
Copyright:
© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

DOI link to article:
http://dx.doi.org/10.1109/PDP.2015.107

Date deposited:
23/12/2015
A formal specification and prototyping language for multi-core system management.

A. Iliasov, A. Rafiev, F. Xia, R. Gensh, A. Romanovsky, A. Yakovlev
School of Computing Science, Newcastle University, Newcastle upon Tyne, UK

Abstract—We relate the experience of defining a formal domain specific language (DSL) for the construction and reasoning about OS-level management logic of multi-core systems. The approach is based on a novel, iterative development principle where results of prototyping studies feed back into the next language revision. We illustrate the DSL with several examples of executable scripts.

I. INTRODUCTION

With the end of processor frequency growth, scaling into the direction of multi- and many-core systems has become the primary way to translate the advances in manufacturing technology into increased computer performance. This change has already started to affect operating systems and application software design. Clearly, we have to plan now for the many-core systems of tomorrow.

In this paper we propose a method and toolkit that has grown around an effort to build a database of knowledge related to multi-core systems. The exercise started with a simple compilation of fact using a natural language. The result turned out to be inadequate due to many ambiguities, hidden assumptions and subtle inter-relations between concepts. A decision was made to transfer into a formal specification language. We have chosen Event-B modelling language [1] mainly for our previous experience with it. The formal specification of the knowledge database resolved the known ambiguities and exposed many omissions and inconsistencies. The effort took several months and resulted in a substantial model with hundreds of properties. It was expected that the model is consulted when designing or writing OS-level software for multi-core system. It turned out that a formal model of this scale is a difficult read and not a practical blueprint to relate with actual software. Moreover, there was a lack of confidence in the model completeness and liveness properties - conditions that are impractical to address with static theorem proving and turned out impossible to delegate to a model checker due to model complexity and scale.

To address these weaknesses while staying in the formal domain, it was decided to build a control flow extension to Event-B that would "drive" (to take the term from a conceptual inspiration - the CSP[B technique [2]]) the original Event-B model. The pursuit of this goal led to the development of a novel approach to the construction of formal imperative-style domain-specific languages where the heart of the language (its domain-specific part such as commands, variables and constants) is designed in a separate formal notation (Event-B) while the "glue" part (control flow constructs like if and while) are generic and reused by all DSL instances. The method and the accompanying tools are referred to as DSL-Kit.

The most profound implication is the facilitation of iterative design: design prototypes realised in a DSL would often highlight deficiencies in the DSL itself (lack of progress, missing concepts); this may be addressed by going back to Event-B specification of the domain, doing necessary changes, proving them correct and automatically, with the help of a translation tool, transferring the result into DSL-Kit to construct a new instance of domain DSL. DSL-Kit itself offers fairly reach reasoning facilities. On top of this, it can execute (with many limitations) DSL specifications and a custom state visualisation may be plugged in to facilitate debugging and design comprehension.

The first part of the paper (Section III) is a narrative build around an Event-B model that attempts to capture the essence of multi-core systems from the mostly software and OS perspective. The model does not include any notions of control logic. It rather gives a concrete definition of the subjects of the prospective control system and attempts to explain their inter-relationships; it also describes the general life cycle of a multi-core system.

The second part (Section IV) describes the transition in the DSL-Kit environment and relates the initial experience with the design of an adaptive run-time controller. We show how the domain model may be turned into a formal virtual machine over which control logic executes. The reasoning style also changes from refinement proofs and inductive verification of safety invariant of Event-B to the verification of imperative programs in the style of Floyd-Hoare logic [3].

II. EVENT-B

A. Event-B

The basis of our discussion is a formalism called Event-B [1]. It belongs to a family of state-based modelling languages that represent a design as a combination of state (a vector of variables) and state transformations (computations updating variables).

An Event-B development starts with the creation of a very abstract specification. A cornerstone of the Event-B method is the stepwise development that facilitates a gradual design of a system implementation through a number of correctness-preserving refinement steps. The general form of an Event-B model (or machine) is shown in Figure 1. Such a model encapsulates a local state (program variables) and provides operations on the state. The actions (called events) are characterised by a list of local variables (parameters) \( \mathit{vl} \), a state predicate \( g \) called event guard, and a next-state relation \( S \) called substitution or event action.

Event guard \( g \) defines the condition when an event is enabled. Relation \( S \) is given as a generalised substitution statement [4] and is either deterministic (\( x := 2 \)) or non-deterministic update of model variables. The latter kind comes in two notations: selection of a value from a set, written as...
MACHINE M
SEES Context
VARIABLES v
INITIALISATION I(c, s, v)
EVENTS
E₁ ≝ any v l where g(c, s, v, v) then S(c, s, v, v, v') end

END

Fig. 1. Event-B machine structure.

x : {2, 3}; and a relational constraint on the next state v', e.g., x : x' ∈ {2, 3}.

III. DOMAIN MODEL

A. Event-B domain model specification

  a) Cores: A system contains a number of cores; at any moment a core may be operating or switched off. The set of all cores in a system is defined by a finite and non-empty set CORES. Current core status (on or off) is given by function variable status:

\[
\text{status} \in \text{CORES} \rightarrow \text{STATUS}
\]

where STATUS = \{ON, OFF\}. At this level, we may observe a core being switched on or off, as captured by the following two events:

\[
\begin{align*}
on &\text{ any } c \text{ where status}(c) = \text{OFF} \text{ then status}(c) := \text{ON} \text{ end} \\
off &\text{ any } c \text{ where status}(c) = \text{ON} \text{ then status}(c) := \text{OFF} \text{ end}
\end{align*}
\]

b) Frequency and voltage: Two essential characteristics of a running core are the voltage of its power supply and the clock frequency. These are the principal attributes used to control core performance, power and reliability. The following partial functions define frequency and voltage of a core; these are undefined for cores switched off.

\[
\begin{align*}
freq &\in \text{CORES} \rightarrow \mathbb{N} \quad \text{vdd} \in \text{CORES} \rightarrow \mathbb{N} \\
dom(freq) = \text{status}^{-1}(\{\text{ON}\}) &\quad \dom(vdd) = \text{status}^{-1}(\{\text{ON}\}) \\
freq := \varnothing &\quad \text{vdd := } \varnothing
\end{align*}
\]

Since all the cores are initially off, voltage and frequency functions are initially undefined. Voltage and frequency attributes are changed dynamically by event core_dvfs:

\[
\begin{align*}
\text{core_dvfs} &\triangleq \\
\text{any } c, f, v \text{ where status}(c) = \text{ON} \land \ldots &\text{then} \\
freq(c) := f &\text{vdd}(c) := v \ldots \text{end}
\end{align*}
\]

The dots stand for omitted clauses; in a refined version, the event checks whether frequency/voltage pairs are correct for a given core, and also whether power budget and temperature constraints are satisfied for the new settings. Switching a core on also necessitates setting some initial voltage and frequency values..

B. Heat generation

Heat exchange happens between a core and the environment and a core and its neighbouring cores. The environment is assumed to possess infinite heat capacity so that its temperature is not affected by heat exchange.

For some core c, the amount of heat generated per unit of time is determined by core frequency f and core voltage V, according to the following law:

\[
\frac{dH}{dt} = C_c f c V^2
\]

where \(C_c\) is a core-specific constant. Heat rate \(\frac{dH}{dt}\) and heat exchange laws define core temperature delta over a time period. For any given core, given its heat capacitance, we can determine the time duration necessary to increase core temperature by one Kelvin. Alternatively, considering some fixed time period, we can determine temperature change caused by the heat rate during the period.

The following event controls core temperature change. At this step, it is un-timed and time will be added in a later refinement step.

\[
\begin{align*}
\text{core_temp} &\triangleq \\
\text{any } c \text{ where status}(c) = \text{ON} &\text{then} \\
temp(c) := \max(\text{temp}(c) + \text{HRATE}(c) \rightarrow vdd(c) \rightarrow freq(c) + \sum_{n \in \text{NRATE}(c)} \text{NRATE}(n \rightarrow c \rightarrow vdd(c) \rightarrow freq(c) - \text{ERATE}(c), 0)\} &\text{end}
\end{align*}
\]

In the above HRATE(...), NRATE(...) and ERATE(...) are the heat rates times time delta for the heat gained through resistive heating, heat exchange with neighbour cores and heat loss to the environment. When a core overheats, the system immediately (we shall clarify the meaning of immediacy with the introduction of time) switches off the core.

\[
\begin{align*}
\text{core_shutdown} &\triangleq \\
\text{any } c, t \text{ where status}(c) = \text{ON} &\text{then} \\
\text{temp}(c) > \text{CORE_TEMP_CRIT}(c) \land \ldots &\text{then} \\
\text{status}(c) := \text{OFF} \| \ldots &\text{end}
\end{align*}
\]

We shall later see that shutting down a core cancels all the jobs and stops all the running threads. All the computation progress is irrecoverably lost.

  c) Threads: A thread is a basic concurrency and program structuring unit in our model. By a thread we understand a potentially infinite sequence of commands that are continuously or intermittently executed by a core, perhaps switching between cores during its lifetime. At any given time, there is some number (potentially zero) of threads in the system.

\[
\text{threads} \subseteq \text{THREADS}
\]

where THREADS is the universe of threads. A thread may be assigned to a core and then it is said to be running or scheduled. The thread/core association is functional and partial in the domain:

\[
\text{affinity} \in \text{threads} \rightarrow \text{CORES}
\]

It is only possible to schedule a thread on a running core:

\[
\text{run(affinity)} \subseteq \text{status}(\{\text{ON}\})
\]

Note that affinity is functional in one direction only (i.e., it is not injective); indeed, it is possible to map several threads onto the same core, for instance,

\[
t, h \in \text{threads} \land t \neq h \land \text{affinity}(t) = \text{affinity}(h)
\]

describes a situation where distinct threads t, h are running on a same core. In our model we choose to fix the lowest time resolution at the scale of ~ 1 millisecond; thus for a sequence of events with overall time smaller than this limit, we see all
the events happening at the same time or concurrently. This is a standard abstraction technique and it allows one to conduct a formal refinement to a higher time resolution.

A new thread may be added to the system and, at some point, it may be destroyed:

\[
\begin{align*}
\text{th\_start} & \triangleq \text{any } t \text{ where } t \not\in \text{threads} \text{ then threads} := \text{threads} \cup \{t\} \text{ end} \\
\text{th\_stop} & \triangleq \text{any } t \text{ where } t \in \text{threads} \text{ then threads} := \text{threads} \setminus \{t\} \text{ end}
\end{align*}
\]

An existing thread may be scheduled to run on some operating system core \(c\). An already running thread may be unscheduled and be left in a dormant state to be scheduled again:

\[
\begin{align*}
\text{thread\_schedule} & \triangleq \text{any } t, c \text{ where } \\
& \text{t} \in \text{threads} \setminus \text{dom(affinity)} \text{ then affinity}(t) := c \\
& \text{status}(c) = \text{ON} \text{ end} \\
\text{thread\_unschedule} & \triangleq \text{any } t, c \text{ where } \\
& t \in \text{dom(affinity)} \text{ then affinity}(t) := \{\} \text{ end}
\end{align*}
\]

\(d)\) Application: Several threads are grouped into an application. The primary role of an application is to define workload type shared by a number of threads. Applications may be created and destroyed during the system lifetime. The set of current applications is defined by set

\[\text{apps} \subseteq \text{APPS}.\]

Each thread belongs to an application and each application owns at least one thread. This is captured by the following (surjective and total) relation

\[\text{app\_threads} \in \text{apps} \leftrightarrow \text{threads}.\]

For each thread there is just one owning application:

\[\text{app\_threads}^{-1} \in \text{threads} \rightarrow \text{apps}.\]

When an application is created, it appears together with one thread of its own (this even refines event \text{thread\_start}):

\[
\begin{align*}
\text{app\_start} & \triangleq \text{any } a, t \text{ where } \\
& a \notin \text{apps} \land t \notin \text{threads} \text{ then } \\
& \text{apps} := \text{apps} \cup \{a\} \text{ end} \\
\text{app\_threads} := \text{app\_threads} \cup \{a \mapsto t\}
\end{align*}
\]

Destroying an application cancels all the running application threads and removes them from the system:

\[
\begin{align*}
\text{app\_stop} & \triangleq \text{any } a, t \text{ where } \\
& t \in \text{threads} \land a \mapsto t \in \text{app\_threads} \land \text{app\_threads}\{a\} = \{t\} \text{ then } \\
& \text{threads} := \text{threads} \setminus \{t\} \text{ end} \\
& \text{affinity} := \text{affinity} \setminus \{t\} \text{ end} \\
& \text{apps} := \text{apps} \setminus \{a\} \text{ end} \\
\text{app\_threads} := \text{app\_threads} \setminus \{a \mapsto t\}
\end{align*}
\]

e) Workload: The purpose of an application is to provide a computation service. This it accomplishes by assigning incoming \textit{workload} to application threads. The unit of a workload is a \textit{job}. Every job is specific to an application so that only threads of a certain application may process a given job.

The pending and executing jobs of a systems are defined by variable \textit{jobs}:

\[\text{jobs} \in \text{JOBS} \rightarrow \text{apps}\]

where \text{JOBS} is the universe of all jobs. To run a job, it must be allocated to a thread. At any given time a thread executes at most one job. The following partial injection captures this relationship:

\[\text{job\_alloc} \in \text{dom(\text{job\_alloc})} \rightarrow \text{threads}\]

A job allocation must agree with the ownership of a thread to which the job is assigned:

\[\text{jobs}^{-1}\circ \text{job\_alloc} \subseteq \text{app\_threads}\]

Here \text{jobs}^{-1} is the converse of function \text{jobs} and \(f; h\) denotes forward functional composition.

To reason about computational complexity of a job, we define the number of \textit{steps} comprising a given job. A step is a normalised complexity measure independent of core properties and shared by all the jobs:

\[\text{job\_steps} \in \text{dom(jobs)} \rightarrow \mathbb{N}\]

A step execution time varies from core to core and with core frequency. A new job may appear at any moment; it must be assigned to an existing application.

\[
\begin{align*}
\text{job\_create} & \triangleq \text{any } j, a, w \text{ where } \\
& j \notin \text{dom(jobs)} \land a \in \text{apps} \land w > 0 \text{ then } \\
& \text{jobs} := \text{jobs} \cup \{j \mapsto a\} \text{ end} \\
& \text{job\_steps}(j) := w \text{ end}
\end{align*}
\]

In the above, \(w\) defines the job complexity in the terms of steps. An existing but yet unassigned job may be allocated to a thread of the job application. The thread in question must be mapped to a core but not already executing any other job:

\[
\begin{align*}
\text{job\_allocate} & \triangleq \text{any } j, t, \text{ where } \\
& j \in \text{dom(jobs)} \setminus \text{dom(\text{job\_alloc})} \land t \in \text{app\_threads}\{\text{jobs}(j)\} \land \text{ran(\text{job\_alloc})} \cap \text{dom(affinity)} \text{ then } \\
& \text{job\_alloc} := \text{job\_alloc} \cup \{j \mapsto t\} \text{ end}
\end{align*}
\]

When a job is assigned to a scheduled thread, the job “runs” going through a predefined number of timed steps. At some point, a job finishes and vanishes from the system.

\(f)\) Timing: A number of already defined phenomena require some form of timing. There is no native support for clocks and timers in Event-B but there are a number of established approaches to time modelling. The one we use is closely related to timed automata with integer clocks. To keep track of time progress and time various activities of a system we introduce a number discrete timers. A timer functions like a stopwatch - it counts down to zero from the initial integer value. All the system timers count in synchrony (that is, driven by one global clock) and there is no limit on the number of timers in the system. Once any timer reaches zero, time freezes to allow for a sub-system interested in this timer to react to a deadline; to enable further progress, the sub-system must also reset the timer or delete it. The set of all clocks is defined by function \(\tau:\)

\[
\tau \in \text{TA} \rightarrow \text{TIME}, \quad \text{TIME} = \mathbb{N} \cup \{\text{DISABLED}\}
\]

\(\tau(x)\) gives the current value of clock \(x\) which is \(\geq 0\) before deadline, \(0\) on the deadline and \text{DISABLED} if the clock is not used. To avoid complicated progress arguments, we assume there is a plentiful supply of clocks. At this level of abstraction it suffices to require that set \(\text{TA}\) is infinite.

All the clocks are synchronously updated by an event \textit{time}:
time ⊑ any p1, p2 where
0 ≠ ran(τ)
p1 = τ⁻¹[D, (DISABLED, 0)]
p2 = dom(τ) \ p1
∀ x ∈ status \ (ON) ⇒ temp(x) ≤ CORE_TEMP_CRIT(c)
then
τ := (p1 < τ) ∪ {x ∈ p2 | x → τ(x) − 1}
end

Notice the disabling condition 0 ∈ ran(τ) which stops the
timer until the deadline of a clock is processed. The last guard
prevents clock progress when there is an overheated core: it makes reaction to core overheating immediate.

g) Job deadlines: One application of timing is the
definition of the job deadlines and, to make the concept
meaningful, the notion of job execution time. The latter is
linked to the notion of jobs steps defined above.

A user deadline is set at the point of job allocation as
opposed to the point of job creation. We do not yet model
job queues (it is rather a set in this model) so there is no
fairness property for job allocation. An extended version of
the job_allocate event sets a user deadline for a job.

job_allocate ⊑ any j, t, ta, to_user, udlin where

τ(ja) = DISABLED ∧ τ(to_user) = DISABLED
| ta / to_user ∧ udlin ∈ N
then
job_step_time(j) := ta∥user_deadline(j) := to_user
τ := τ + [F STEP_TIME(affinity(t) → freq(affinity(t)),
to_user → udlin)]
end

The user deadline clock is stored in user_deadline(j); an-
octher clock times the execution of job steps and is stored in
job_step_time(j). The assignment to τ initialises these
two clocks. Note that the step time is defined by the
kind and the frequency of a core on which a thread executing
the job is scheduled. User deadline clock user_deadline(j)
runs down without pauses from the point of job allocation
irrespective of whether a thread processing the job is running
or not.

Job steps are timed according to the respective core per-
formance and are affected by a change of a core frequency.
Should a job fail to meet user deadline, it is cancelled even
before all the job steps are done. Finally, if a job completes
before a user deadline expires, it is accepted as a successful
job execution (event job_finish).

h) Scheduling: As defined above, a core runs several
threads at the same time. In our model, we do not go into the
minute details of scheduling within a group of threads assigned
to a core but rather assume that core indeed runs all the
mapped threads in parallel (that is, the time band in which
the core model is given does not allow us to distinguish between
instances of individual thread executions and the whole picture
is blurred to give an illusion of a multi-threaded core). There
are limits to a number and kind of threads that may be run
on single core before the execution times of individual threads
are affected. Intuitively, if a thread is computationally intensive
and never has to wait for data, it cannot share a core with
another thread without sacrificing performance. In practice,
many applications require data retrieval or do blocking system
calls which make a thread idle and hence free a core to
run another thread. The purpose of the scheduling refinement
step is to bundle threads from various applications into thread
groups that may be assigned to the same core without hindering
thread performance.

Variable thread_load characterises a thread in terms of
lower bound of operations per second (or a comparable nor-
malised measure) necessary to run the thread at full speed.

Thread load must be defined for all the running threads,
dom(affinity) ⊆ dom(thread_load). The crucial property is
that the overall load of threads assigned to a core does not
exceed the core computational capability at the current core
frequency:

∀ x ∈ status \ (ON) ⇒
sum(affinity⁻¹[x]) ≤ CORE_LOAD_MAX(c → freq(c))

Thread load affects the way threads are mapped to cores:

thread_schedule ⊑ any t, c where

... sum(affinity⁻¹[c]) ≤ thread_load(t) +
thread_load(t) ≤ CORE_LOAD_MAX(c → freq(c))
then
affinity(t) := c
end

For a short while, the power budget may be lower than
the power already drawn. To emphasize the immediacy of a
reaction, the system timer is stopped until the power budget
constraint is resolved either by shutting cores or adjusting their
frequency and voltage.

time ⊑ any ... where ∗ ∗ ∗ sum(c.power) ≤ purh then ... end

IV. DSL-KIT

Ability to do design prototyping with the developed formal
domain model was perceived to be one of the more interesting
directions. To this end, we have developed an extension to
the Event-B Rodin [6] modelling framework that adds a
capability of rapidly defining a custom DSL on the basis of
an Event-B specification. The extension, called DSL-KIT, works by combining a general-purpose imperative specifi-
cation language with a custom set of 'commands' defined by an
Event-B specification. The imperative layer provides control
flow construct that allow one to write scenarios or imperative
programs in the terms of Event-B events. From the viewpoint
of Event-B, the imperative layer is an extra refinement step
strengthening event guards and declaring new hidden state.

The core of DSL-KIT is a formal specification language
based on the following principal structuring units:

• system - the top-level unit defined as a parallel com-
position of several actors;

• module - a self-contained unit providing definitions of
actors and actions;

• actor - a unit of concurrency; its body is sequential
code guarded by rely/guarantee conditions;

• action - a function-like entity;
actor defs($c$ : CORES) rely ... 

\{  
if (status($c$) = OFF) {  
  core_threads : set(THREADS) = affinity$^{-1}(\{c\});  
  ml : int = CORE_LOAD_MAX($c$, freq($c$));  
  if (core_threads = $\{\}$) {  
    total_load : int = sum(thread_load[core_threads]);  
    if (ml < total_load + STEP) {  
      if (enabled core_defs_running(...)) {  
        core_defs_running($c$, vdd($c$) + CH, freq($c$) + CH);  
      } else if (ml > total_load + STEP and ml > MIN_LOAD) {  
        if (enabled core_defs_running(...)) {  
          core_defs_running($c$, vdd($c$) − CH, freq($c$) − CH);  
        } else {  
          # nothing is running, run down frequency gradually  
          if (enabled core_defs(...)$−1$ and ml > MIN_LOAD) {  
            core_defs($c$, vdd($c$) − 1, freq($c$) − 1);  
          }  
        }  
      }  
    }  
  }  
\}  
}\}  

Fig. 2. Cores-DSL script for a simple on-demand frequency/voltage governor.

- control flow statements - sequential composition, if, while, for and auxiliary variable declarations;
- expressions and predicates, expressed in the Event-B mathematical language; this makes possible to relate DSL-Kit to Event-B without logic mapping.

Not all scenarios defined on top of the Event-B domain model correspond to behaviour permitted by the domain model. For instance, the following defines a scenario made of a sequence of two events:

\[
\text{on}(c, v, f); \text{off}(c); 
\]

The first event switches a core on and the second switched off the same core (assuming all identifiers are locally bound). If, instead, we write

\[
\text{on}(c, v, f); \text{on}(c, v, f); 
\]

it appears that the second event cannot be enabled and the scenario must halt prematurely. In fact, even in the first example, we could not know that \text{on}(c, v, f) starts in state where the guard of even \text{on} instantiated with arguments $c, v, f$ is enabled. To collect all such conditions automatically, we need to know the kind of states in which an event (or, for generality, any imperative statement) is enabled and also the kind of states it produces upon termination.

The module containing the Cores domain model is made available for new actor and system definitions. Module variables (translation of Event-B variables) are accessible read-only only. Their state may only be modified via module actions. Event-B invariants thus become module axioms (Event-B has inv and progress problems for safety and correctness are proven statically) was accomplished by building design prototypes in the developed DSL. We have done 15 revision cycles over the course of five months since the initial Event-B domain model was developed.

V. Conclusions

We have presented a formal DSL for multi-core systems. It is a completely proven Event-B specification with 9 refinement steps and over 600 hundred proof obligations. Much of the final detailisation and 'debugging' (that is, fixing omissions and progress problems for safety and correctness are proven statically) was accomplished by building design prototypes in the developed DSL. We have done 15 revision cycles over the course of five months since the initial Event-B domain model was developed.

Acknowledgement

This work is supported by EPSRC grant EP/K034448/1.

References