Simulation Modeling of a Large-Scale Formal Verification Process

He Zhang, Gerwin Klein, Mark Staples, June Andronick, Liming Zhu, and Rafal Kolanski
NICTA, Australia
University of New South Wales, Australia
{*firstname.lastname*}*@nicta.com.au*

***Abstract*—The L4.verified project successfully completed a large-scale machine-checked formal verification at the code level of the functional correctness of the seL4 operating system microkernel. The project applied a middle-out process, which is significantly different from conventional software development processes. This paper reports a simulation model of this process; it is the first simulation model of a formal verification process. The model aims to support further understanding and investigation of the dynamic characteristics of the process and to support planning and optimization of future process enactment. We based the simulation model on a descriptive process model and information from project logs, meeting notes, and version control data over the project's history. Simulation results from the initial version of the model show the impact of complex coupling among the activities and artifacts, and frequent parallel as well as iterative work during execution. We examine some possible improvements on the formal verification process in light of the simulation results.**

***Keywords*-software process modeling; process simulation; formal verification; system dynamics; microkernel**

I. Introduction

Formal software verification is the verification method that provides the strongest known assurance that a software system implementation is consistent with its specification. Formal verification does not merely check all lines of code or all decisions in a program, but all possible behaviors for all possible inputs.

More commonly applied verification methods are testing and code inspection. While they provide high return for lower assurance levels, they do not scale well to providing high assurance and become prohibitively expensive for the assurance level that formal verification can provide. While formal verification is cheaper for high assurance than testing, it still is a high-effort verification method and currently only feasible to apply for life- or mission-critical software systems. Most previous industrial use of formal methods has only performed formal specification, rarely formal verification [1], and if the latter, then often only for lightweight properties, not for a full proof of implementation correctness [2].

The recent formal verification of the seL4 (secure embedded L4) microkernel [3] has demonstrated that this method does scale to industrially relevant software systems and sizes on the order of 10,000 lines of C code. seL4 is part of the L4 family of high-performance operating system (OS) microkernels [4]. The L4.verified project has performed formal verification not only at the design level, but down to C source code; and not only for lightweight properties, but for the full functional correctness of a highly complex software system—the seL4 microkernel. While a microkernel is a comparatively small software system, its verification with an overall effort of 25 person years was a large-scale research project.

Because of its relatively high effort and up to now infrequent use in practice, we believe it important to analyze and investigate the process of formal verification and how it influences the rest of the development process. The process used in the L4.verified project was significant in enabling its success. In earlier work [5], we reported on a detailed, descriptive model of the verification process used in L4.verified that was validated by project data and experience. A qualitative finding was that the middle-out approach provides advantages over pure top-down and bottom-up processes for formal methods. However, a descriptive process model neither reflects the dynamic behavior of the process, nor does it provide predictive power for supporting detailed project planning and execution. As there is little empirical evidence about formal verification processes, we decided to employ a simulation model to further investigate them, based on our experience with L4.verified. This paper builds on our previous work [5], and reports a new process simulation study of the L4.verified project. The objective of this research is to contribute to better understanding of how large formal methods projects can be run successfully. Simulation results will inform process improvements for overall project performance and process adaptability.

To our knowledge, in terms of the earlier systematic surveys [6], [7] and more recent observations, this is the first process simulation model of a formal verification process. Based on a tailored descriptive process model of L4.verified, in this paper we report our work that: *1*) developed a continuous process simulation model—VPMsim 1.0; *2*) approximately calibrated parts of the simulation model with the data from L4.verified's project repository and team leaders' recollections; and *3*) investigated the possible impacts of process decisions or changes to this project.

The paper is structured as follows. We first provide a brief overview of the L4.verified project and seL4 microkernel in Sect. II. Sect. III describes the middle-out verification

978-1-4673-2352-9/12/$31.00 © 2012 IEEE

ICSSP 2012, Zurich, Switzerland

process of L4.verified using a conceptual model and a descriptive model. In Sect. IV, we elaborate the simulation model (VPMsim) of the verification process and discuss its calibration with L4.verified project data. Sect. V shows how the simulation model can be applied to investigate formal verification processes. We discuss experiences with and limitations of the current work, before concluding and identifying future research.

II. Background

Software verification can be accomplished by any of several means or their combination. Common verification methods are test, review, and analysis, which can be performed manually or automatically. A number of process simulation models studied these verification methods [6], [7]. Formal methods are another verification option that is able to prove correctness of software with respect to mathematically-specified requirements. It has not yet been investigated using process simulation.

A. L4.Verified Project

The L4.verified project completed the implementation and formal verification of the seL4 microkernel. A kernel is the part of the OS that runs in the privileged mode of the hardware. It has direct access to all hardware resources and provides the basic mechanisms for implementing the rest of the system. A microkernel, as opposed to more common monolithic OS kernels, is reduced to the bare minimum of functionality and code. The seL4 kernel comprises 8,700 lines of C code and 600 lines of assembler (without counting blank lines and comments). This radical reduction in size comes with a price in complexity. It results in high coupling and a high degree of interdependency between different parts of the kernel, as apparent in the function call graph of seL4 in Fig. 1. The motivation for the radical reduction in size and for formally proving functional correctness, is to provide high levels assurance for safety, security, or correct functionality of systems built on top of seL4. Formal verification gives the highest degree of assurance we can provide [8]. The small size of seL4 reduces the amount of critical code that must be formally verified.

The L4.verified project ran over 4 years from 2005 to 2009. It involved two teams: OS kernel developers and formal methods practitioners. A previous paper [5] has reported a number of general lessons about the management and execution of the project.

B. Related Work in Process Simulation

Systematic literature surveys [6], [7] show that software verification, e.g., inspection and testing, is one of the five most common topics in process simulation. Nonetheless, the model reported in this paper is the first reported simulation model for a formal verification process. This subsection reviews process simulation models that focus on non-formal verification and validation in software development.

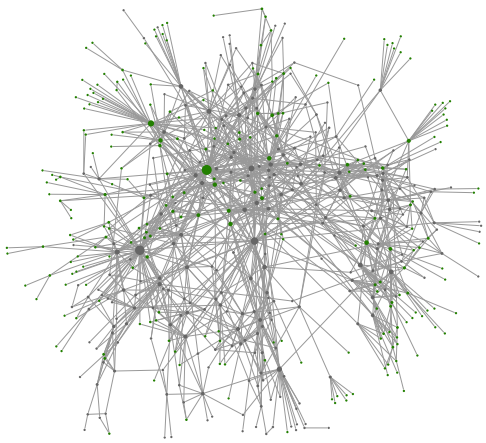


Figure 1. Function call graph of the seL4 kernel

Software verification ensures product quality, and has been investigated by different process simulation techniques in varying organizational settings. Raffo et al. [9], [10] modeled V&V (verification and validation) as a portion of the traditional V-model style development process (i.e. ISO 12207) adopted on NASA's software development projects. They created a discrete-event simulator to quantitatively assess the economic benefits of performing V&V activities on development projects and to optimize that benefit across alternative V&V integration strategies. This enabled NASA to more effectively allocate scarce resources for V&V activities.

GENSIM 2.0 [11] is a System Dynamics based process simulator that models and simulates a generic development-verification (D-V) process. The GENSIM 2.0 model is constructed with three levels of refinement and their validation counterparts: requirements D-V, design D-V, and code D-V. A variety of verification activities could be adopted on each level. GENSIM 2.0 is able to assess the overall effectiveness and performance (e.g., product quality, project duration and effort/cost) of varying combinations of different development, verification, and validation strategies and techniques depending on the inputs of 28 parameters.

III. A Middle-out Formal Verification Process

This section describes the middle-out formal verification process of the L4.verified project using a high-level conceptual model and a detailed descriptive model.

A. Conceptual Process Model

The goal of the L4.verified project was to develop and formally verify a high-performance kernel. It is a challenge to design a formally verifiable kernel while maintaining high performance. To obtain high performance, kernel developers

usually take a *bottom-up* approach to design, focusing on low-level details that allow efficient management of hardware. In contrast, formal methods practitioners often prefer a *top-down* approach based on simple models with a high level of abstraction.

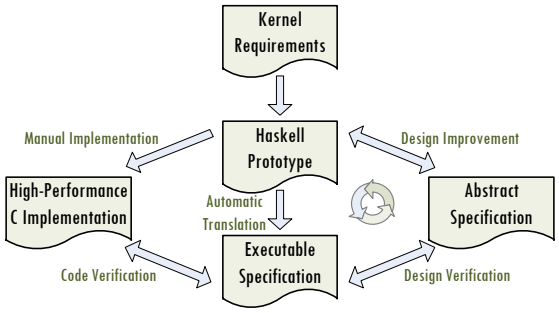


Figure 2. Conceptual process model of the middle-out verification

To achieve both objectives, the L4.verified project bridged the gap between verifiability and performance using an iterative, concurrent prototype-base, middle-out process shown in Fig. 2. This is significantly different from the conventional pure *bottom-up* and *top-down* approaches. It is based around an intermediate target that is used and understood by both the kernel developers and the formal methods practitioners, with the aim of rapidly iterating through design, specifications, implementation and formal model until convergence. This intermediate target is a *prototype* of the kernel written in the functional language Haskell (in the middle of Fig. 2). It is translated automatically into the *executable specification* of the kernel in the theorem prover Isabelle/HOL [12]. The importance of the use of executable specifications in formal verification in a theorem prover has been recognized previously [13].

The *abstract specification*, on the right of Fig. 2, is a formal description of the functionality of the kernel. It specifies the outer interface and effects of each system call, but does not describe in detail how these effects are implemented. In other words it describes *what* is expected from the kernel, whereas the *executable specification* describes *how* the kernel will achieve its purpose. In that sense the *executable specification* represents the *design* of the kernel. The proof that the *executable specification* refines the *abstract specification* was carried out first. This proof can be seen as *design verification*.

On the left of the conceptual model (Fig. 2), a low-level *high-performance implementation* of the kernel was manually written in the C language. The second proof shows that the source code correctly implements the *executable specification*, which we will also refer to as *code verification*. Note that the C code is translated directly and automatically into the theorem prover for verification [14].

B. Descriptive Process Model

The formal verification of seL4, in combination with the development of the kernel itself, did not follow a conventional software engineering process reported in the literature. Instead the project followed the implicit conceptual process described above. In earlier work [5], we reported on a postmortem analysis of the process applied in this project and formulated a detailed, descriptive process model that shows process patterns and potential process factors for reuse and scaling of formal verification in software and systems development. In this subsection, the descriptive process model [5] is tailored for our initial simulation model, by eliminating the maintenance phase. This was done to simplify creation and calibration of this initial simulation model.

As shown in the conceptual model in Fig. 2, the L4.verified project used a middle-out approach, starting with an executable specification, which was then proved to be consistent with a high-level abstract specification, and later with the low-level source code. Fig. 3 shows the tailored descriptive process model of L4.verified. Each activity in the model is directly linked to its input and output artifacts, and annotated with the performers (OS or FM team), type (manual, automatic, or interactive) and its step number. Note that the terms *activity* and *step* are used interchangeably in this paper. We do not include activities on proof tools and libraries in this version. The formal verification activities are *technical development processes* [15] modeled between the three levels of abstraction.

The steps `S1` to `S6` in Fig. 3 roughly correspond to the transformations between artifacts in the conceptual model. While the main differences between the conceptual model and the descriptive representation of the process in Fig. 3 are the detailed artifact- and work-flow and the explicit decision points being modeled.

The *initial kernel requirements* (and *new feature & change requests*) on the top left in Fig. 3 are fed into the first step `S1` – *prototype development*. The dashed line denotes the exogenous artifacts that come from outside the process rather than artifacts being generated during the process. The output *prototype* (Haskell) is automatically translated to be *executable specification* at `S2`. `S3` defines the *abstract specification* based on the *prototype*. When `S2` and `S3` become stable, the refinement between these two specifications is proved in `S4`. The defects (inconsistencies between the two specifications) detected in `S4` are returned to `S1` and `S3` separately for rework. On the code level, when the *prototype* becomes stable (i.e. `S4` has gone through its first major iteration), `S5` is triggered to manually implement the kernel in C. Later in `S6`, the source code is verified against the design (*executable specification*). Because the design becomes mature after `S4`, code-level defects are usually fixed directly in `S5` and `S6`. In rare cases they are

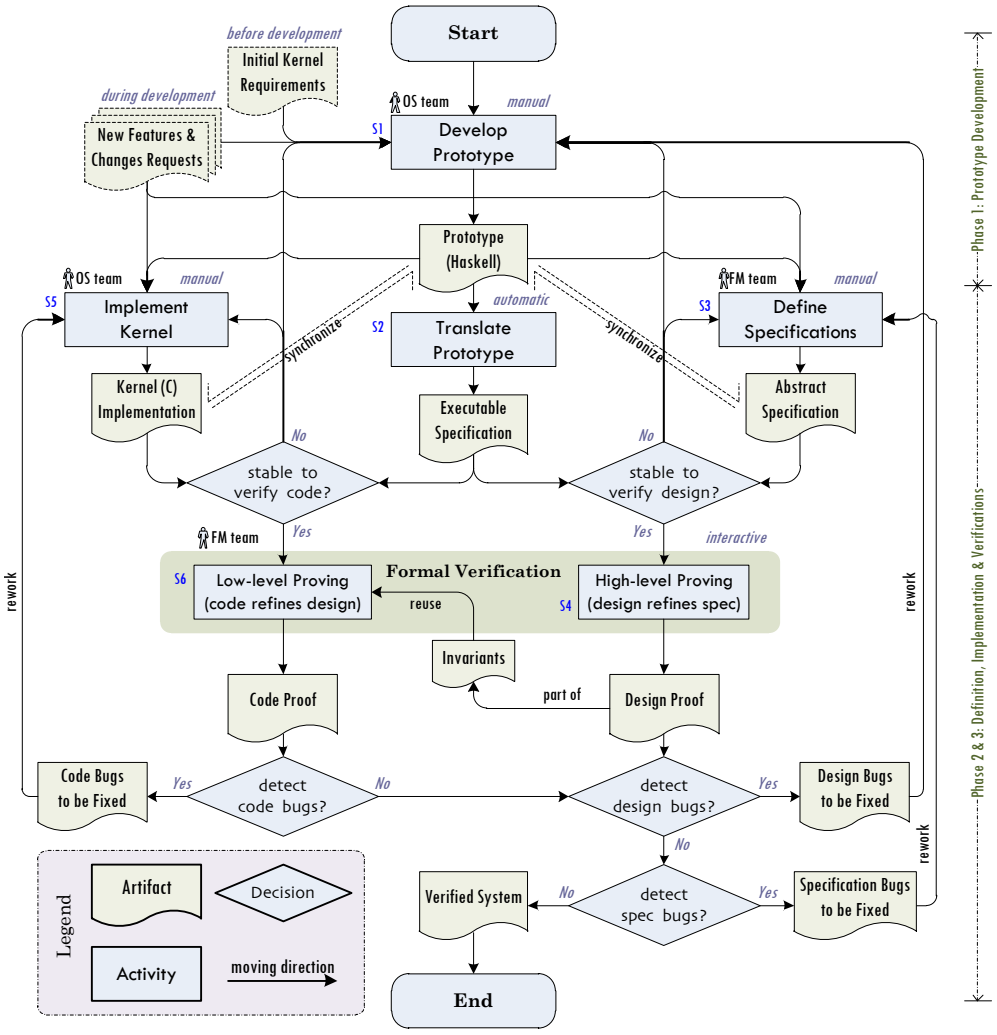


Figure 3. Descriptive model of L4.verified project

escalated to the design level (S1) or even up to the abstract specification level (S3).

In the real project, this process experienced multiple iterations through steps S1–S6. They were triggered by feature changes in the prototype as well as by defects discovered during either verification phase, for example the loop S1-S3-S4-S1-S2-S4-...

An interesting artifact in formal verification is the set of *invariants*. In the middle-out process, the *invariants* are mostly proved as part of the design verification (S4). These *invariants* are reused heavily in the code verification, which helps to reduce the workload in step S6. Though theoretically S4 and S6 could be performed in parallel, significant savings in low-level (code) verification were possible because the invariants from S4 had stabilized. In terms of the experience from L4.verified, starting S6 too early may negate this effect. Note that S6 may also induce additional invariants to be proved on the *executable specification*. Invariant proofs are the highest-effort parts of this verification.

More generally, the descriptive process model identifies three main phases of the project annotated on the right-hand side of the diagram: *1*) *prototype development*, which clearly appears in the beginning; *2*) specification *definition* and design *verification*, an iterative process on the right of the diagram; and *3*) kernel *implementation* and code *verification*, another iterative process on the left of the diagram. The entire process in Fig. 3 terminated when there were neither bugs remaining or being reported in all artifacts, nor new features or change requests coming into the process.

IV. The VPMSim Simulation Model

In order to understand and investigate the dynamic behaviors of the middle-out formal verification process, we

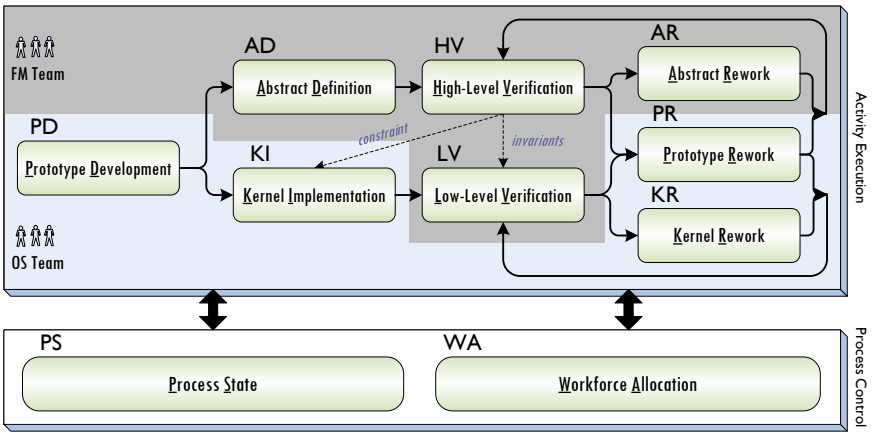


Figure 4. Model structure of VPMsim 1.0

developed a process simulation model based upon the descriptive model (Fig. 3). This section elaborates the initial version of the simulation model—VPMsim 1.0.

A. Modeling Scope and Approach

There is a difference in modeling scope between the earlier descriptive process model [5] and the simulation model in this study. In the complete descriptive process model, after the release of the *verified system* (cf. Fig. 3), the L4.verified project progressed into the *maintenance* phase. This is out of the scope for the simulation modeling reported in this paper. For this stage, the simulation model focuses on the phases from the beginning until the first release of the project, i.e. excluding the maintenance phase.

Our research focus for this first version of VPMsim is on the direct activities of the production and verification of the artifacts in Fig. 3. This means we did *not*: *1*) model the development of supporting tools for translation and theorem-prover; *2*) explicitly model the conventional verification, e.g., unit test, whose effects are reflected when calibrating the development rates of *prototype development* and *kernel implementation*; *3*) model the activities that did not contribute to development and verification based on requirements, e.g., documentation and code cleanup.

We have used a continuous process simulation approach, System Dynamics (SD), which allows less micro-process level data than would be required for discrete simulation.

B. Model Structure and Execution

VPMsim 1.0 was developed using Vensim, the most commonly used SD modeling and simulation package in software process research over the past decade [7]. Vensim provides a graphical workbench and a number of extra features on the top of SD, e.g., *views* and *subscripts*. The VPMsim 1.0 model comprises ten views, 180+ parameters, including auxiliary ones, and over 2000 lines of code.

Fig. 4 shows the high-level structure of the simulation model that is based on the descriptive model of the middle-out process (Fig. 3). The development and formal verification activities of the middle-out process are modeled and organized by *views*, a mechanism offered by Vensim that facilitates development and understanding of large scale SD models. It also increases module reuse within a complex model. The current version (1.0) model is composed of ten *views*, eight of which correspond to the specific activities (steps) of the formal verification process modeled in Fig. 3. As shown in Fig. 4, they are ***prototype development*** (`PD`), ***abstract definition*** (`AD`), ***kernel implementation*** (`KI`), ***high-level verification*** (`HV`), ***low-level verification*** (`LV`), ***prototype rework*** (`PR`), ***abstract rework*** (`AR`), and ***kernel rework*** (`KR`). Note that step `S2` (Fig. 3) is not modeled as a view in Fig. 4 since this step can be automated. Another important change from the descriptive model is that rework views of abstract, prototype and kernel are created apart from their development views, because differences between the two types of view are significant, e.g., inputs, productivity, workforce and process control. On the bottom of Fig. 4, there are the other two views—***process state*** (`PS`), which defines important variables shared by the other views of activities and exogenous variables (e.g., *requirements generation*), and ***workforce allocation*** (`WA`) that implements dynamic workforce allocation among the activities in parallel. In addition to *view*, *subscripting*, a mechanism provided by Vensim that enables the variables holding different values for multiple entities simultaneously, is also used in building VPMsim 1.0, but for modeling defect severity levels only (i.e. *high*, *medium* and *low*).

Due to limited space, we do not show all views of the SD model. We only describe the relationships and constraints among the views in simulation. The above eight activities (views) are performed by the FM and OS teams respectively. This is denoted by the darker and lighter gray background of

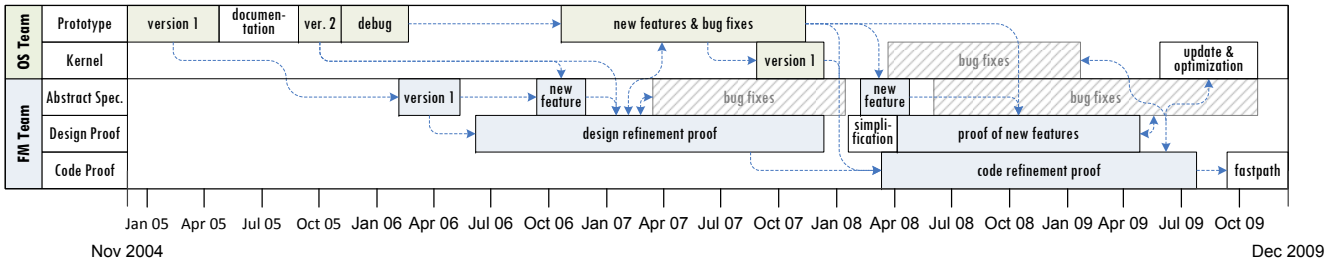


Figure 5. Timeline of the main artifacts developed in sequence

the views in Fig. 4. The simulation starts with the PD view by the OS team. (The automated translation of *prototype* is absorbed in the PD view in the simulation model.) PD is followed by AD (*abstract spec.* by the FM team) and KI (*kernel* by the OS team). Note that in the real project and in the simulation, the stabilization of PD may always trigger AD immediately for an early HV. The lower-level KI waits until a majority of HV completes. The question of when to kick off KI depends on a number of factors (e.g., the progress of HV), and is one of the *what if* questions to be explored in simulation (cf. Sect. V). When the first version of the *abstract spec.* is completed, it is proved in HV against the *prototype* (design) from PD. During HV, any design defects and abstract (specification) defects detected are fed into PR and AR correspondingly for rework. The corrected prototype and abstract are re-verified in HV.

In the L4.verified project, kernel programming started when a large portion of the design proof was completed. In the simulation model, the implemented kernel (from KI) is verified against the *prototype* in LV. Any detected code-level defects are fixed in KR. Only a small number of design defects have to go into PR or AR for rework. All of these corrections may lead to re-verification at LV or even at HV. During simulation, requests for new features and changes are generated and introduced into the process from the left (PD in Fig. 4). The resulting updates to *prototype*, *abstract spec.*, and *kernel* also have to be re-verified.

C. Model Parameters and Calibration

The VPMsim model consists of a large number of parameters that represent inputs, outputs, and policies or constraints at the activity (*view*) level and overall process level. Many other parameters have to be calibrated against empirical data for specific projects, teams, artifacts, and techniques. Table I lists a subset of the model parameters as examples. Note that many of the parameters may vary over time and between specific activities, artifacts, teams, and iterations.

The development and verification of the seL4 kernel from prototyping through implementation, including all formal models and proofs, has been managed using version control systems. Around 9,000 changesets provide detailed information about the evolution of artifacts over the full lifetime of the project, including the ongoing maintenance phase. Each changeset identifies who made the change, of which artifact, at what time, and the size change of the artifact. Analysis of these changesets give us estimates of the growth of artifacts and the workforce allocation over its project period. For other information for model calibration, such as for invariants, defect distribution, and policy issues, we made relatively coarse estimates based on the team leaders' recollections. More detail about how the data was retrieved from the repositories can be found in earlier work [5]. Some *calibrated* parameters are marked in Table I.

By analyzing graphical representations of the repository data (cf. [5]) combined with explanatory project logs (comments), we constructed Fig. 5. It shows the real progress on each of the five major artifacts during the project duration (without maintenance phase) in swim lanes. On the leftmost of Fig. 5 the five artifacts are grouped by their development/verification teams. The intermediate states of each artifact (denoted in rectangles) are positioned in the diagram in terms of their artifact type (lane) and occurring time along with the project's timeline. We marked the critical states for model calibration in gray. The shadowed states (*bug fixes*) are also important, but cannot easily be distinguished within the repository data. The dashed-line arrows indicate the sequential order and dependencies across the artifacts' states. The timeline diagram clearly shows how the different artifacts' states overlapped, and how dependencies and iterations happened in the project.

V. Model Use and Results

As the L4.verified project is the first instance of the middle-out formal verification process and VPMsim 1.0 was calibrated with the data of this project only, the proper application of this model is to re-investigate this project. In this section, we first validate the model by simulating the original L4.verified project as a baseline, then use it to examine some possible changes to the project.

A. Simulation Baseline

We first defined the input parameters as close as possible to the real L4.verified project, and then ran the simulation model. According to the project timeline Fig. 5, there were

TABLE I
A SUBSET OF VPMSIM 1.0 MODEL PARAMETERS

Parameter	Attribute of	Type	Value (range)	View(s)	Team(s)
Requirements generation rate	Project	Input		PS	n/a
Overall team size	People	Input	1-10 (*3-7 for FM, 1-4 for OS*)	WA	FM/OS
Workforce turnover rate	People	Input		WA	FM/OS
Bug-fix threshold for rework	Project	Input	1-3	HV/LV	FM/OS
Nominal developer's productivity	People	Calibrated	vary over time, teams and artifacts	PD/AD/KI/HV/LV/AR/PR/KR	FM/OS
Artifact conversion ratio	Product	Calibrated	vary over time and between artifacts	PD/AD/KI/HV/LV	FM/OS
Bug density	Product	Calibrated/Subscript	vary among artifacts	PD/AD/KI	FM/OS
Verification effectiveness	Process	Calibrated/Subscript	0.7-1.0	HV/LV	FM
Artifact size adjustment factor by bug-fix	Product	Calibrated/Subscript	0-2000 (mean)	PD/AD/KI/AR/PR/KR	FM/OS
Size of completed artifact	Product	Output		PD/AD/KI/HV/LV	FM/OS
Number of bugs detected by verification	Product	Output		HV/LV	FM
Number of residual defects	Product	Output		PD/AD/KI/HV/LV/AR/PR/KR	FM/OS
Team utility	People	Output		PD/AD/KI/HV/LV/AR/PR/KR	FM/OS
Total effort	Project	Output		PS	FM/OS

three major versions of the prototype, which corresponded to the evolved requirements (new features+change requests) during the project. The simulation generates the exact same amount of requirements changes at these three time points. Also the *kernel implementation* and *abstract definition* are triggered as shown in Fig. 5. Another important input is the workforce turnover. The baseline model simulates the personnel's entries and exits in both teams as recorded in the project repository. Due to space limitations, Fig. 6 only shows the changes of some important output parameters generated in this run.

In the simulation (Fig. 6-d), the baseline project completes on Dec 25, 2008 with a total effort of 5,400 person-days (roughly 15 person-years), which conforms to the project teams' experience (14 person-years for kernel-specific verification) [5]. The effort on formal verification related work (done by the FM team) takes about 80% of the total effort (4400 person-days), in particular nearly half on the design refinement proof (approximately 2400 person-days). The frequent ups and downs shown in Fig. 6-c indicate the intensive personnel switches between parallel activities. Note that the simulated baseline project finished earlier than the real project. The length of the gap is almost one calendar year. It is caused by the following possible reasons: *1*) the VPMsim 1.0 models development and verification, but not other activities such as documentation and code cleanup; *2*) the simulation only handles weekdays and weekends, but the public holidays and team members' (annual and sick) leave are not calculated; *3*) the workload on development related to the theorem prover is not considered in this model. However, by comparing the trends and quantitative measures of the scales of main output parameters (e.g., sizes of artifacts in Fig. 6-a and -b) between the baseline simulation and the original project, the simulation results are close to the reality and most noticeable differences can be reasonably explained. Hence, based on the model calibration and the baseline simulation, we consider VPMsim 1.0 acceptable for further process investigation of L4.verified.

B. Model Application Scenarios

One important characteristic of the L4.verified project is the parallel work on development and verification. Fig. 5 shows a number of parallel activities in the project, in particular the second half of the project. Note that some activities started late in the diagram with a big gap to their predecessors. The real L4.verified project had particular reasons for this, but in other hypothetical projects following the middle-out process these reason may not be present. In a new project, these steps could potentially start earlier in the process, in parallel with other activities, to optimize the overall process performance. For example, *1*) the abstract definition, *2*) the manual implementation of the kernel in C, *3*) the development of new features introduced in prototype (after ver. 2). Accordingly, we chose 6 months as an observation period and investigated the impact of three possible change scenarios to the baseline project using simulation:

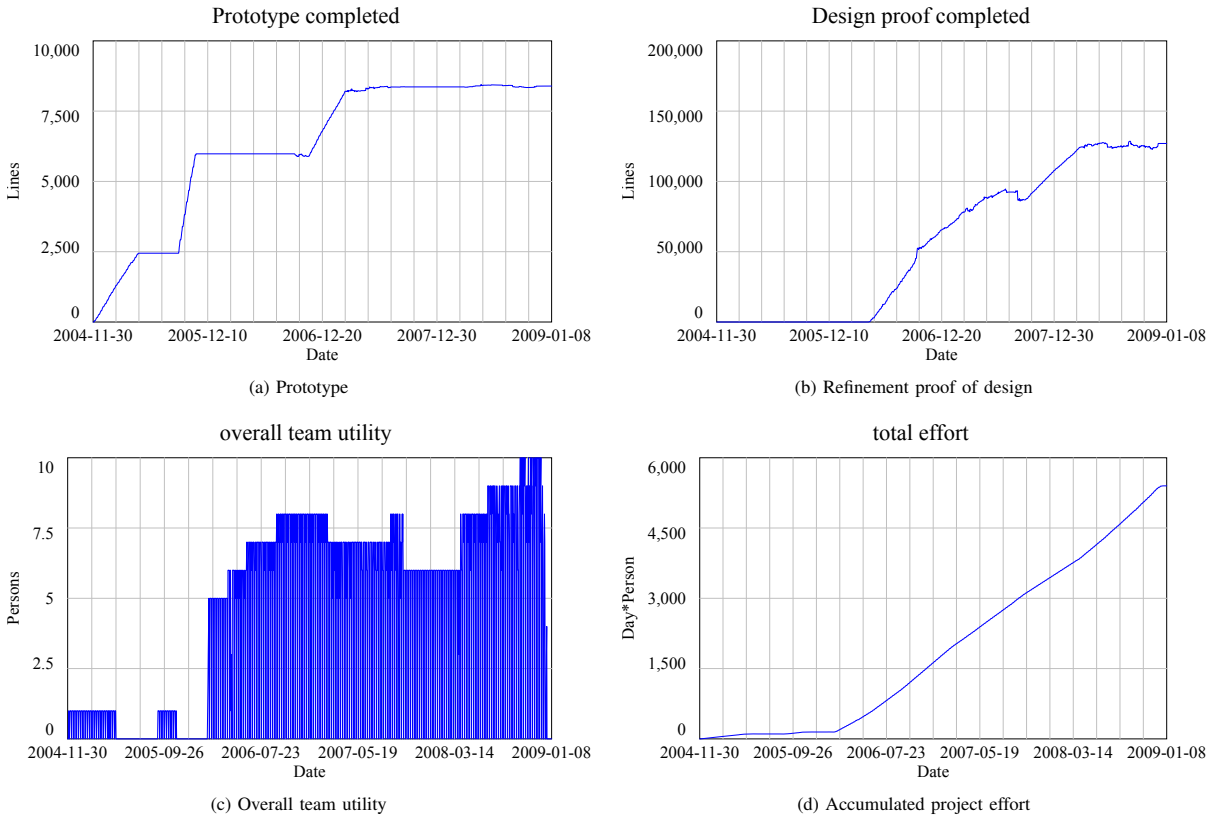
- S1 : start the *abstract definition* six months earlier than the baseline;
- S2 : enable the *kernel implementation* six months earlier than the baseline;
- S3 : introduce new features to the last major version (ver. 3) of *prototype development* six months earlier than the baseline.

Of these scenarios S2 would have been feasible in the real L4.verified project, S1 and S3 not, at least not easily.

C. Simulation Results

The simulation ran in terms of the process changes suggested above without changing any other model parameters. The results of some output parameters are shown in Fig. 7 in comparison with the baseline. Based on the simulation results, we discuss the possible impacts of the proposed process changes on overall project performance, particularly effort and duration.

S1: Fig. 5 shows the *abstract definition* started about 14 months later than the kickoff of the project. Theoretically, this step can start once the initial version of prototype is stable. For triggering it six months earlier the simulation



(a) Prototype

(b) Refinement proof of design

(c) Overall team utility

(d) Accumulated project effort

Figure 6. Simulation baseline of the L4.verified project

predicts savings on both project cost (effort) and duration. The scenario S1 finishes on May 1, 2008 with a cost of 4,437 person-days. This suggested change may result in a nearly seven-month advance of the overall project. This possible improvement can be attributed to the early involvement of the FM team, which allows an earlier high-level verification on design (shown as Fig. 7-c). As two main rounds of design proof complete earlier, it further reduces the later parallel verification work on both levels and increases the verification rate for the code proof.

S2: For moving the *kernel implementation* six months forward, the simulation predicts a delay of the overall project for five months, as well as an effort increase of about 800 person-days compared to the baseline. By looking into the simulation, we found that at the suggested time, the *kernel implementation* starts when the prototype is not mature and stable yet (in parallel with ver. 3 development and rework), and so it incurs a number of additional code defects that cannot be fixed immediately (a flat defect level in Fig. 7-b). Meanwhile, since the OS team has fewer members than FM team, frequent switches between prototype and kernel may lower the team's productivity. As a result, the extended completion of prototype and kernel further delays the verification on two levels.

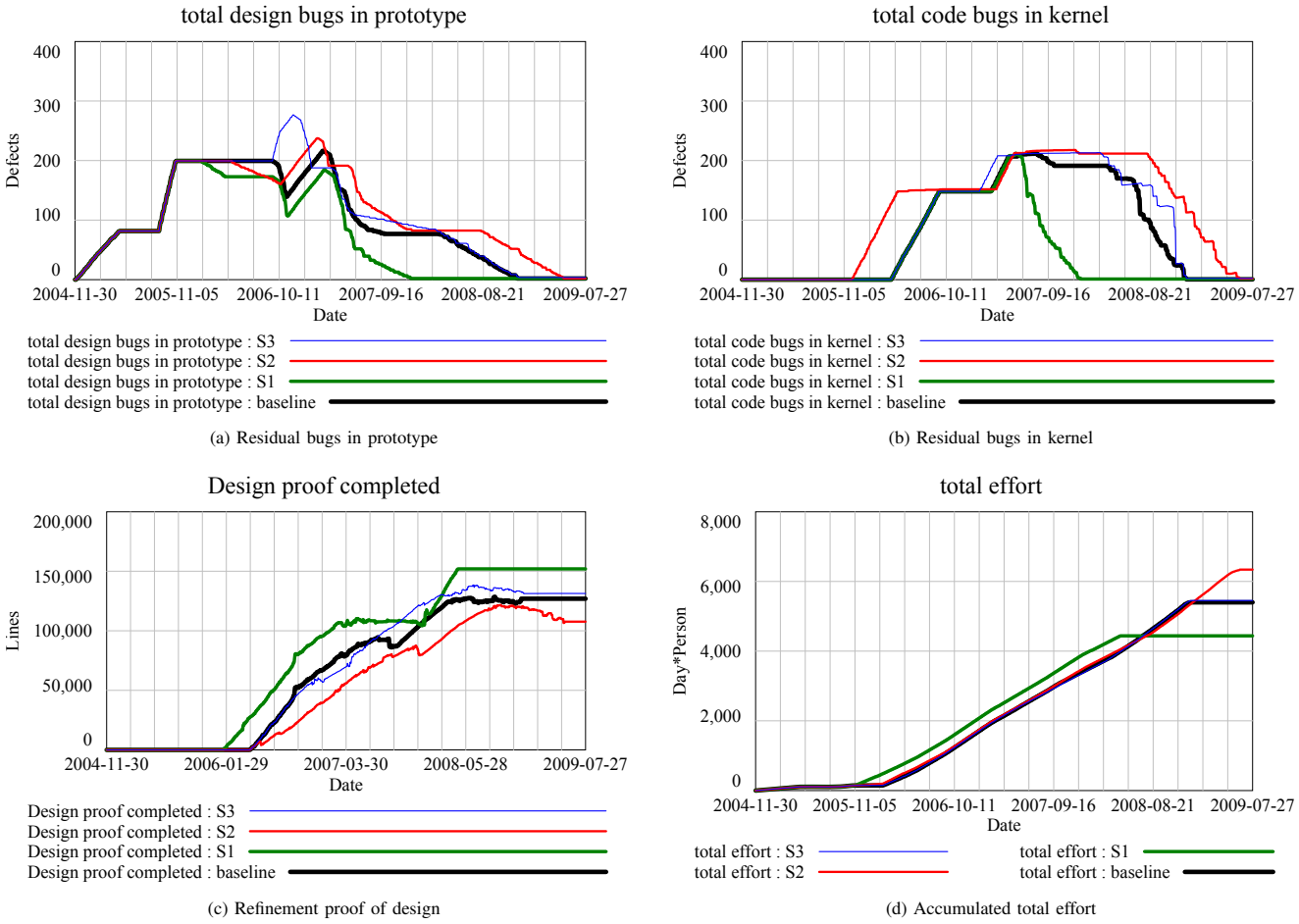
S3: In Fig. 5, the third group of requirements (new features) were introduced to prototype nine months after the version 2. When introducing these new features six months earlier, the simulated project effort and duration remain almost unchanged compared to the baseline. Although the OS team develops version 3 of the prototype earlier, bug fixing also relies on the progress on the design refinement proof (cf. the accumulated design bug level in Fig. 7-a). In addition, as the *kernel implementation* starts at the same time as in the baseline, there are no improvements on the performance of the code proof.

Each of the above scenarios only suggests process change on one parameter: the start date of one step. The simulation also supports testing combinations of other possible process changes, but this is beyond the scope of this paper.

VI. Discussion

A. Experience

The simulation model, VPMsim 1.0, reflects the characteristics of the formal verification process in model structure and simulation results: *1*) concurrent development/verification activities, *2*) frequent iterations and re-verifications, *3*) dynamic and concurrent resource (workforce) allocation, and *4*) the effect of invariants in code verification.



(a) Residual bugs in prototype

(b) Residual bugs in kernel

(c) Refinement proof of design

(d) Accumulated total effort

Figure 7. Simulated possible scenarios of the project

When we started to develop the simulation model, we tried to apply more customizable *process patterns* (such as [11]) and the use of *subscripting* to simplify the model structure and maximize the reuse of model components across activities (*views*). However, as shown in Fig. 3, each activity has different types of input and output artifacts and complex control flows with other activities, which resulted in complicated control logic behind the model components. This explosion in complexity further complicated model debugging. In response to this, we restructured the model with eight dedicated *views* for the activities. This allows a more straight-forward modeling and debugging of activity-specific characteristics, but sacrifices some component reusability.

Fig. 6-c reflects frequent staff switches between concurrent activities in the project life-cycle. In the extreme case, one developer may work on three different artifacts simultaneously and quickly switch between them. Due to the inherent limitation of continuous modeling, this phenomenon was seldom modeled and reported in the literature of SD based process simulation. In order to correctly model the dynamic workforce allocation between the parallel work, we developed a large and complex model component (`WA` view in Fig. 4). This component can be reused in modeling other concurrency-intensive processes in the SD approach.

B. Limitations

The metrics used in VPMsim 1.0, such as lines of proof and numbers of invariants, are relatively coarse measures, used for simulation study. They sometimes do not realistically capture and reflect the essence of size and progress in a formal verification process. For example, because the invariants are all connected, isolating their individual effects in verification is almost impossible. The investigation and development of appropriate metrics for a formal verification process is beyond the scope of this paper, and requires more theoretical research and empirical evidence from the practical application of formal verification.

Another limitation is the precision of the data for model calibration. The details about size of change of artifacts and workforce allocation are based on version control. However, the data set only reflects each artifact's commit times, not the effort spent on the artifact. In particular, if a person worked

on multiple activities simultaneously, a precise estimate of their effort allocation to different activities is hard to achieve.

The VPMsim 1.0 is implemented using System Dynamics. We found it is difficult to model such a complex process at a fine-grained level using continuous simulation. For instance, continuous simulation merely allows the tracking of process entities on an average level, e.g., feature, defect, invariant and developer, we cannot assign properties to each of them and trace their change individually. Though we can use mechanisms such as subscripting in Vensim to setup finer categories, the help this provides for modeling precision is still limited. Discrete simulation is more suitable for handling an individual entity's movement through a process, especially in iterative and parallel styles, and may result in a higher precision with more details for analysis.

VII. Conclusions and Future Work

The L4.verified formal verification project succeed in large part due to the middle-out process used, together with other formal and technical innovations. Based on the descriptive process model formulated in our previous study, we developed a large scale process simulation model—VPMsim 1.0 to further investigate this unique process. This paper reports the model and simulation results after the initial calibration. Specifically, we *1*) developed the first instance of continuous simulation model of a formal verification process; *2*) calibrated the model with the data from a real, large-scale project; *3*) show the potential value of a process simulator in support of formal verification in practice; *4*) report our experience in modeling and simulating a formal methods project.

The initial results and experience of this research on formal verification process offer a number of suggestions for future work, such as *1*) converting VPMsim 1.0 to a discrete-event (or hybrid) simulation model that allows detailed modeling and tracking of the entity flow; *2*) extending the scope of VPMsim to cover the maintenance phase of the L4.verified project and support for future decision making; *3*) applying the simulation model in other formal methods projects and enhance its adaptability.

Acknowledgment

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Computing Surveys*, vol. 41, pp. 19:1–19:36, Oct. 2009.

[2] K. L. McMillan, "Fitting formal methods into the design cycle," in *31st Design Automation Conference*, ser. DAC '94. New York, NY, USA: ACM, 1994, pp. 314–319.

[3] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," in *22nd SOSP*. Big Sky, MT, USA: ACM, Oct. 2009, pp. 207–220.

[4] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, pp. 70–77, Sep. 1996.

[5] J. Andronick, R. Jeffery, G. Klein, R. Kolanski, M. Staples, H. Zhang, and L. Zhu, "Large-scale formal verification in practice: A process perspective," in *34th International Conference on Software Engineering (ICSE'12)*, 2012, to appear.

[6] H. Zhang, B. Kitchenham, and D. Pfahl, "Reflections on 10 years of software process simulation modelling: A systematic review," in *International Conference on Software Process (ICSP'08)*, vol. LNCS 5007. Leipzig, Germany: Springer, May 2008, pp. 345–365.

[7] ——, "Software process simulation modeling: An extended systematic review," in *International Conference on Software Process (ICSP'10)*, vol. LNCS 6195. Paderborn, Germany: Springer, July 2010, pp. 309–320.

[8] F. P. Brooks, *The design of design: Essays from a computer scientist*. Addison-Wesley, 2010.

[9] D. M. Raffo and W. Wakeland, "Assessing IV&V benefits using simulation," in *28th Annual NASA Goddard Software Engineering Workshop (SEW'03)*. Greenbelt, MD: IEEE, Dec. 2003, pp. 97–101.

[10] D. M. Raffo, U. Nayak, S.-o. Setamanit, P. Sullivan, and W. Wakeland, "Using software process simulation to assess the impact of IV&V activities," in *PROSIM Workshop 2004)*, Edinburgh, Scotland, May 2004, pp. 197–205.

[11] V. Garousi, K. Khosrovian, and D. Pfahl, "A customizable pattern-based software process simulation model: Design, calibration and application," *Software Process: Improvement and Practice*, vol. 14, no. 3, pp. 165–180, 2009.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, ser. LNCS. Springer, 2002, vol. 2283.

[13] J. Strother Moore, "Symbolic simulation: An ACL2 approach," in *Formal Methods in Computer-Aided Design*, ser. LNCS. Springer, 1998, vol. 1522, pp. 530–530.

[14] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *34th POPL*, M. Hofmann and M. Felleisen, Eds. Nice, France: Springer, Jan. 2007, pp. 97–108.

[15] L. Zhu, R. D. Jeffery, M. Staples, M. Huo, and T. T. Tran, "Effects of architecture and technical development process on micro-process," in *International Conference on Software Process (ICSP'07)*, vol. LNCS 4470. Minneapolis, MN, USA: Springer, May 2007, pp. 49–60.