Micro-Machinations
A DSL for Game Economies

Paul Klint[1] and Riemer van Rozen[2]

[1] Centrum Wiskunde & Informatica**
[2] Amsterdam University of Applied Sciences**

Abstract. In the multi-billion dollar game industry, time to market limits the time developers have for improving games. Game designers and software engineers usually live on opposite sides of the fence, and both lose time when adjustments best understood by designers are implemented by engineers. Designers lack a common vocabulary for expressing gameplay, which hampers specification, communication and agreement. We aim to speed up the game development process by improving designer productivity and design quality. The language Machinations has introduced a graphical notation for expressing the rules of game economies that is close to a designer's vocabulary. We present the language *Micro-Machinations* (MM) that details and formalizes the meaning of a significant subset of Machination's language features and adds several new features most notably modularization. Next we describe *MM Analysis in R*ASCAL (MM AiR), a framework for analysis and simulation of MM models using the Rascal meta-programming language and the SPIN model checker. Our approach shows that it is feasible to rapidly simulate game economies in early development stages and to separate concerns. Today's meta-programming technology is a crucial enabler to achieve this.

1 Introduction

There is anecdotal evidence that versions of games like Diablo III[1] and Dungeon Hunter 4[2] contained bugs in their game economy that allowed players to illicitly obtain game resources that could be purchased for real money. Such errors seriously threaten the business model of game manufacturers. In the multi-billion dollar game industry, time to market limits the time designers and developers have for creating, implementing and improving games. In game development speed is everything. This applies not only to designers who have to quickly assess player experience and to developers that are under enormous pressure to deliver software on time, but also to the performance of the software itself. Common software engineering wisdom does not always apply when pushing technology to the limits regarding performance and scalability. Domain-Specific Languages (DSLs) have been successfully applied in domains ranging from planning and

** This work is part of the EQuA project. `http://www.equaproject.nl/`

[1] `http://us.battle.net/d3/en/forum/topic/8796520380`

[2] `http://www.data-apk.com/2013/04/dungeon-hunter-4-v1-0-1.html`

M. Erwig, R.F. Paige, and E. Van Wyk (Eds.): SLE 2013, LNCS 8225, pp. 36–55, 2013.
© Springer International Publishing Switzerland 2013

financial engineering to digital forensics resulting in substantial improvements in quality and productivity, but their benefits for the game domain are not yet well-understood.

There are various explanations for this. The game domain is diffuse, encompassing disparate genres, varying objectives and concerns, that often require specific solutions and approaches. Because the supporting technologies are constantly changing, domain analysis tracks a moving target, and opportunities for domain modeling and software reuse are limited [1]. Existing academic language-oriented approaches, although usually well-scoped, are often poorly adaptable, one-off, top-down projects that lack practical engineering relevance. Systematic bottom-up development and reuse have yielded libraries called game engines but such (commercial) engines are no silver bullet either, since they only provide general purpose solutions to technical problems and need significant extension and customization to obtain the functionality for a completely new game. Engines represent a substantial investment, and also create a long-term dependency on the vendor for APIs and support.

Our objective is to demonstrate that game development can benefit from DSLs despite the challenges posed by the game domain and the perceived shortcomings of existing DSL attempts. We envision light-weight, reusable, inter-operable, extensible DSLs and libraries for well-scoped game concerns such as story-lines, character behavior, in-game entities, and locations. We focus in this paper on the challenge of speeding up the game development process by improving designer productivity and design quality. Our main contributions:

- Micro-Machinations (MM), a DSL for expressing game economies.
- Micro-Machinations Analysis in Rascal (MM AiR), an interactive simulation, visualization and validation workbench for MM.
- The insight that combining state-of-the-art tools for meta-programming (Rascal[3] [2]) and model checking (Promela/Spin[4] [3]) enable rapid prototyping of and experimentation in the game domain with frameworks like MM AiR.

2 Micro-Machinations

2.1 Background

Our main source of inspiration is the language Machinations [4] that has been based on an extensive analysis of game design practices in industry and provides a graphical notation for designers to express the rules of game economies. A *game economy* is an abstract game system governed by rules (e.g., how many coins do I need to buy a crystal) that offers players a playful interactive means to spend and exchange atomic game resources (e.g., crystals, energy). Resources are characterized by *amount* and *unit kind.*

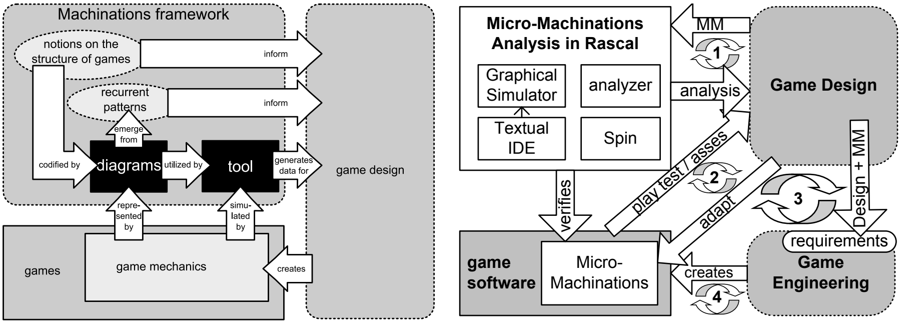
[3] `http://www.rascal-mpl.org/`

[4] `http://spinroot.com/spin/whatispin.html`

Its focus is on the simulation of game designs. Various game design patterns have been identified in this context [4,5] as well. Machinations takes an approach that closely resembles Petri Nets that have been used in the game domain by others. For instance, Brom and Abonyi [6] use Petri nets for narratives, and Araújo [7] proposes general game design using Petri Nets. Other approaches to formalisms for game development related to design include hierarchical state machines [8], behavior trees [9], and rule-based systems [10].

Machinations is a visual game design language intended to create, document, simulate and test the internal economy of a game. The core *game mechanics* are the rules that are at the heart of a game. Machinations diagrams allow designers to write, store and communicate game mechanics in a uniform way. Perhaps the hardest part of game design is adjusting the game balance and make a game challenging and fun.

Figure 1a shows the Machinations framework as presented in [4]. Machinations can be seen as a design aid, that augments paper prototyping, which is used by designers to understand game rules and their effect on play. The Machinations tool[5] can be used to generate automatic random runs, that represent possible game developments, as feedback on the design process. Machinations is already in use by several game designers in the field.



(a) Machinations Conceptual Framework (b) Micro-Machinations Architecture

Fig. 1. Side-by-side comparison of Machinations (a) and Micro-Machinations (b)

Micro-Machinations (MM) is an evolutionary continuation of Machinations aiming at software prototyping and validation. MM is a formalized extended subset of Machinations, that brings a level of precision (and reduction of non-determinism) to the elements of the design notation that enables not only simulation but also formal analysis of game designs. MM also adds new features, most notably modularization. MM is intended as embedded scripting language for game engines that enables interaction between the economic rules and the so-called *in-game entities* that are characterized by one or more atomic resources.

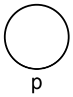
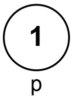
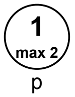
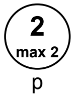
[5] http://www.jorisdormans.nl/machinations/

An advantage of early paper prototyping is that loosely defined rules can be changed quickly and analyzed informally. Later, during software prototyping the rules have to be described precisely and making non-trivial changes usually takes longer. To start software prototyping as early as possible, a quick change in a model should immediately change the software that implements it. Therefore we study the precise meaning of the language elements and how they affect the game state. By leveraging meta-programming, language work-benches and model checking we can provide additional forms of analysis and prototyping. This enables us to answer questions about models designers might have, that affect both the design and the software that implements it. Figure 1b shows schematically how MM relates to game development.

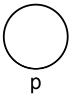
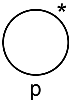
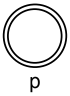
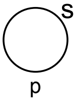
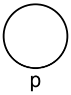
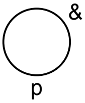
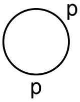
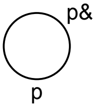
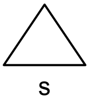
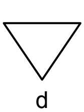
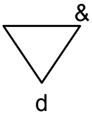
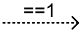
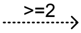
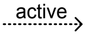
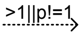
Our objectives are to introduce short and separate design iterations (1 and 2) to free time for separate software engineering iterations (4) and alleviate relying on the usually longer interdependent development iterations (3).

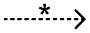
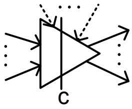
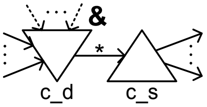
2.2 Micro-Machinations Condensed

MM models are graphs that consist of two kinds of elements, *nodes* and *edges*. Both may be annotated with extra textual or visual information. These elements describe the rules of internal game economies, and define how resources are step-by-step propagated and redistributed through the graph. Here is a cheat sheet for the most important language elements[6].

 `pool p` **Empty pool**	 `pool p at 1` **Pool & resource**	A *pool* is a named node, that abstracts from an in-game entity, and can contain *resources*, such as coins, crystals, health, etc. Visually, a pool is a circle with an integer in it representing the current amount of resources, and the initial amount at which a pool starts when first modeled. Pools may specify a maximum capacity for resources, which can never be exceeded, that is visually a prefix max followed by an integer.
 `pool p at 1 max 2` **Limited capacity**	 `pool p at 2 max 2` **Full pool**	
$\longrightarrow$ `->` **Resource connection flow rate of one**	$\xrightarrow{\text{all}}$ `-all->` **Resource connection unlimited rate**	A *resource connection* is an edge with an associated expression that defines the rate at which resources can flow between source and target nodes. During each transition or *step*, nodes can *act* once by redistributing resources along the resource connections of the model. The *inputs* of a node are resource connections whose arrowhead points to that node, and its *outputs* are those pointing away.
$\xrightarrow{/2}$ `-/2->` **Resource connection half flow rate**	$\xrightarrow{4*p+1}$ `-4*p+1->` **Resource connection flow expression**	

[6] For conciseness we only give an informal description here, closely adhering to [4].

 pool p **Passive pool**	 auto pool p **Automatic pool**	The *activation modifier* determines if a node can act. By default, nodes are *passive* (no symbol) and do not act unless activated by another node. *Interactive* (double line) nodes signify user actions that during a step can activate a node to act in the next state. *Automatic* (*) nodes act automatically, once every step. *Start* (s) nodes are active in the initial state, but become passive afterwards.
 user pool p **Interactive pool**	 start pool p **Start pool**	
 pool p **Pool with pull act and any modifier**	 all pool p **Pool with pull act and all modifier**	Nodes act either by pulling (default, no symbol) resources along their inputs or pushing (p) resources along their outputs. Nodes that have the *any modifier* (default, no symbol), interpret the flow rate expressions of their resource connections as upper bounds, and move as many resources as possible. Additionally, these nodes may process their resource connections independently and in any order. Nodes that instead have the *all modifier* (&) interpret them as strict requirements, and the associated flows all happen or none do.
 push pool p **Pool with push act and any modifier**	 push all pool p **Pool with push act and all modifier**	
 source s **Source**		A *source* node, appearing as a triangle pointing up, is the only element that can generate resources. A source can be thought of as a pool with an infinite amount of resources, and therefore always pushes all resources or all resources are pulled from it. The any modifier does not apply, and resources may never flow into a source. Also, infinite amounts may not flow from sources.
 drain d **Drain with any modifier**	 all drain d **Drain with all modifier**	A *drain* node, appearing as a triangle pointing down, is the only element that can delete resources. Drains can be thought of as pools with an infinite negative amount of resources, and have capacity to pull whatever resources are available, or whatever resources are pushed into them. No resources can ever flow from a drain.
 .==1.> **Condition edge equals one expr**	 .>=2.> **Condition edge greater equals expr**	A node can only be active if all of its *conditions* are true. A *condition* is an edge appearing as a dashed arrow with an associated Boolean expression. Its source node is a pool that forms an implicit argument in the expression, and the condition applies to the target node.
 .active.> **Condition edge active expr**	 .>1||p!=1.> **Condition edge composed expr**	

 .*.> **Trigger edge**	A *trigger* is an edge that appears as a dashed arrow with a multiply sign. The origin node of a trigger activates the target node when for each resource connection the source works on, there is a flow in the transition that is greater or equal to that of the associated flow rate expression. Additionally, automatic pulling nodes without inputs and automatic pushing nodes without outputs always activate targets of their triggers.
 converter c from A to B **converter**	Converters are nodes, appearing as a triangle pointing right with a vertical line through the middle, that consume one kind of resources and produce another. Converters are not core elements because they can be rewritten as a combination of a drain, a trigger and a source. Unlike basic node types, converters therefore take two steps to complete. Converters can only pull, and the any modifier does not apply. If specified, the unit kinds on the inputs and outputs must match the converter's unit kinds.
 all drain c_d of A source c_s of B c_d .*.> c_s **desugared converter**	

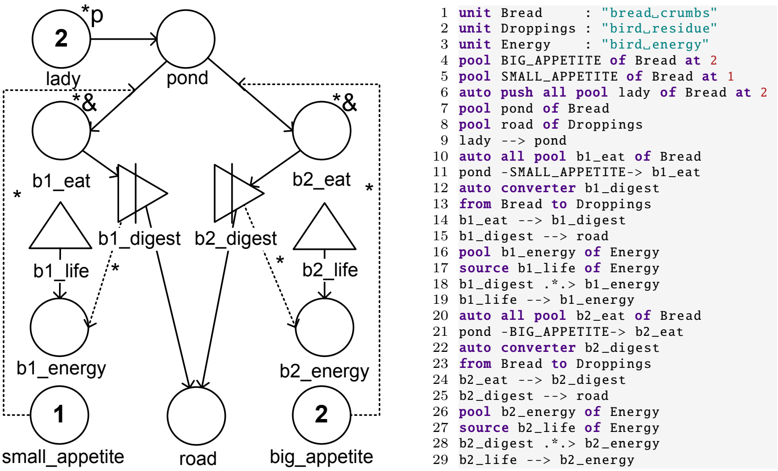
2.3 Introductory Example

Figure 2a shows an example how a designer might model a lady feeding birds in the original Machinations language. Figure 2b shows the textual equivalent as introduced in MM. The lady automatically throws bread crumbs in a pond (*p) one at a time, and two birds with different appetites compete for them. The first has a small appetite and the latter a big a appetite. Both birds automatically try to eat the whole amount (*&) their appetite compels them to. The edges from *small_ appetite* and *big_ appetite* are not triggers but *edge modifiers*, and we have replaced them by flow rate expressions in MM (lines 11 & 21). Birds digest food automatically which gives them energy and produces *droppings* on the road.

2.4 Game Designer's Questions

Given a model such as the example from Section 2.3, a designer might have the following questions.

- **Inspect:** Given a game state, what are the values of the pools, which nodes are active and what do they do?
- **Select:** Given a game state, what are the possible transitions? Are there alternatives? What are these alternatives and what are their successor states?
- **Reach:** Given this model, does a node ever act? Does a flow ever happen? Does a trigger ever happen? Where in the model can resources be scarce? Is an undesired state reachable, e.g., can the player ever have items from the



(a) Visual Model

```
unit Bread     : "bread␣crumbs"
unit Droppings : "bird␣residue"
unit Energy    : "bird␣energy"
pool BIG_APPETITE of Bread at 2
pool SMALL_APPETITE of Bread at 1
auto push all pool lady of Bread at 2
pool pond of Bread
pool road of Droppings
lady --> pond
auto all pool b1_eat of Bread
pond -SMALL_APPETITE-> b1_eat
auto converter b1_digest
from Bread to Droppings
b1_eat --> b1_digest
b1_digest --> road
pool b1_energy of Energy
source b1_life of Energy
b1_digest .*.> b1_energy
b1_life --> b1_energy
auto all pool b2_eat of Bread
pond -BIG_APPETITE-> b2_eat
auto converter b2_digest
from Bread to Droppings
b2_eat --> b2_digest
b2_digest --> road
pool b2_energy of Energy
source b2_life of Energy
b2_digest .*.> b2_energy
b2_life --> b2_energy
```

(b) Textual model that demonstrates code duplication

Fig. 2. Modeling two birds that both eat from the same pond

store without paying crystals? Is a desired state always reachable, e.g., can the game be won or can the level be finished?
- **Balance:** Are the rules well balanced?

2.5 Technical Challenges

Before answering these questions (in Section 2.6), we discuss engineering challenges and how to tackle them leveraging meta-programming, language workbenches and model checking.

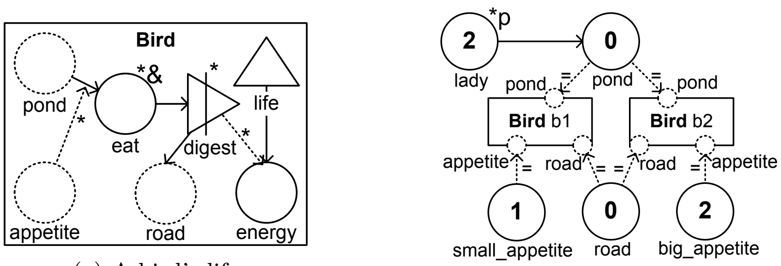
- **Parse:** To analyze any of these questions we need a representation that can easily be parsed. Therefore, MM introduces a textual representation of the game model, that serves as an intermediate format, that is compact and easy to read, parse, serialize and store.
- **Reuse:** Having a closer look at the example in Figure 2a, we see mirroring in the game graph that corresponds to code duplication in Figure 2b. We need modular constructs for reuse, encapsulation, scaling views, partial analysis and testing, and embedding MM in games (by way of connecting nodes and edges with in-game entities).
- **Inspect:** We need an environment that enables users to inspect states by visualizing serialized models.
- **Select:** Detailed insight in the game behavior can be obtained by interactively choosing successors and seeing transition alternatives. This is similar to debugging when stepping through code, and requires the calculation of alternatives. This can, for instance, reveal a lack of resources or capacity.

- **Analyze Context Constraints:** Some structural elements of models, related to contextual constraints can introduce errors that we want to catch statically. Examples are: (i) Sources cannot have inputs; (ii) Drains cannot have outputs; (iii) Edges are dead code if no active node can use them by pushing or pulling; and (iv) Edges are doubly used when both origin and target are pushing and pulling, which can lead to confusing results. Modeling errors can also be detected. Optionally, resource types of nodes can be defined making resource connections easily checkable. Additionally, missing references can be reported.
- **Analyze Reachability:** Analyzing reachability is hard because it requires calculation of all possible paths through the game graph. Normally, we cannot calculate all possible executions of programs due to the sheer number of possibilities, and use abstractions to allow forms of analysis. Because a MM model is itself an abstraction of the actual game, and types and instances —MM's modularization mechanism is described in more detail in Section 2.7— enable partial analysis, we can exhaustively verify models in an experimental context using model checking techniques. The challenge is to translate MM diagrams to models that a model checker can analyze, and making that analysis *scalable*. Non-deterministic choices lead to a combinatorial explosion of execution path and this results in a state explosion in the model checker. When searching for undesired situations, an exhaustive search may not be necessary, since the moment an invalid state is found, the execution stack trace represents a result.
- **Balance:** Providing useful analysis to support balancing games is very hard, since this requires analyzing multiple types of play, each dynamic with different *unpredictable* player choices and non-deterministic events. Experimental set-ups in which instance interfaces are subjected to modeled input may provide designers with useful feedback, but building such set-ups is hard and is the expertise of game designers.
- **Prototype and Adjust:** Prototyping game software and making adjustments requires code. In addition to the MM format we require a light-weight embeddable interpreter that enables using script for prototyping and adjusting game software. A simple API for integrating MM in existing architectures should at least provide a means for calculating successor states (step), observing pools value changes, activating interactive nodes and reading and storing information. We require that this API relates the run-time state of models to the state and the behavior of game elements that affect how the game behaves when played. This is not further explored in the current paper.

2.6 Answers to Game Designer's Questions

We will now answer the questions raised in Section 2.4 and illustrate them using the bird feeding example.

Figure 3 shows a rewrite of the example using new language elements to be detailed in Section 2.7. Figure 3a shows the definition of *Bird*, which references



(a) A bird's life

(b) A lady feeding two birds

Fig. 3. Graphically modeling birds that eat, digest and live

```
 1 Bird(ref appetite,ref pond,ref road)
 2 {
 3   //birds eat exactly all they want
 4   auto all pool eat of Bread
 5   pond -appetite-> eat
 6   auto converter digest
 7     from Bread to Droppings
 8   eat --> digest  //digest Bread
 9   digest --> road //produce Dropping
10   pool energy of Energy
11   source life of Energy
12   digest .*.> energy
13   life --> energy
14   assert fed: energy > 0 || road < 2
15     "birds␣always␣get␣fed"
16 }
```

(a) A bird's life

```
unit Bread     : "bread␣crumbs"
unit Droppings : "bird␣residue"
unit Energy    : "bird␣energy"
pool BIG_APPETITE of Bread at 2
pool SMALL_APPETITE of Bread at 1
//a lady throws crumbs in the pond
auto push all pool lady of Bread at 2
pool pond of Bread
pool road of Droppings
lady --> pond
Bird b1 //b1 has a big appetite
BIG_APPETITE .=.> b1.appetite
pond .=.> b1.pond   road .=.> b1.road
Bird b2 //b2 has a small appetite
SMALL_APPETITE .=.> b2.appetite
pond .=.> b2.pond   road .=.> b2.road
```

(b) A lady feeding two birds

```
lady-1->pond
step
pond-1->b1_eat
lady-1->pond
step
pond-1->b1_eat
b1_eat-1->b1_digest_drain
step
b1_eat-1->b1_digest_drain
b1_life-1->b1_energy
b1_digest_source-1->road
step
b1_life-1->b1_energy
b1_digest_source-1->road
step
violate b2_fed
```

(c) Bird b2 starves

Fig. 4. Textual model and analysis that shows birds with a big appetite starve

external nodes *pond*, *road* and *appetite*. These external nodes act as formal parameters of the Bird specification and are bound twice in Figure 3b. Figure 4a and Figure 4b show the textual equivalent of this model.

Next, we introduce assertions and pose that birds shall never starve by adding an assertion at line 14 of Figure 4a.

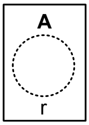
Then, we run the analysis to check for reachability and find that (i) bird *b2* starves because *b1_ eat* always happens before *big_ appetite* is available, and (ii) the acts of bird *b2_ eat* and *b2_ energy* are unreachable for all execution paths.

Finally, we can explore the model and understand it better by inspecting states, observing lack of alternative transitions, and automatically simulating the trace that lead to the assertion violation visually, shown textually in Figure 4c.

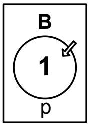
2.7 Language Extensions

We have designed MM and have introduced new language features as necessary to attain our goals. MM has modular constructs for reuse, encapsulation, scaling views, partial analysis and testing, and relating MM to in-game entities. MM has reduced non-determinism and increased control over competition for resources and capacity by introducing priorities. Time is modeled and understood, in a way that is embeddable in games. Finally, invariants are introduced for defining simple properties for analysis.

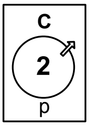
Types definitions and instances. The following table introduces[7] our modularization features *type definitions* and *instances*.

<table>
<tr>
<td>


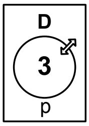
`A(ref r){ ... }`

Type definition reference definition
</td>
<td>


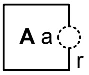
`B(in p){`
`pool p at 1 }`

Type definition input modifier
</td>
<td rowspan="2">
A *type definition* is a named diagram that functions as parameterized module for encapsulating elements. Type definitions define internal elements and how the they can be used externally. A *reference*, represented by a circle with a dashed line, is an alias that refers to a node that is defined externally. Internal nodes annotated with an *interface modifier input*, *output* or *input/output* become interfaces on the instances of the type. The input modifier denotes that an interface accepts inputs, output implies it accepts outputs and input/output accepts both. Interface modifiers appear as an arrow in the top right corner of a node, where an input modifier point into the node, an output modifier points out of the node, and an in-/output modifier does both.
</td>
</tr>
<tr>
<td>


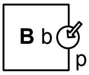
`C(out p){`
`pool p at 2 }`

Type definition output modifier
</td>
<td>


`D(inout p){`
`pool p at 3 }`

Type definition in-/output modifier
</td>
</tr>
<tr>
<td>


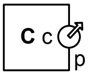
`A a`

Type instance reference interface
</td>
<td>


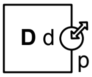
`B b`

Type instance input interface
</td>
<td rowspan="5">
An *instance* is a named object that has individual instance data, whose interfaces are defined by its type and can be bound to other models, acting as formal parameters.

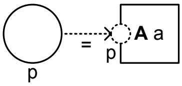
An *interface* makes internal elements of an instance available to the outside, and can be used by connecting resource connections. Visually, an interface is a small circle at the border of an instance with its name under it. Input interfaces have an arrow pointing into the circle, outputs have an arrow pointing outward, and in-/outputs have a bidirectional arrow. The direction of the arrow implies the validity of the direction of the edges that connect to it. Only reference interfaces appear with a dashed line.

References must be bound to definitions using edges called *bindings*, represented by dashed arrows annotated with an equal sign, that originate from a defining node and target a reference. Additionally, instances can be nested inside type definitions and build a name space, e.g., a nested `pool p` inside an instance `a` of type definition `A` is referred to as `a.p`.
</td>
</tr>
<tr>
<td>


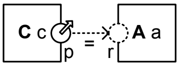
`C c`

Type instance output interface
</td>
<td>


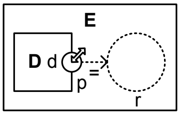
`D d`

Type instance in-/output interface
</td>
</tr>
<tr>
<td colspan="2">


`pool p   A a   p .=.> a.p`

Type instance with reference binding
</td>
</tr>
<tr>
<td colspan="2">


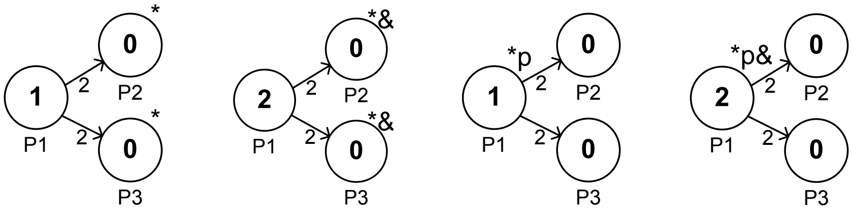
`C c   A a   c.p .=.> a.r`

Type instances with reference binding
</td>
</tr>
<tr>
<td colspan="2">


`E(inout p,ref r){D d  d.p .=.> r}`

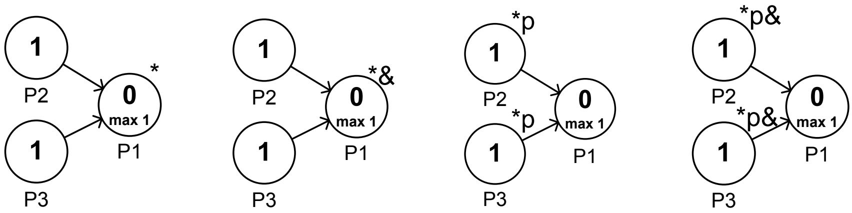
Type definition with nested instance and reference binding
</td>
</tr>
</table>

[7] Once again, for conciseness, only informally.



(a) P2 or P3 pulls (b) P2 or P3 pulls (c) Push to P2 or P3 (d) P1 cannot push

Fig. 5. Non-determinism due to shortage of resources



(a) Pull from P2 or P3 (b) P1 cannot pull (c) P2 or P3 pushes (d) P2 or P3 pushes

Fig. 6. Non-determinism due to shortage of capacity

Nodes have priorities. The sources of non-determinism that we have identified are nodes *competing* for resources and the *any* modifier. Alternative transitions exist due to lack of resources or capacity, as illustrated by Figure 5 and Figure 6.

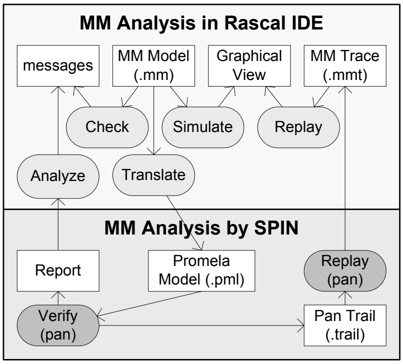
We have already mentioned that each activated node can act once during a step. Since the order in which nodes act is not defined, models under-specify behavior and this can result in undesirable non-determinism. To allow a degree of control, we specify that active nodes with the following actions and modifiers are scheduled in the following order: pull all, pull any, push all, push any. Groups of nodes from different categories do not compete for resources or capacity, which helps in analyzing models and in understanding them. Section 4 makes use of this feature.

Steps take time. MM does not support different *time modes* as Machinations does. In MM each node may act at most once during a step, which conforms to the Machinations notion of *synchronous time.* We do not support *asynchronous time*, in which user activated nodes may act multiple times during a step without affecting other nodes. Machinations supports a *turn-based mode*, in which players can each spend a fixed number of action points on activating interactive nodes each step. We note that *turns* are game assets that can be modeled, using pools, conditions and triggers, enabling turn-based analysis. MM does not specify how long a step takes, it only assumes that steps happen and its environment determines what the step intervals are.

Invariants. Defining property specifications to verify a model against can be hard, requiring knowledge of linear temporal logic. Defining invariants, Boolean expressions that must be true for each state, is easier to understand. MM adds *assertions* which consist of a name, a boolean expression that must invariantly be true, and a message to explain what happened when the assertion is violated, i.e. becomes false for some state. Figure 4a contains an example of an assertion (lines 14–15).

3 MM AiR Framework

Figure 7a shows the main functions of the MM Analysis in Rascal (MM AiR) framework and Figure 7b relates them to the challenges they address. The framework is implemented as a Rascal meta-program of approximately 4.5 KLOC. We will now describe the main functions of the framework.



(a) MM AiR IDE functions

§	functionality	challenges
3.1	*check* contextual constraints (parse, desugar, perform static analysis)	define syntax, semantics, reuse, constraints
3.2	*simulate* MM model (interpret and evaluate successor states, interactive graphical visualizations)	make models debuggable, improve scalability and performance
3.3	*translate* MM to Promela	relate formalisms, ensure interoperability, improve scalability
3.4	*verify* MM in Spin	ensure interoperability, improve scalability
3.5	*analyze* reachability	ensure interoperability
3.6	*replay* behaviors and verification results	source level debugging, ensure interoperability, readability

(b) Sections, functions and challenges

Fig. 7. MM AiR Overview

3.1 Check Contextual Constraints

Starting with a grammar for MM's textual syntax, using Rascal we generate a basic MM Eclipse IDE that supports editing and parsing textual MM models with syntax highlighting. This IDE is extended with functionality to give feedback when models are incorrect or do not pass contextual analysis. This is implemented in a series of model transformations, leveraging Rascal's support for pattern matching, tree visiting and comprehensions. This includes labeling the model elements, for storing information in states and for resource redistributions in transitions. We check models against the contextual constraints described in Section 2.5.

3.2 Simulate Models

Simulate provides a graphical view of a MM model and enables users to inspect states, choose transitions and successors, and navigate through the model by stepping forward and backward. We generate figures and interactive controls for simulating flattened states and transitions. This is easily done by applying RASCAL's extensive visualization library, which renders figures and provides callbacks we use to call an interpreter. The *interpreter* calculates successor states by evaluating expressions, checking conditions and generating transitions.

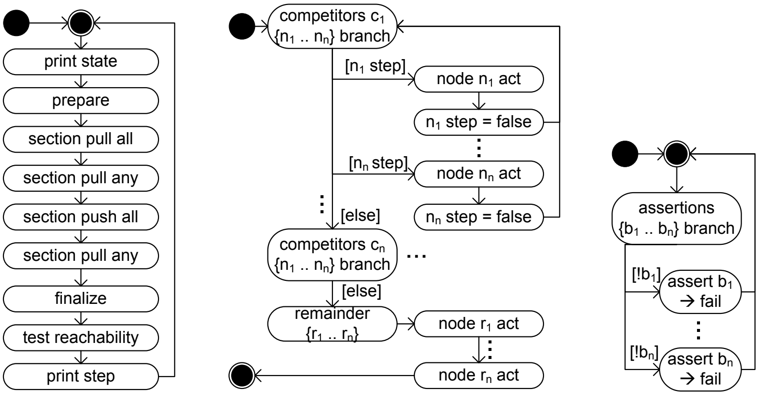
3.3 Translate to PROMELA

The biggest challenge in analyzing MM is providing a scalable reachability analysis. We achieve this by translating MM to PROMELA, the input language of the SPIN model checker. A naive approach is to model each node as a process, enabling every possible scheduling permutation to happen. However, not every scheduling results in a unique resource distribution, which hampers performance and scalability. Therefore we take steps to reduce the number of calculations without excluding possible behaviors. We take the following measures to reduce the state space explosion.

- **Reduce non-determinism.** We model only necessary non-determinism. We have identified two sources that are currently in MM: nodes *competing* for resources or capacity and the *any modifier*. For competing nodes every permutation potentially results in a unique transition that must be computed, but nodes that do not compete can be sequentially processed.
- **Avoid intermediate states.** PROMELA has a `d_step` statement that can be used to avoid intermediate states, by grouping statements in single transitions.
- **Store efficiently and analyze partially.** Pools can specify a maximum that we use to specify which type to use in PROMELA (bit, byte or int), minimizing the state vector. For partial analysis we can limit pool capacities.

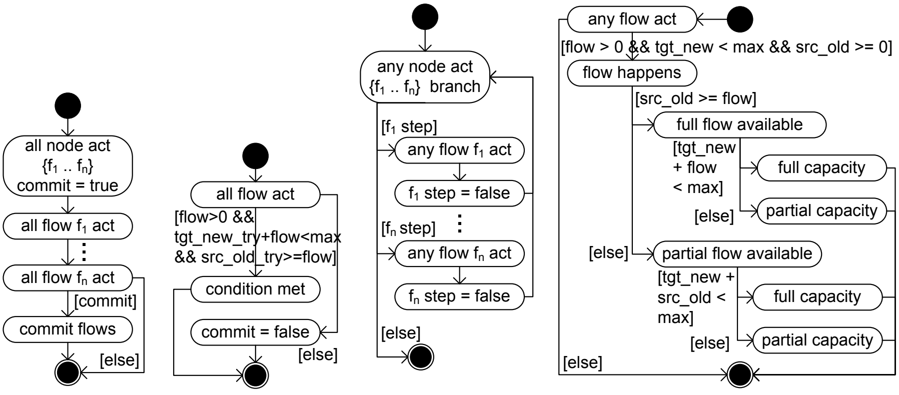
Translating an MM model to PROMELA works as follows. We bind references to definitions and transform the model to core MM. We generate one `proctype` per model, schematically shown in Figure 8, and a monitor `proctype` that tests assertions for each state. Figure 8a depicts their general structure. At the beginning of a step the state is printed, and step guards are enabled if a node is active. This is followed by *sections* for each priority level as determined by node type. In each section, groups of nodes may be competing for resources or capacity.

For each group of competitors c_i consisting of nodes n_1, ..., n_n, we introduce a non-deterministic choice using guards that are disabled after a competing node acts as shown in Figure 8b. The remaining independent nodes r_1, ..., r_n are just sequentially processed, since they never affect each other during a step. Figure 8c shows that each path in the monitor process remains blocked until an invariant becomes false, and a violation is found.



(a) Process (b) Section (c) Monitor

Fig. 8. Skeleton for generated PROMELA code: process, section and monitor



(a) All Node (b) All Flow (c) Any Node (d) Any Flow

Fig. 9. Skeleton of generated PROMELA code for nodes

The behavior of nodes with the *all modifier* is deterministic, as shown in Figure 9a and Figure 9b. All flows f_1, ..., f_n are executed sequentially and per flow conditions are checked. The effect of all flows is only committed if the conditions for all flows have been satisfied.

The behavior of nodes with the *any modifier* is shown in Figure 9c and Figure 9d models the non-determinism by introducing a non-deterministic choice between the flows f_1, ..., f_n.

Individual nodes act by checking shortages of resources on the old state from which subtractions are made and check shortage of capacity on the new state, to which additions are also made. Finally, when each node has acted the state is finalized by copying the new state to the current state. Temporary values and guards are reset, and active nodes are calculated by applying activation modifiers, triggers and conditions. Next reachability is tested, the step is printed and we start at the beginning to determine the next step.

3.4 Verify Invariant Properties

MM models are verified against their assertions by translating them to Promela and then running a shell script. The script invokes Spin and compiles it to a highly optimized model-specific Promela analyzer (Pan). It then runs this verifier to perform the state space exploration, and captures the verification report Pan outputs, which may contain unreached states and associated Promela source lines. If the verifier finds an assertion violation, it also produces a *trail*, a series of numbers that represent choices in the execution of the state machine representing the Promela model. The challenge is interoperability, relating the verification report and the trail back to MM and showing understandable feedback to the user. We show how this is solved in Section 3.5 and Section 3.6.

3.5 Analyze Reachability

We tackle the interoperability challenge of relating a Spin reachability analysis to MM as follows. During the generation of Promela we add *reachability tests*, in which states and source lines become reachable if an element acts. We collect the source lines using a tiny language called MM Reach, which specifies the test case by defining whether a node receives full or partial flow via a resource connection or if it activates a trigger. We extract unreached Promela source lines from the Pan verification report and map them back to MM elements to report the following messages, which are relative to a partial or exhaustive search.

- **Starvation.** Nodes that never push or pull full or partial flow via a resource connection starve, and represent dead code.
- **Drought.** A resource connection through which resources do not flow runs dry, and is unused dead code.
- **Inactivity.** A trigger that never activates its target node is idle.
- **Abundance.** A node with the any modifier that always receives full flow along all of its resource connections indicates a lack of shortage.

3.6 Replay Behaviors

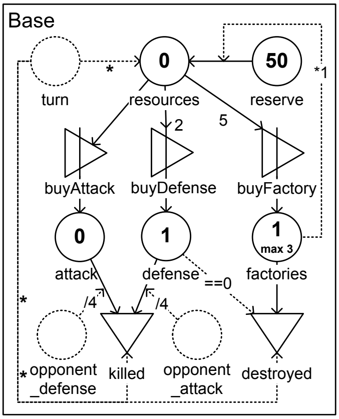
We tackle the interoperability challenge of relating Pan trails for Promela models we obtained in Section 3.4 to MM model resource redistributions by introducing an intermediate language called MM Trace (MMT). A sequence of MMT statements forms a program that contains the transitions that an MM model performs, which MM AiR graphically replays in a guided simulation.

Replaying a trail on Pan simulates the steps of a Promela model while calling `printf` statements that generate an MMT program, ending in an assertion violation. The program is obtained by embedding the following MMT statements prefixed with `MM:` for filtering in the Promela model.

- **Flow.** Node causes flow to occur: `source-amount->target`
- **Trigger.** Trigger activates a target node in the next state: `trigger node`
- **Violation.** A state violates an assertion: `violate name`
- **Step.** Terminate a transition: `step`

4 Case Study: SimWar

SimWar is a simple hypothetical Real-Time Strategy (RTS) game introduced by Wright [11] that illustrates the game design challenge of *balancing* a game. This entails ensuring different player choices and strategies represent engaging and challenging experiences. Common strategies for RTS games are *turtling*, a low-risk, long-term strategy that favors defense, and *rushing*, a high-risk short-term strategy that favors attack. Adams and Dormans [4] study the game using the Machinations tool. By simulating many random runs, they show the game is indeed poorly balanced and that turtling is the dominant strategy.



(a) SimWar Base

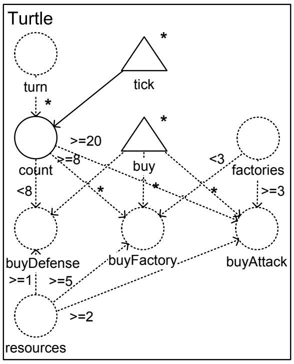
```
 1 Base(in BuyAttack, in BuyFactory, in BuyDefense,  //choices
 2      ref opponent_attack, ref opponent_defense, ref turn,
 3      out attack, out defense, out factories, out resources){
 4   turn .*.> resources //turn triggers resources to pull
 5   turn .*.> killed    //turn triggers killed
 6   turn .*.> destroyed //turn triggers destroyed
 7   pool reserve of Gold at 50  //Gold reserve (starts at 50)
 8   pool resources of Gold      //Gold resources (for purchases)
 9   pool factories of Factory at 1 max 3 //factories for income
10   pool defense of Defense at 1 //defending units
11   pool attack of Attack        //attacking units
12   drain killed of Defense, Attack //units can be killed
13   drain destroyed of Factory      //factories can be destroyed
14   converter buyDefense from Gold to Defense //buy defense
15   converter buyAttack  from Gold to Attack  //buy attack
16   converter buyFactory from Gold to Factory //buy factory
17   reserve -factories-> resources //produce income
18   resources -5-> buyFactory //buyFactory consumes 5 Gold
19   buyFactory --> factories  //buyFactory produces 1 Factory
20   resources -1-> buyDefense //buyDefense consumes 2 Gold
21   buyDefense --> defense    //buyDefense produces 1 Defense
22   resources -2-> buyAttack  //buyAttack consumes 1 Gold
23   buyAttack --> attack      //buyAttack produces 1 Attack
24   factories -all-> destroyed           //factories destuction
25   defense -opponent_attack/4-> killed  //defense casualty rate
26   attack  -opponent_defense/4-> killed //attack casualty rate
27   defense .defense == 0.> destroyed    //undefended condition
28 }
```

(b) SimWar Base

Fig. 10. The rules of SimWar

Our MM adaptation of SimWar, shown in Figure 10, is based on [4], but it models the rules for players in a definition called *Base*, avoiding duplication. It also replaces probabilities on resource connections with amounts. Two players compete by spending *resources*, choosing to buy *defense* (cost 1), *attack* (cost 2) or *factories* (cost 5). This is modeled by three converters in line 15–17 of Figure 10b that pull their respective costs from *resources* when activated.

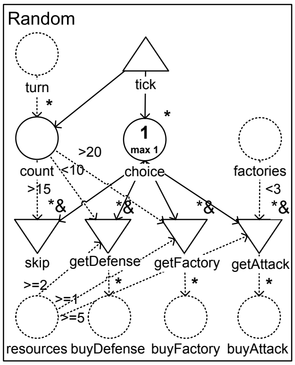
Factories produce income every turn, and represent an investment enabling more purchases. We model this by *turn* triggering *resources* (line 5), which pulls from *reserve* (line 9) the current amount of factories (line 18). A player must destroy their opponent's factories to win. Two references, *opponent_ defense* and *opponent_ attack* determine the (rounded down) casualty rate of one in four (line 26, 27) for *attack* and *defense* respectively. Opponents fight until one player has no defense, and her factories are destroyed (line 28).



(a) Turtle Strategy

```
Turtle(ref buyAttack, ref buyDefense,
       ref buyFactory, ref factories,
       ref resources, ref turn){
  source tick
  turn .*.> count
  tick --> count
  pool count
  auto source buy
  buy .*.> buyAttack
  buy .*.> buyFactory
  buy .*.> buyDefense
  count     .>=20.> buyAttack
  factories .>=3.>  buyAttack
  resources .>=2.>  buyAttack
  count     .<8 .>  buyDefense
  resources .>=1.>  buyDefense
  count     .>=8.>  buyFactory
  factories .<3.>   buyFactory
  resources .>=5.>  buyFactory
}
```

(b) SimWar Turtle



(c) Random Strategy

```
Random(ref buyAttack, ref buyDefense,
       ref buyFactory, ref factories,
       ref resources, ref turn){
  source tick
  turn .*.> count
  tick --> count
  pool count
  tick --> state
  auto pool state max 1
  auto all drain skip
  auto all drain getFactory
  auto all drain getAttack
  auto all drain getDefense
  getAttack  .*.> buyAttack
  getFactory .*.> buyFactory
  getDefense .*.> buyDefense
  state --> skip
  state --> getAttack
  state --> getDefense
  state --> getFactory
  count     .>15.>   skip
  resources .>= 2.>  getAttack
  count     .>= 20.> getAttack
  resources .>= 1.>  getDefense
  count     .<10.>   getDefense
  resources . >= 5.> getFactory
  factories . <3.>   getFactory
}
```

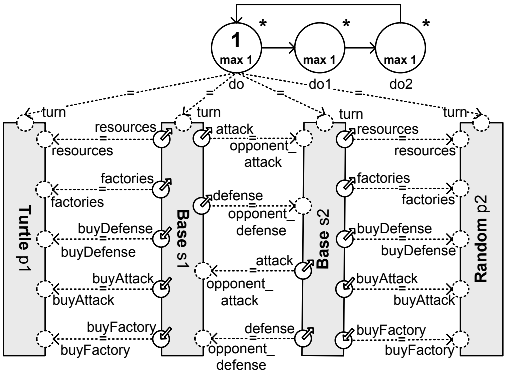
(d) SimWar Random

Fig. 11. SimWar Test Strategies

4.1 Experimental Setup

In an experiment with SimWar and two strategies shown in Figure 11 we apply the MM AiR framework, analyzing (i) the reachability of modeling elements, and (ii) the existence of a strategy that beats a turtling strategy.

The *Turtle* strategy, defined in Figure 11a and Figure 11b, simply counts turns, and based on this triggers references for buying. The *Random* strategy defined in Figure 11c and Figure 11d also counts, but adds a non-deterministic element which uses priorities. Drains *skip*, *getDefence*, *getFactory*, *getAttack* compete for the resource in *choice* before it pulls a resource from *tick*, enabling the next choice. In our test set-up shown in Figure 12, we bind instances of Random and Turtle to a Base instance in lines 16–20 & 25-29 of Figure 12b. We bind base instances as opponents in lines 13–14, 22–23 and bind *turn* to *doit*, our means for activity. Finally, we assert in lines 30–31 that the factories of Turtle are never destroyed. A violation of this assertion represents a behavior of Random that beats Turtle.

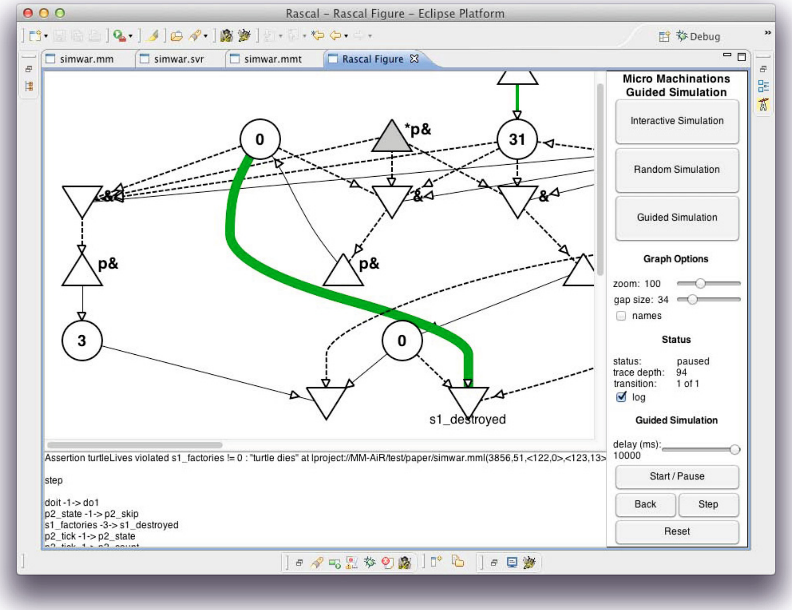


(a) A Turtle instance battling a Random instance

```
unit Gold    : "gold"
unit Factory : "factories"
unit Defense : "defense"
unit Attack  : "attack"
Turtle p1 Base s1 //player p1 is turtling
Random p2 Base s2 //player p2 is random
auto all pool doit at 1 max 1
auto all pool do1 max 1
auto all pool do2 max 1
doit -->  do1 do1 -->  do2 do2 -->  doit
doit.==1.>do1 do1.==1.>do2 do2.==1.>doit
doit          .=.> s1.turn
s2.defense    .=.> s1.opponent_defense
s2.attack     .=.> s1.opponent_attack
doit          .=.> p1.turn
s1.resources  .=.> p1.resources
s1.buyAttack  .=.> p1.buyAttack
s1.buyFactory .=.> p1.buyFactory
s1.buyDefense .=.> p1.buyDefense
s1.factories  .=.> p1.factories
doit          .=.> s2.turn
s1.defense    .=.> s2.opponent_defense
s1.attack     .=.> s2.opponent_attack
doit          .=.> p2.turn
s2.resources  .=.> p2.resources
s2.buyAttack  .=.> p2.buyAttack
s2.buyFactory .=.> p2.buyFactory
s2.buyDefense .=.> p2.buyDefense
s2.factories  .=.> p2.factories
assert turtleLives:
  s1.factories != 0 "turtle␣dies"
```

(b) SimWar Battle

Fig. 12. SimWar experimental test setup



zoom: 100
gap size: 34
□ names
status: paused
trace depth: 94
transition: 1 of 1
☑ log
delay (ms): 10000

```
Assertion turtleLives violated s1_factories != 0 : "turtle dies" at |project://MM-AiR/test/paper/simwar.mml(3856,51,<122,0>,<123,13>

step

doit -1-> do1
p2_state -1-> p2_skip
s1_factories -3-> s1_destroyed
p2_tick -1-> p2_state
```

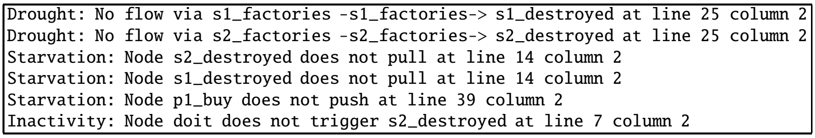
Fig. 13. MM AiR playing back a counter-example showing *Turtle* defeated

4.2 Experimental Results

We apply MM AiR by translating the models to Promela and running Spin. Pan reports using 2500MB of memory, mostly for storing 10.5M states of 220 bytes, generating 188K states/second, taking 56 seconds on an Intel Core i5-2557M CPU. It reports 11.9M transitions, of which 9.5M are atomic steps, and an assertion violation (s1_factories!=0) at depth 8810.

The shortest trail yields an MMT file of 95 steps. Figure 13 shows its graphical play-back. We find 22 strategies that beat our Turtle behavior, but these strategies all fall into the turtling category, confirming the strategy is dominant.

During its limited state space exploration, Pan collects unreached Promela source lines. Using these, our analysis reports the following:



```
Drought: No flow via s1_factories -s1_factories-> s1_destroyed at line 25 column 2
Drought: No flow via s2_factories -s2_factories-> s2_destroyed at line 25 column 2
Starvation: Node s2_destroyed does not pull at line 14 column 2
Starvation: Node s1_destroyed does not pull at line 14 column 2
Starvation: Node p1_buy does not push at line 39 column 2
Inactivity: Node doit does not trigger s2_destroyed at line 7 column 2
```

Initially puzzled by the first drought and the second starvation message, we concluded that the assertion in the monitor process is violated before the reachability check happens. Indeed node *p1_buy* never pushes, since it has no resource connections, it serves only to trigger choices.

The final message of inactivity tells us that *s2_destroyed* is never triggered by *doit*, the binding of *turn*. This experiment shows MM AiR provides feedback for analyzing and refining MM models intended to be embedded in game software.

5 Conclusions

Machinations was a great first step in turning industrial experience in game design into a design language for game economies. In this paper we have taken the original Machinations language as starting point and have analyzed and scrutinized it. It turned out that the definitions of various of the original language elements were incomplete or ambiguous and therefore not yet suitable for a formal analysis of game designs. During this exercise, we have learned quite a few lessons:

- Formal validation of rules for game economies is feasible.
- Unsurprisingly, modularity is a key feature also for a game design language. Modularity not only promotes design reuse, but also enables modular validation that can significantly reduce the state space.
- In our refinement and redefinition of various language features, we have observed that non-determinism had to be eliminated where possible in order to reduce the state space.
- While a graphical notation is good for adoption among game designers, a textual notation is better for tool builders.
- Promela is a flexible language that offers many features to represent the model to be validated. Different representation choices lead to vastly different performance of the model checker and it is non-trivial to choose the right representation for the problem at hand.

- The RASCAL language workbench turned out to be very suitable for the design and implementation of MM AiR. In addition to compiler-like operations like parsing and type checking MM AiR also offers editing, interactive error reporting and visualization. It also supports generation of PROMELA code that is shipped to the SPIN model checker and the resulting execution traces produced by SPIN can be imported and replayed in MM AiR.

MM Air in its current form is an academic prototype, but it is also a first step towards creating embeddable libraries of reusable, validated, elements of game designs. Next steps include the use of probabilistic model checkers, mining of recurring patterns in game designs and finally designing and implementing embeddable APIs for MM. These will form the starting point for further empirical validation. We see as the major contributions of the current paper both the specific design and implementation of MM and MM AiR and the insight that the combination of state-of-the-art technologies for meta-programming and model checking provide the right tools to bring game design to the next level of productivity and quality.

Acknowledgements. We thank Joris Dormans for answering our many questions about Machinations, Tijs van der Storm for providing advice and feedback, and the anonymous reviewers for giving valuable suggestions.

References

1. Blow, J.: Game Development: Harder Than You Think. ACM Queue 1, 28–37 (2004)
2. Klint, P., van der Storm, T., Vinju, J.: EASY Meta-programming with Rascal. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 222–289. Springer, Heidelberg (2011)
3. Holzmann, G.: SPIN Model Checker, the: Primer and Reference Manual, 1st edn. Addison-Wesley Professional (2003)
4. Adams, E., Dormans, J.: Game Mechanics: Advanced Game Design, 1st edn. New Riders Publishing, Thousand Oaks (2012)
5. Dormans, J.: Level Design as Model Transformation: A Strategy for Automated Content Generation. In: Proceedings of the 2nd International Workshop on Procedural Content Generation in Games, PCGames 2011, ACM, New York (2011)
6. Brom, C., Abonyi, A.: Petri Nets for Game Plot. In: Proceedings of Artificial Intelligence and the Simulation of Behaviour (AISB) (2006)
7. Araújo, M., Roque, L.: Modeling Games with Petri Nets. In: Proceedings of the 3rd Annual DiGRA Conference Breaking New Ground: Innovation in Games, Play, Practice and Theory (2009)
8. Fu, D., Houlette, R., Jensen, R.: A Visual Environment for Rapid Behavior Definition. In: Proc. Conf. on Behavior Representation in Modeling and Simulation (2003)
9. Champandard, A.J.: Behavior Trees for Next-Gen Game AI (December 2007), `http://aigamedev.com`
10. McNaughton, M., Cutumisu, M., Szafron, D., Schaeffer, J., Redford, J., Parker, D.: ScriptEase: Generative Design Patterns for Computer Role-Playing Games. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 88–99. IEEE Computer Society, Washington, DC (2004)
11. Wright, W.: Dynamics for Designers. Lecture delivered at the Game Developers Conference (2003)