state model checking, higher order model checking and more recent developments in hardware design.

1. Chronology, methodology, ideology

Game Semantics is a *denotational semantics* in the conventional sense: for any term, it assigns a certain mathematical object as its meaning, which is constructed compositionally from the meanings of its sub-terms in a way that is independent of the operational semantics of the object language. What makes Game Semantics particular, peculiar maybe, is that the mathematical objects it operates with are not sets and functions, as the reader familiar with denotational semantics in the tradition of Scott and Strachey might expect [81].

To understand how Game Semantics works it is perhaps easiest to start with the fundamental notion of *observable action*: the simplest kind of event that a program, or term in general, can be involved in (as producer or consumer) during the course of its execution. For example, the program that interacts with its environment in the simplest possible form can be implicated in two events: starting execution and finishing it. If we construe the notions of *input* and *output* broadly enough, the former is an input and the latter is an output. Then, from this point of view there can only be two semantically distinct behaviours of this kind, one that starts then finishes, and one that starts and does not finish. A marginally more complex interaction would be the case of a program that computes a boolean value for which it must be able to produce two distinguishable results. In this class of programs we should be able to find three distinct behaviours, associated with the following three sequences of observable actions: $start \cdot true$, $start \cdot false$, $start$.

The idea of observable action can be extended to programs of higher-order type, inductively on the structure of the type. The observable actions of a function are those of the function body and those of its arguments. For example, a (call-by-name) function of type $(\mathsf{bool} \times \mathsf{bool}) \to \mathsf{bool}$ has nine observable actions. They are the three distinct actions for boolean programs taken thrice, once for each argument and once for the result. Let us tag argument actions with $a1$ and $a2$ and result actions with r to prevent confusion. The set of possible behaviours is richer now, including for example interactions such as

$$start_r \cdot start_{a1} \cdot true_{a1} \cdot start_{a2} \cdot true_{a2} \cdot true_r \quad (1)$$

or

$$start_r \cdot start_{a2} \cdot true_{a2} \cdot start_{a1} \cdot true_{a1} \cdot true_r. \quad (2)$$

Both interactions above represent a computation in which a function starts executing, evaluates one of the arguments, then the other, then produces a result. In both cases the arguments produce *true*, as does the function. However, these two interactions arise in two semantically distinct functions, which evaluate their arguments in different orders.

In a programming language with side-effects it is of course relevant in what precise order a function evaluates its arguments. And modelling a (program) function as a schedule of interactions rather than as a (mathematical) function makes this order perspicuous. Even modelling concurrent evaluation is relatively straightforward using this approach:

$$start_r \cdot start_{a1} \cdot start_{a2} \cdot true_{a1} \cdot true_{a2} \cdot true_r. \quad (3)$$

With the hope that the basic computational intuitions are now established, we will introduce the game-semantic terminology for the concepts above:

legal moves represent the basic atomic actions occurring in computation.

arenas represent all the moves that a term of a given type can perform, along with some basic structure. In the

1043-6871/09 $25.00 © 2009 IEEE
DOI 10.1109/LICS.2009.26

IEEE computer society

example above it is intuitively clear that a $true$ action can only happen subsequent to a corresponding $start$ action. This fundamental causal connection between moves is called *enabling* and is part of the arena structure, just like the following two concepts.

players represent the input-output polarity of moves. An input action, i.e. an action of the computational context, is called an *Opponent* move, whereas an output action, i.e. an action of the program, is called a *Proponent* move.

question and answer are two labels for legal moves. Moves that are *active*, that *elicit information*, are called questions (*start* above) whereas moves that are *passive*, that *provide information*, are called *answers* (*true, false* above).

plays are models of particular computations, trace-like schedules of interactions where the events are moves.

strategies model the way the Proponent, i.e. the term, interacts with the Opponent, i.e the context, by making moves that are determined by the play up to that point. Strategies are the mathematical objects that are assigned as denotations for terms.

Computation in a programming language cannot proceed haphazardly. Besides the basic causal relations on moves expressed by enabling there are also a number of constraints dictated by both the evaluation strategy of the language and the computational primitives it uses. These constraints induce a notion of *legality* on plays. For example, a language which is sequential requires an *alternation* of turns between the two players, which makes play such as the one in (3) illegal, because it represents an impossible interaction in a sequential language. Other constraints, such as "*innocence*," can be formulated at the level of strategies, indicating certain limitations on the way in which the Proponent can interpret the information available, i.e. the history of computation, in order to decide its next move.

It is desirable that a semantic model for a language is fully complete, i.e. every object in the model corresponds to a term, and fully abstract, i.e. two programs have the same denotations precisely when they are observationally equivalent. These notions are important because they are the only objective measures of precision for a semantic model. In the case of Game Semantics, finding the right constraints on plays and strategies can lead to precise models that satisfy these desiderata.

The game-theoretical metaphor reflected in the terminology has historical origins, and a first encounter with the topic of Game Semantics may leave a casual reader baffled and perhaps unsatisfied. In conventional game theory the notions of winning, losing and pay-off are paramount; in Game Semantics they are not used at all. The terminology can only be understood from a historical perspective.

The first semantic model based on "dialogue games" was given by Lorenzen, for intuitionistic logic [62]. A formula is seen as a game between a Proponent, who aims to establish its truth, and an Opponent, who aims to invalidate it. The rules of the game are in this case dictated by the formula and validity is defined as the existence of a *winning strategy* for Proponent. Lorenzen's ideas instigated a new school of logic [32], but their view of semantics remained static, providing no mechanism for composition of games which, at a logical level, corresponds to cut-elimination.

Girard's Linear Logic [45] and the resource-sensitive view of inference that linearity represents was quickly and widely perceived as being profoundly relevant to computer science. Girard's linear implication $A \multimap B$ corresponds, through the Curry-Howard correspondence, to a function space in which the argument is accessed precisely once, a view difficult to express in the conventional mathematical sense, which regards arguments of functions atemporally. This more refined view of logic led Blass to revisit Lorenzen's dialogical model, but this time around with a due emphasis on modelling proof dynamics [18]. Some technical problems in Blass's game model were corrected by Abramsky and Jagadeesan [6], who also established stronger connections with the dynamics of linear logic via its so-called "*Geometry of Interaction*" [7].

Although aimed at logic, many intuitions in game models for linear logic are of an explicitly computational nature, where computation is seen essentially as interaction. The computation-as-interaction paradigm was actually already established independently in programming language semantics, connected to efforts for understanding *sequentiality*. In his historical survey [26], Curien indicates that the earliest work on this was that of Vuillemin and Milner, subsequently generalised by Kahn and Plotkin to concrete data structures [20]. The idea was further refined by Curien into a model in which functions are equipped with schedules of interaction, a model which was then turned into the programming language CDS [17].

These two sources of insight (dialogical games and computation as interaction) came together in the work of three research teams who independently solved the long-standing open problem of definability for the prototypical language PCF [79] — to wit, what computations can be defined in a purely functional language? Along with Nickau [72], the seminal work of Abramsky, Jagadeesan and Malacaria [8], and Hyland and Ong [52] is the starting point of Game Semantics in its present formulation.

A burst of activity in developing fully abstract game models followed in the wake of this important result, much of it driven by Abramsky and his students McCusker and Laird. Game-semantic models for local-

state [10], call-by-value [11], polymorphism [50], recursive types [9], general references [5], control [56], exceptions [57], non-determinism [47], probabilistic computation [27], message-passing concurrency [58] and shared-variable concurrency [42] followed.

Research in game semantics remains very active and models of increasingly realistic programming languages are being produced. For example, in recent years languages with nominal features (in the sense of [78]) have been studied [4, 82, 59]. Interest in foundational aspects of Game Semantics was always high, such as understanding the connections between strategies and abstract machines [29], or finding better formulations of game-semantic constructs and constraints [28, 66, 46, 67].

2. Concrete representation of game models

The process-like nature of game semantics was clear from the outset, Hyland and Ong giving an early encoding of their game model of PCF into the polyadic pi-calculus [51]. Although their paper foresees the advantages that a concrete representation of the model would offer, the particular encoding they use turned out not to be a popular platform for program analysis, mainly due to its complexity.

A more accessible encoding of the game model was used by Ghica and McCusker [39], who focused on the second-order fragment of Idealized Algol [10]. By using a language with state, many simple yet challenging examples could be used to illustrate the usefulness of the model, and by restricting to a low-order fragment some of the more complex features of the game model ("justification pointers") could be actually omitted. Concretely, arenas are alphabets (sets of symbols) and strategies can be represented by regular languages over these alphabets. Constants of the language have as denotations strategies that can be represented using regular languages. For example, the denotation of the sequential composition operation $\mathsf{seq} : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3$ is the regular (finite) language $r_3 r_1 d_1 r_2 d_2 d_3$. Indeed, sequential composition seen as a function interacts with its context as follows: r_3 starts the sequencing, r_1 starts the first argument, d_1 is the first argument terminating, r_2 starts the second argument, d_2 is the second argument terminating, d_3 is the termination of the sequencing operation. The denotation of iteration $\mathsf{while} : (\mathsf{bool}_1 \times \mathsf{com}_2) \to \mathsf{com}_3$ is

$$r_3(q_1 t_1 r_2 d_2)^* q_1 f_1 d_3. \tag{4}$$

It evaluates repeatedly and sequentially the first two arguments while the first argument produces t(rue) and it terminates when the first argument produces f(alse).

To construct denotations for terms the key operation is that of *composition* of strategies. In general, the details of

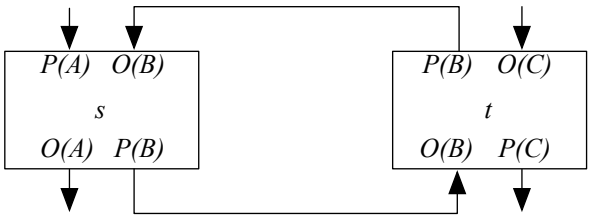

Figure 1. Composition by mutual feedback

the rigorous definition of composition in a game model are complex and may vary depending on the model, but the basic intuition is that of "mutual feedback" from the generalized Geometry of Interaction construction [2], as in Fig. 1. Seen as processes, strategies s and t which share an arena B compose by sending their Proponent moves to the other strategy, which will accept them as Opponent moves. In effect this achieves both a synchronisation of the two strategies on the moves occurring in the shared arena B and the hiding of arena B from the interface of composite strategy. In its concrete representation, strategy composition can be realised using only operations that preserve regularity, so that denotations of terms are always regular languages.

The game model is denotational, therefore it can model *open* terms, i.e. terms with free identifiers. The denotation assigned to free identifiers is based on the following intuition: in our programming language, a function can call its arguments in any order any number of times. However, since our language lacks concurrency or control each argument must start being evaluated only when the argument being previously evaluated has terminated. In the game model this intuition is spelled out as a collection of simpler constraints on the legality of plays. Thus, a free identifier of type $(\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3$ can only have behaviours of shape $r_3(r_1 d_1 + r_2 d_2)^* d_3$. By contrast, the same type in a concurrent language leads to a strategy representable by the language

$$r_3(r_1 d_1 \bowtie r_2 d_2)^{\circledast} d_3, \tag{5}$$

where $\bowtie$ is the shuffle operation on languages and $\circledast$ its iterated closure [41].

However, the denotation of the free identifier is slightly more complicated, because it must be consistent with the type signature of the term

$$f : (\mathsf{com}'_1 \times \mathsf{com}'_2) \to \mathsf{com}'_3 \vdash f : (\mathsf{com}_1 \times \mathsf{com}_2) \to \mathsf{com}_3,$$

which requires moves both to the left and to the right of the $\vdash$. The actual strategy is

$$r_3 \overline{r}'_3 (r'_1 \overline{r}_1 d_1 \overline{d}'_1 + r'_2 \overline{r}_2 d_2 \overline{d}'_2)^* d'_3 \overline{d}_3,$$

where Proponent moves are marked with an over-line. Note that the behaviour of the Proponent is simply to copy any

move of the Opponent to the other side of the $\vdash$. This *copy-cat* strategy works in fact as an identity for composition of strategies.

Since equivalence of regular language is decidable and strategies for terms in the language can be represented using regular languages, it follows immediately that observational equivalence in this language is decidable. Also, it is easy to give simple proofs to several test equivalences that required sophisticated mathematical techniques in conventional denotational [73] or operational [77] models.

This result was adapted to call-by-value [33] then generalised to the third-order fragment [74]. The concrete, algorithmic representation of games opened the door for examining the computational properties of observational equivalence in a variety of languages and language fragments. In a series of research papers Murawski, Ong and Walukiewicz constructed a detailed taxonomy of algorithmic properties for languages with functions and local state and an order-n recursion combinator (Tab. 1).

A concrete representation of the game model can be also used to analyse programs in a semantic-directed way. The pioneering work in this area is that of Hankin and Malacaria [63, 64, 65], who used approximate representations of game models as a basis for control-flow analysis. However, this work did not lead to any implementations.

The first automated verification tool based on game semantics was introduced by Abramsky, Ghica, Murawski and Ong [3], and it was based on the regular-language representation of second-order Algol [40]. The semantics-directed approach to program modelling and verification was asserted to have several qualitative advantages over conventional operational approaches to program verification via model-checking (c.f. [24]).

The first advantage was the possibility to model *open terms*, that is program fragments with missing components. We will refer to this feature as *external compositionality*.

Consider for example, the program below:

```
bool v := false;
let set() be { v := true }
let get() be { !v }
f(set, get)
```

Not knowing the definition of function `f()` it is not possible to model it using conventional (operational) methods. However, using a game-semantic approach we know all the possible ways in which function `f()` may call its arguments — these are the rules of the game, as discussed in the previous sections. Assuming that the only imperative feature is local state, and that the language is sequential, the game model can be represented by a regular language which is recognised by the finite state machine in Fig. 2.

Tracing the various execution paths through the automaton we can convince ourselves that the program works as

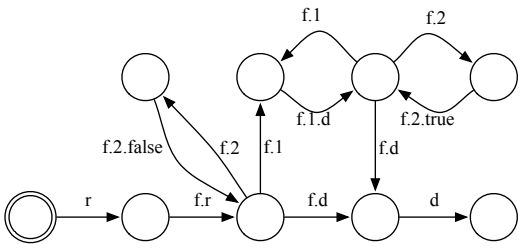

Figure 2. FSM for the "switch" ADT

a simple "switch" abstract data type (ADT), where the *get* method first produces "false," then after *set* is called it produces "true". Such properties can be encoded into temporal logic and verified for the model and therefore for the program using the established techniques of *model checking* [23].

So by being able to verify open terms we can verify ADTs or any collection of functions in libraries, in the most general legal environment they can function in. Just by using undefined functions the game model will generate what is equivalent to correct and complete testing drivers and stubs for the verification of any arbitrary collection of functions. We consider this to be a significant advantage.

The second advantage is what we call *internal compositionality*. The model of a term is created from the models of its sub-terms by a composition operation. The main consequences of this will be seen in the next section, but one immediate advantage is that the internal state of programs is hidden. For example, in Fig. 2 the value of boolean cell `v` is hidden. Only modelling the interactions between a term and its context can lead to substantial savings in larger programs that use many local variables. The following table shows how for a sorting program the number of states in the game model can be much smaller than the size of a model obtained by state exploration [3]:

Size of array	Game model	State exploration
5	163	6×10^4
10	948	3×10^6
15	2,858	9×10^{10}
20	1,153,240	5×10^{13}

3. Finite-state model checking

One main line of research in games-based software model checking is the identification of interesting programming language fragments for which the game-semantic representation of strategies is finite-state, and using model-checking techniques to verify certain properties.

A series of papers which illustrates this methodology uses as a starting point the game semantic model for a

order	pure	$+\mathbf{while}$	$+\mathbf{Y}_0$	$+\mathbf{Y}_1$
1	coNP [69]	PSPACE [69]	decidable [70]	$= 1 + \mathbf{Y}_0$
2	PSPACE [69]	PSPACE [69]	decidable [70]	undecidable [75]
3	EXPTIME [71]	EXPTIME [71]	decidable [70]	undecidable [75]
4	undecidable [68]	undecidable [68]	undecidable [68]	undecidable [75, 68]

Table 1. Taxonomy of complexity results for fragments of IA

concurrent, higher-order and imperative programming language [41]. Even at the first order fragment the concrete representation of strategies for this language is not finitary, e.g. the language in (5), and in general it is undecidable.

One way to identify a reasonable fragment of this language, having finite-state models, is to use a type system which imposes static (finite) bounds on the number of concurrent threads that functions are allowed to create [44]. This still allows a large number of interesting concurrent algorithms to be coded up and automatically verified [43]. As before, programs that are parametrised by undefined procedures can be modelled and verified, and the hiding of internal state manipulation can lead to significant savings in terms of overall state-space. For example, for a producer-consumer program the state-exploration model with 3×10^9 states can be represented by a strategy with 24,489 states. Another tool that follows a similar methodology is APEX, used for the automated verification of probabilistic programs via finite-state representation of the probabilistic game model [60].

As is the case with conventional model checking, the size of the model grows very fast with the size of the program — the so-called "state-explosion" problem. We believe that the best methodology for avoiding this problem is by exploiting the external compositionality of the game-semantic module, which allows the verification of a larger program by breaking it up into smaller sub-programs to be verified independently. However, it is important and useful to adapt conventional state-reduction heuristics to the game-semantic setting in order to allow the verification of larger programs. For this, we can find inspiration in the world of conventional model checking, which is a rich source of state-reduction heuristics.

One of the established methods for state-reduction is that of *abstract interpretation* [25]. The basic idea of this heuristic in the finite-state setting is that of quotienting the state-space by an equivalence relation, and reducing it therefore to an approximate (or "abstract") transition system. Transitions between abstract states are as inherited from the concrete system. Obviously, this method allows the arbitrary scaling down of a system, and it also makes possible the finitary approximation of an infinite state system. It can be easily seen that such an approximation is *sound* in the sense that if the precise system has illegal transitions, they are inherited by its approximation. However, it is possible for the approximate system to have transitions, possibly illegal, that do not exist in the original. Thus, an analysis of the approximate system for errors can give "false positives", but cannot give "false negatives." The same quotienting can be applied to the set of labels in the system to reduce the number of transitions.

For game-based models, approximation was implemented in the GAMECHECKER tool [30, 31]. What makes games-based approximation interesting is the way it exploits the internal compositionality of the model. Obviously, the approximate model cannot be generated from the precise model, but from the source code. This usually requires a certain amount of syntactic manipulation of the code [16]. Using a games-based methodology it is enough to approximate strategies for the basic constructs of the language, then the model of the program is obtained through the usual method of composition of models of sub-programs. This not only simplifies the implementation substantially, but makes it possible to approximate any occurrence of a construct separately. Branches of the syntax tree that are deemed to be more important can be approximated more precisely, whereas those that are less relevant can be approximated more aggressively. In Fig. 3 we can see a finite-state approximation of the strategy $\mathsf{eq} : \mathsf{exp} \times \mathsf{exp} \to \mathsf{bool}$ that tests two integers for equality. The "abstract values" $-$ and $+$ represent the equivalence classes for negative and positive integers, respectively. Note that comparing $+$ with $+$ or $-$ with $-$ leads to the ambiguous state (marked with grey) from which both true and false can be (non-deterministically) produced as a result for the comparison.

Note that games-based approximation cannot fit in the framework of abstract interpretation, for technical reasons. Exploiting the internal compositionality of the model to change the level of approximation at various points in the program requires the approximations to compose, whereas it is well known that abstract interpretation cannot normally compose in this manner [1].

It is often impossible to determine what is the best approximation method for a given program. A popular heuristic is to start with a very broad approximation then gradually refine it, using the false counterexamples as sources of information regarding possible errors [22]. GAMECHECKER

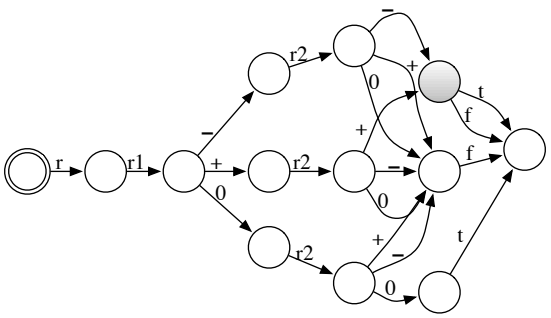

Figure 3. Strategy for equality, approximated

implements refinement using the following heuristic: when finding an error trace, check whether it has any sources of non-determinism introduced by the approximation, such as the grey state in Fig. 3. If no such states exist, then the trace can be "certified" as an authentic error. If not, it can be mapped back to the syntax and be used to identify subterms that need to be better approximated.

A better implementation of the ideas in GAMECHECKER is MAGE, which additionally uses the *lazy* and *symbolic* heuristics [14]. Laziness works by avoiding the construction of whole models prior to verification of certain properties. Instead, they construct the model step-by-step as it is being verified, which can save a lot of work if the program has errors that can be found before the entire model is constructed. If the program has no errors then laziness does not achieve any savings. Symbolic model checking means avoiding explicit representation of the finite state machine of a model when the transition function can be symbolically represented. Thus, instead of looking for the "next state" in a table of transitions, it can be recalculated whenever it is needed. This reduces space requirements and increases execution time, which is usually desirable in model checking.

These two heuristics work very well in conjunction with approximation refinement, because approximation can introduce a vast number of false errors, which the tool can zoom in on and resolve very fast. A comparison of MAGE with GAMECHECKER confirms this clearly, using the particular example of searching for buffer overflows in a certain program. The largest array size GAMECHECKER could handle was 32, in which overflow (underflow respectively) were detected in over three hours (over eight minutes, respectively). MAGE detected the same errors in 1.15 and 0.03 seconds, respectively, and could verify arrays of size up to 1024 in under 15 minutes for overflow and 0.03 seconds for underflow. In fact the games-based tool compared very well against the much more sophisticated BLAST tool [48] on these examples. For an array of size 224, the largest BLAST could handle, overflow was found in 506 seconds compared with 19 seconds for MAGE; BLAST failed for a 225 elements array, which was verified by MAGE in 19.3 seconds.

Approximation-refinement, especially performed lazily, is a remarkably effective way of finding bugs in programs. However, if a program has no bugs, certifying its correctness is a more challenging problem. Simplistic ways of quotienting the state space are likely to continually give rise to false counterexamples, so that certification is impossible and approximation needs to be refined up to a point where it exceeds available resources. In order to certify a program the quotienting of the state space needs to use more advanced heuristics. The most popular one is that of *predicate abstraction* [16], and it works by picking up predicates from the source code, for example from assertions and as guards to `if` and `while` statements, and represent the state space as a combination of these predicates.

The latest version of MAGE implements predicate abstraction for game semantics [15]. Unlike the previous heuristics, predicate abstraction does not fit naturally into the game-semantic paradigm. It relies in an essential way on global state, a concept which in game semantics is played down. The ability to exploit the internal compositionality of the game model is also hindered by the global nature of predicate abstraction. Although predicate abstraction for game semantics can be formulated and implemented efficiently using a game-like model in which plays are annotated with global state (c.f. [61]) we must wonder whether we could find in the future a "smart" abstraction for games, perhaps akin to predicate abstraction, but more germane to the internally compositional way game models are constructed.

Most of the work described in this section concentrated on the development of state-reduction heuristics for game models in a way that exploits internal compositionality. Much less work was dedicated to exploiting the external compositionality of the model, although perhaps in the longer term this direction of activity could prove to be more beneficial, allowing the compositional verification of large systems by breaking them up in smaller sub-systems. In order for such an approach to work, it is important that the individually verified sub-systems can be assembled into the larger system without introducing errors. This requires a suitable program logic. A games-based program logic [34] can go farther than the usual assertion-based, Hoare-like program logics, because the game model allows the formulation and validation of temporal properties about the way procedures use their arguments. This line of research remains in its infancy.

Finally, it is possible to combine the internal and the external compositionality of the model to create brand new ways of attacking the difficult problem of program verification. The new technique of *clipping* [38] is a syntactic

approximation that works by replacing entire branches in the syntax tree by free identifiers of the right type, called *stumps*. The semantic motivation for this is that the game model of a free identifier is an approximation of the game model for any (closed) term of that type, therefore replacing one with the other is a conservative approximation. The approximation can be iteratively refined by replacing stumps with "clipped" version of the branches of the syntax tree which they replaced. Since arbitrarily large sub-terms can be replaced with free identifiers it is expected that clipping can achieve great reductions of the verification effort.

4. Higher-order model checking

The techniques in the previous section explain how game semantics introduces a new perspective on extracting a model from a term, a model which is internally and externally compositional. Once the model is produced, the techniques for verifying its properties are fairly standard—adaptations of conventional model-checking heuristics such as approximation, refinement or laziness, adapted in such a way that they can exploit the compositionality of the game model. What is genuinely new is the ability to model open terms observationally; the algorithmics however are fairly standard.

A more adventurous development is the work of Ong and collaborators who used game semantics not only to extract a model from a piece of code but also to reason about such models in cases when they are computationally more complex than finite state machines. This work consists of a series of profound results which redraw the known boundary between the expressible and the decidable by showing that the model-checking problem for expressive logics, such as monadic second order logic (MSO) or the modal mu-calculus, formulated on infinite structures generated by higher-order recursion schemata or higher-order push-down automata is actually decidable.

The question of MSO decidability for higher-order push-down automata was first formulated by Knapik, Niwiński and Urzyczyn [54, 55]. Their first results hinged on a seemingly arbitrary syntactic condition called *safety*, as did subsequent results by Caucal [21]. Ong et. al. showed that for certain classes of structures the safety condition was spurious, as unsafe structures have safe formulations [13], and also that up to a certain order in the hierarchy the safety constraint has no impact on decidability [12]. Finally, Ong answered the general question of the decidability of modal mu-calculus on arbitrary n-order recursion schemata in the affirmative, showing also that it is n-EXPTIME complete [76].

The proof of the main result uses ideas which are inspired from game semantics, such as that of *traversal*, which is a particular representation of innocent strategies, as used in the model of the language PCF. Further interesting consequences of this line of research came from the semantic, rather than syntactic, analysis of the concept of safety. It turns out that this restriction is not arbitrary but it can be captured into an elegant type system, the safe lambda calculus [19]. On the syntactic level, this calculus has the interesting property that reduction does not necessitate renaming of bound identifiers, making it into an "alpha-conversion free calculus." Semantically, this leads to a simplification of the structure for representing plays, eliminating the need for justification pointers.

This line of work has been up to this point mostly theoretical, but a first model checking tool for higher-order programs has been recently developed, HOMER [49], hopefully opening the door to a new way of automatically verifying complex properties of computer programs.

5. Hardware synthesis

The problem of synthesising digital circuits from behavioural specifications written in higher level programming languages (*hardware compilation*) has been studied for a long time, yet a definitive solution has not been forthcoming. We believe that one of the main reasons for this failure is the lack of proper mechanisms for abstraction which work well in the context of digital circuits [36]. This problem is interesting in itself, but we also believe that the constant increase in sophistication of digital devices such as field-programmable gate arrays (FPGA) makes the practical importance of hardware compilation more important. In working with FPGAs the ability to create complex designs fast is paramount, and working with higher level languages is in general faster and easier than working with lower level hardware description languages.

Game semantics offers a new and intriguing approach to solving this problem. It has been known for a while that structuring computation around concepts such as *channel*, *event* and *communication* recommends process calculi as good intermediate abstractions for hardware. Refining processes into hardware-level representations has been extensively studied (e.g. [83]). Game Semantics is a process-like model which encapsulates the right mathematical structure needed to interpret programming languages. Like process calculi, it is event-oriented and can be refined into hardware. Hardware synthesis from game semantics connects these two separate ideas.

There is an immediate relation between the following game-semantic and hardware concepts, already implicit in the Geometry of Interaction foundations of game semantics [2].

- An *arena* corresponds to a circuit's *interface*.
- A *move* corresponds to a *port* in the interface.

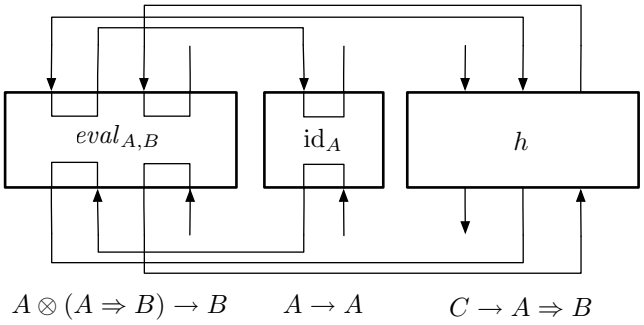

Figure 4. Universal property for $eval_{A,B}$

- A *Proponent* move is an *output* port.
- An *Opponent* move is an *input* port.
- A *question* is a *request* port.
- An *answer* is an *acknowledgment* port.
- An *occurrence* of a move is a *signal* on a port.
- A *play* is a *waveform* on an interface.
- The rules of the game are a *protocol* of access to the interface.

From a hardware point of view it is fair to think of game semantics as a type-directed way of deriving circuit interfaces, and a compositional access protocol for these interfaces. The existence of a fixed access protocol is important in practice, because it allows the design of circuits in a way that exploits behavioural constraints on the environment in which they operate, which leads to more efficient implementations. The compositionality of the protocol is important for the correctness of a hardware compiler, as it ensures that a composite circuit formed from protocol-compliant units itself complies with the protocol.

The first concrete step in this direction was this author's *Geometry of Synthesis* [35]. It is a directly expressed circuit semantics for a procedural programming language based on the *Syntactic Control of Interference* type system [80]. As the name indicates, it is a semantics based on the Geometry of Interaction construction, which is used to interpret the structural fragment of the type system (abstraction, application, identity) using circuits in a purely black-box manner. The connections between the Geometry of Interaction [2] monoidal categories [53] and structural properties of circuits have been known for a long time, but it was perhaps not appreciated enough that they have an important practical application in hardware synthesis. The fact that the axioms of a monoidal closed category can be expressed merely as properties of "boxes and wires" means that the operations of abstraction and application can be implemented very easily,

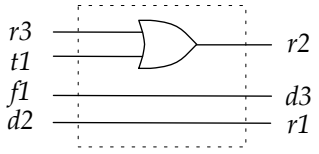

Figure 5. Circuit implementing iteration

with no overhead, by just keeping the right accounting of ports and connectors.

An interface of type $A \otimes B$ has the ports of type A and B, disjointly unioned; an interface of type $A \Rightarrow B$ is as that for tensor product, but with ports in A taken with reversed input-output polarity. Morphisms $A \to B$ are circuits with interface $A \Rightarrow B$. Composition of circuits is precisely the physical realisation of composition of strategies, as indicated in Fig. 1. The identity is the circuit of shape $A \to A$ which contains only wires connecting corresponding ports in the two occurrences of A. The evaluation morphism in the category, which models function application, is simply a set of wires that connects the appropriate ports in the function to those of its argument. The universal property that the evaluation morphism must satisfy is

$$\forall f{:}A \otimes C \to B.\ \exists h{:}C \to A \Rightarrow B.\ f = eval_{A,B} \circ (\mathrm{id}_A \otimes h).$$

This axiom is represented by the circuit in Fig. 4, and its "proof" is produced simply by untangling the wires. This means that one of the main problems with hardware compilation can be solved elegantly by following the Geometry of Interaction route, which gives a natural (zero-overhead!) way to implement functions and procedures in hardware.

The problem that remains to be solved is finding appropriate interpretations for language constructs, and this is where game semantics can help. The event-based nature of game semantics gives a strong hint as to what the implementations of the constructs in hardware should be. For example, the strategy for iteration described by the regular language in Eqn. 4 suggests the implementation in Fig. 5. In this circuit, if the environment provides inputs consistent with a legal behaviour of Opponent then the circuit will provide outputs which are consistent with the Proponent moves in the strategy.

Although the basic intuitions are straightforward, the practical implementation of a compiler based on these ideas raises some interesting technical issues. Game semantics has a fundamentally asynchronous presentation, in which it is not possible for two moves to occur at the same time. In game semantics of concurrency [41] an interleaved presentation of concurrent plays is used; even presentations that avoid interleaving are still meant to reflect an asynchronous view of concurrency in the game [67]. This means that game semantics is a formalism naturally suited for an asynchronous model of circuit design. Producing synchronous

circuits from game semantics is a more complex process which must somehow account for simultaneity and instantaneity in the semantics. These issues are currently an active area of research [37].

Acknowledgement. The author is supported by a UK EPSRC Advanced Research Fellowship. Andrzej Murawski, Paul Levy and Russell Harmer helped with suggestions.

References

[1] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *J. Log. Comput.*, 1(1):5–40, 1990.

[2] S. Abramsky. Retracing some paths in process algebra. In *CONCUR*, pages 1–17, 1996.

[3] S. Abramsky, D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Applying game semantics to compositional software modeling and verification. In *TACAS*, pages 421–435, 2004.

[4] S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *LICS*, pages 150–159. IEEE Computer Society, 2004.

[5] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *LICS*, pages 334–344, 1998.

[6] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic (extended abstract). In *FSTTCS*, pages 291–301, 1992.

[7] S. Abramsky and R. Jagadeesan. New foundations for the Geometry of Interaction. In *LICS*, pages 211–222. IEEE Computer Society, 1992.

[8] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[9] S. Abramsky and G. McCusker. Games for recursive types. In C. Hankin, I. Mackie, and R. Nagarajan, editors, *Theory and Formal Methods*, pages 1–20. Imperial College Press, 1994.

[10] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for Idealized Algol with active expressions. *Electr. Notes Theor. Comput. Sci.*, 3, 1996.

[11] S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, pages 1–17, 1997.

[12] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. The monadic second order theory of trees given by arbitrary level-two recursion schemes is decidable. In *TLCA*, pages 39–54, 2005.

[13] K. Aehlig, J. G. de Miranda, and C.-H. L. Ong. Safety is not a restriction at level 2 for string languages. In *FoSSaCS*, pages 490–504, 2005.

[14] A. Bakewell and D. R. Ghica. On-the-fly techniques for game-based software model checking. In *TACAS*, pages 78–92, 2008.

[15] A. Bakewell and D. R. Ghica. Compositional predicate abstraction from game semantics. In *TACAS*, pages 62–76, 2009.

[16] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[17] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.

[18] A. Blass. A game semantics for linear logic. *Ann. Pure Appl. Logic*, 56(1-3):183–220, 1992.

[19] W. Blum and C.-H. L. Ong. The safe lambda calculus. In *TLCA*, pages 39–53, 2007.

[20] S. D. Brookes. Historical introduction to "concrete domains" by G. Kahn and Gordon D. Plotkin. *Theor. Comput. Sci.*, 121(1&2):179–186, 1993.

[21] D. Caucal. On infinite terms having a decidable monadic theory. In *MFCS*, pages 165–176, 2002.

[22] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[23] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[24] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *ICSE*, pages 439–448, 2000.

[25] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[26] P.-L. Curien. Symmetry and interactivity in programming. *Bulletin of Symbolic Logic*, 9(2):169–180, 2003.

[27] V. Danos and R. Harmer. Probabilistic game semantics. In *LICS*, pages 204–213, 2000.

[28] V. Danos and R. Harmer. The anatomy of innocence. In *CSL*, pages 188–202, 2001.

[29] V. Danos, H. Herbelin, and L. Regnier. Game semantics & abstract machines. In *LICS*, pages 394–405, 1996.

[30] A. Dimovski, D. R. Ghica, and R. Lazic. Data-abstraction refinement: A game semantic approach. In *SAS*, pages 102–117, 2005.

[31] A. Dimovski, D. R. Ghica, and R. Lazic. A counterexample-guided refinement tool for open procedural programs. In *SPIN*, pages 288–292, 2006.

[32] W. Felscher. Dialogues as a foundation of intuitionistic logic. In *Handbook of Philosophical Logic*, volume 3. 1986.

[33] D. R. Ghica. Regular-language semantics for a call-by-value programming language. *Electr. Notes Theor. Comput. Sci.*, 45, 2001.

[34] D. R. Ghica. *A Games-Based Foundation for Compositional Software Model Checking*. PhD thesis, Queen's University, Kingston, Ontario, Canada, 2002.

[35] D. R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375, 2007.

[36] D. R. Ghica. Function interface models for hardware compilation. Technical Report CSR-04-08, University of Birmingham, 2008. (submitted for publication).

[37] D. R. Ghica. Geometry of Synthesis 2: Compiling finite-bound concurrency into hardware. Workshop on Games for Logic and Programming Languages IV, 2009.

[38] D. R. Ghica and A. Bakewell. Clipping: A semantics-directed syntactic approximation. In *LICS*, 2009. (forthcoming).

[39] D. R. Ghica and G. McCusker. Reasoning about Idealized Algol using regular languages. In *ICALP*, pages 103–115, 2000.

[40] D. R. Ghica and G. McCusker. The regular-language semantics of second-order Idealized Algol. *Theor. Comput. Sci.*, 309(1-3):469–502, 2003.

[41] D. R. Ghica and A. Murawski. Angelic semantics of fine-grained concurrency. *Annals of Pure and Applied Logic*, 151(2-3):89–114, 2008.

[42] D. R. Ghica and A. S. Murawski. Angelic semantics of fine-grained concurrency. In *FoSSaCS*, pages 211–225, 2004.

[43] D. R. Ghica and A. S. Murawski. Compositional model extraction for higher-order concurrent programs. In *TACAS*, pages 303–317, 2006.

[44] D. R. Ghica, A. S. Murawski, and C.-H. L. Ong. Syntactic control of concurrency. *Theor. Comput. Sci.*, 350(2-3):234–251, 2006.

[45] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[46] R. Harmer, M. Hyland, and P.-A. Melliès. Categorical combinatorics for innocent strategies. In *LICS*, pages 379–388. IEEE Computer Society, 2007.

[47] R. Harmer and G. McCusker. A fully abstract game semantics for finite nondeterminism. In *LICS*, pages 422–430, 1999.

[48] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[49] D. Hopkins and C.-H. L. Ong. HOMER: a Higher-order Observational equivalence Model checkER. In *CAV*, 2009. (forthcoming).

[50] D. J. D. Hughes. Games and definability for System F. In *LICS*, pages 76–86, 1997.

[51] J. M. E. Hyland and C.-H. L. Ong. Pi-calculus, dialogue games and PCF. In *FPCA*, pages 96–107, 1995.

[52] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[53] A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Mathematical Proceedings of Cambridge Philosophical Society*, 119:447–468, 1996.

[54] T. Knapik, D. Niwinski, and P. Urzyczyn. Deciding monadic theories of hyperalgebraic trees. In *TLCA*, pages 253–267, 2001.

[55] T. Knapik, D. Niwinski, and P. Urzyczyn. Higher-order pushdown trees are easy. In *FoSSaCS*, pages 205–222, 2002.

[56] J. Laird. Full abstraction for functional languages with control. In *LICS*, pages 58–67, 1997.

[57] J. Laird. A fully abstract game semantics of local exceptions. In *LICS*, pages 105–114, 2001.

[58] J. Laird. A game semantics of Idealized CSP. *Electr. Notes Theor. Comput. Sci.*, 45, 2001.

[59] J. Laird. A game semantics of names and pointers. *Ann. Pure Appl. Logic*, 151(2-3):151–169, 2008.

[60] A. Legay, A. S. Murawski, J. Ouaknine, and J. Worrell. On automated verification of probabilistic programs. In *TACAS*, pages 173–187, 2008.

[61] P. B. Levy. Global state considered helpful. *Electr. Notes Theor. Comput. Sci.*, 218:241–259, 2008.

[62] P. Lorenzen. Ein dialogisches konstruktivitaetskriterium. In *Innitistic Methods*, PWN, Proc. Symp. Foundations of Mathematics, pages 193–200, Warsaw, 1961.

[63] P. Malacaria and C. Hankin. Generalised flowcharts and games. In *ICALP*, pages 363–374, 1998.

[64] P. Malacaria and C. Hankin. A new approach to control flow analysis. In *CC*, pages 95–108, 1998.

[65] P. Malacaria and C. Hankin. Non-deterministic games and program analysis: An application to security. In *LICS*, pages 443–452, 1999.

[66] P.-A. Melliès. Asynchronous games 2: The true concurrency of innocence. In *CONCUR*, pages 448–465, 2004.

[67] P.-A. Melliès and S. Mimram. Asynchronous games: Innocence without alternation. In *CONCUR*, pages 395–411, 2007.

[68] A. S. Murawski. About the undecidability of program equivalence in finitary languages with state. *ACM Transactions on Computational Logic*, 6(4):701–726, 2005.

[69] A. S. Murawski. Games for complexity of second-order call-by-name programs. *Theoretical Computer Science*, 343(1/2):207–236, 2005.

[70] A. S. Murawski, C.-H. L. Ong, and I. Walukiewicz. Idealized Algol with ground recursion and DPDA equivalence. In *ICALP*, pages 917–929, 2005.

[71] A. S. Murawski and I. Walukiewicz. Third-order Idealized Algol with iteration is decidable. *Theoretical Computer Science*, 390(2-3):214–229, 2008.

[72] H. Nickau. Hereditarily sequential functionals. In *LFCS*, pages 253–264, 1994.

[73] P. W. O'Hearn and R. D. Tennent. Parametricity and local variables. *J. ACM*, 42(3):658–709, 1995.

[74] C.-H. L. Ong. Observational equivalence of 3rd-order Idealized Algol is decidable. In *LICS*, pages 245–256, 2002.

[75] C.-H. L. Ong. An approach to deciding observational equivalence of Algol-like languages. *Annals of Pure and Applied Logic*, 130:125–171, 2004.

[76] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *LICS*, pages 81–90. IEEE Computer Society, 2006.

[77] A. M. Pitts. Reasoning about local variables with operationally-based logical relations. In *LICS*, pages 152–163, 1996.

[78] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003.

[79] G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.

[80] J. C. Reynolds. Syntactic control of inference, part 2. In *ICALP*, pages 704–722, 1989.

[81] D. Scott and C. Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Oxford Programming Research Group, 1971.

[82] N. Tzevelekos. Full abstraction for nominal general references. In *LICS*, pages 399–410, 2007.

[83] K. van Berkel. *Handshake circuits: An intermediary between communicating processes and VLSI*. PhD thesis, Technische Univ., Eindhoven (Netherlands), 1992.