Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Combining Static and Dynamic Analysis for Advanced Certification of Java™ C2IS

F. Painchaud
DRDC Valcartier

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2004-001
March 2004

Canada

Combining Static and Dynamic Analysis for Advanced Certification of Java™ C2IS

Frédéric Painchaud
Defence Research and Development Canada – Valcartier

Defence Research and Development Canada – Valcartier

Technical Memorandum

DRDC Valcartier TM 2004 - 001

March 2004

Author

Frédéric Painchaud, M.Sc.

Approved by

Yves van Chestein, M.Sc., ing.f.
Head/Information and Knowledge Management Section

Approved for release by

Yves van Chestein, M.Sc., ing.f.
Head/Information and Knowledge Management Section

Abstract

Since information-processing systems are becoming omnipresent in our daily lives and in safety-critical infrastructures, there is a strong move to validate and verify them, in order to make them more fault-tolerant and secure. To meet this challenging objective, it is believed that formal methods must be included within the hardware and software engineering processes, which presently suffer from having insufficient (although growing) tool-support with a sound mathematical foundation. Static and dynamic analysis, certifying compilation, and model checking are good examples of such formal methods for hardware and software validation and verification.

This document first presents a summary of two successfully completed projects on malicious code detection and Java certifying compilation, which used formal methods to certify software packages for security purposes. This is followed by an introduction to model checking and Abstract State Machines as a basis for this project. This new project aims at defining and implementing a novel hybrid model-checking approach for Java. This approach is *hybrid* in the sense that it combines static and dynamic analysis principles in order to provide more precise model checking. Ultimately, this project is therefore about using Abstract State Machines (more precisely, the Abstract State Machine Language) as a modeling formalism for Java programs, developing a model checker for this formalism, and parameterizing this model checker in the hope of being able to decide more security properties.

Résumé

Puisque les systèmes d'information deviennent omniprésents dans la vie de tous les jours et dans les infrastructures à sécurité critique, nous assistons à un effort important visant à les valider et à les vérifier de façon à les rendre plus sûrs et insensibles aux défaillances. Pour atteindre cet objectif stimulant, la communauté informatique croit que les méthodes formelles doivent être incluses dans les processus d'ingénierie matériel et logiciel, qui souffrent actuellement de l'appui insuffisant (quoique croissant) d'outils ayant une base mathématique solide. Les analyses statique et dynamique, la compilation certifiée et la vérification de modèles sont de bons exemples de méthodes formelles pour la validation et la vérification du matériel et des logiciels.

Ce document présente d'abord le résumé de deux projets achevés avec succès qui ont porté sur la détection de code malveillant et la compilation certifiée de Java

et qui ont utilisé des méthodes formelles pour certifier la sécurité de logiciels. Ce résumé est suivi d'une introduction à la vérification de modèles et aux machines à états abstraites utilisées comme base pour ce projet de doctorat. Ce nouveau projet vise à définir et implanter une nouvelle approche hybride de vérification de modèles pour Java. Cette approche est *hybride* en ce sens qu'elle combine des principes d'analyse statique et dynamique pour fournir une vérification de modèles plus précise. En bout de ligne, ce projet concerne donc l'utilisation des machines à états abstraites (plus précisément du "Abstract State Machine Language") comme formalisme de modélisation pour les programmes Java, le développement d'un vérificateur de modèles pour ce formalisme et la paramétrisation de ce vérificateur de modèles dans l'espoir d'être capable de décider plus de propriétés de sécurité.

Executive summary

The Java Architecture is believed to be one of the best commercial architectures with which to build C2IS and other critical military information systems because it includes high-quality mechanisms to enforce security policies such as static verification (i.e., prior to execution) and run-time monitoring (i.e., during execution). However, optimized configurations and specialized extensions are needed to satisfy military requirements in terms of reliability and security. Harmonizing the Java Security concepts with validated design and execution surveillance has been identified as a top R & D priority to allow continuous risk management.

This document presents the preliminary version of a novel software verification approach that combines static verification and run-time monitoring principles in order to successfully fulfill reliability- and security-related military requirements. The idea of this approach came from the experience of the MaliCOTS Project, which ended in 2001, that demonstrated the need for a verification technique that unifies static and dynamic analysis principles in a common framework.

Frédéric Painchaud; 2004; Combining Static and Dynamic Analysis for Advanced Certification of Java™ C2IS; DRDC Valcartier TM 2004 - 001; Defence Research and Development Canada – Valcartier.

Sommaire

Plusieurs croient que l'architecture Java est une des meilleures architectures commerciales pour développer des systèmes d'information de commandement et contrôle, et autres systèmes d'information critiques militaires, parce qu'elle contient des mécanismes de qualité pour mettre en force des politiques de sécurité, comme la vérification statique (c'est-à-dire avant l'exécution) et le monitorage dynamique (c'est-à-dire pendant l'exécution). Cependant, des configurations améliorées et des extensions spécialisées sont nécessaires pour satisfaire aux exigences militaires en termes de fiabilité et de sécurité. L'harmonisation des concepts de sécurité Java, de conception validée et de surveillance de l'exécution est considérée comme une priorité importante en R & D afin de permettre la gestion continue du risque.

Ce document présente une version préliminaire d'une nouvelle approche de vérification logicielle qui combine des principes de vérification statique et de monitorage dynamique de façon à mieux satisfaire les besoins militaires en matière de fiabilité et de sécurité logicielles. L'idée de cette approche est venue de l'expérience du projet MaliCOTS, terminé en 2001, qui a démontré le besoin de développer une technique de vérification qui unifie les principes d'analyse statique et dynamique dans un cadre de travail commun.

Frédéric Painchaud; 2004; Combining Static and Dynamic Analysis for Advanced Certification of Java™ C2IS; DRDC Valcartier TM 2004 - 001; Recherche et développement pour la défense Canada – Valcartier.

Table of contents

List of figures

1 Introduction and Motivations

Information-processing systems are becoming omnipresent in our daily lives, either explicitly via personal computers, Internet, and personal digital assistants, or more implicitly via automatic teller machines, mobile phones, entertainment systems, such as audio equipment and high-end televisions, private and public transport – and the list goes on. Concurrently, information systems are integral parts of military transportation, command and control, and small to large military devices such as satellites, radars, and Unmanned Aerial Vehicles (UAVs). Moreover, due to the strong integration and convergence of information technology in all kinds of broad applications, there is increasing dependence on the reliability of hardware and software components. This increased dependence naturally comes with a price: the ever-growing need to assure that these technologies are correct with respect to their original specifications. Unfortunately, in many cases, this requirement cannot be easily verified.[1] In fact, the panels and breakout sessions at the *Workshop on New Visions for Software Design and Productivity* identified, in 2001, the following emerging application challenges that will require further major advances in software technology [2]:

System integration: Perhaps the biggest impact of the IT explosion in the last decade has been the emerging role of computing and software as the "universal system integrator".

Critical infrastructure role: Large-scale software systems increasingly serve as critical infrastructure for banking functions, medical instruments, emergency services, power distribution, telecommunications, transportation, and national security and defence.

Real-time and embedded systems: A rapidly-growing domain of applications, like pacemakers, controllers for power plants, and flight-critical avionics systems, embeds intelligence in physical devices and systems.

Dynamically-changing distributed and mobile applications: Since connectivity among computers and between computers and physical devices proliferates, our systems have become network-centric.

In general, reliability problems and resultant errors can "only" become a nuisance. However, when it comes to safety-critical systems, such as medical and flight-control systems, reliability becomes a matter of life and death and therefore, errors can have a huge human and economic impact. A major problem, though, is that

[1] In fact, it is proven that, given a program P and specifications S, it is undecidable to determine if P conforms to S, for arbitrary P and S [1]. Thus, the presence of faults in software is generally an undecidable property.

the complexity of these systems is usually so high that they become very vulnerable to errors. Even for mass-marketed (not safety-critical) hardware and software applications, the economic impact can be of paramount importance if errors are discovered during system operation. The error that was found in Intel's Pentium floating-point division unit is a good example that has caused an estimated loss of over five hundred million US dollars [3].

In fact, it is a shame to realize that today's software applications are some of the most error-ridden products released by any industry. Unreliable software is currently estimated to cost the US some sixty billion dollars per year [4]. And, of course, as software plays an increasingly critical role in other products and services we depend on, software errors lead to greater problems and can cause the quality of those products to deteriorate. For example, the "intangible" effects of *unavailability* are [5]:

- lost revenue,
- lost productivity, overtime, rework,
- impact on customer commitments, missed deadlines,
- fines/penalties,
- negative impact on customer satisfaction, and
- weakened market position/business image.

Moreover, the future is likely to increase dramatically the number of computer systems that we consider to be safety-critical. The dropping cost of hardware, the improvement in hardware quality, and other technological developments ensure that new applications will be sought in many domains.

Ultimately, the software industry will have to make a shift of mentality in its development process. Production practices in the automobile, electronics, and consumer goods industries followed today's "find and fix" software development process methodology years ago, with similar substandard results. Only when these industries changed their focus from fixing individual problems to fixing the production process, did we see the quality of these goods rise to the level that we have come to rely on today [6]. In other words, rather than learning from the development practices of other industries, software providers focus on fixing the errors they find, but do not try to prevent them.

Moreover, software consumers are a variable in the software low-quality equation: the imperfect software that provides computers with "intelligence" and functionality is purchased because it is affordable and readily available, and has enjoyed the

natural “forgiveness” that comes with being a relatively new technology. Therefore, consumers currently accept these imperfections.

System validation and verification will thus certainly become an important activity. In the literature, system *validation* generally refers to the process of answering the question: “Are we building the right product?”. Therefore, it has to do with the assurance that the design requirements are met in the product being developed. On the other hand, system *verification* answers the question: “Are we building the product right?”, referring to the assurance that the product being developed meets some predefined quality standards. Unless otherwise noted, this document deals exclusively with software system *verification*. Therefore, all the techniques discussed in this document have a common goal: to ensure software quality (in a broad sense). In fact, this document specifically addresses security concerns. But since software security can be seen as a particular form of software quality, one may still talk about system *verification*.

Current software engineering practices show that the quality of system development is checked to a large extent by humans (using the validation/verification technique called “peer reviewing”) and by testing. *Ad hoc* tool-support is involved, but formal methods and tools with a sound mathematical basis are rarely used. Therefore, in serious software design and implementation, where quality is a major concern, more time and effort is spent on *manual* validation and verification than on construction! In the long run, due to the increasing magnitude and complexity of systems, the pressure to reduce system design and development time (“time to market”), and the urge to increase software quality, it is believed that it will be essential to support the system validation and verification process through techniques and tools that facilitate the *automated* analysis of correctness. As Katoen says, citing Wolper[2], in [3]:

> “Manual verification is at least as likely to be wrong as the program itself.”

Moreover, when it comes to computer security, people generally think in terms of a magical solution that will solve the problem all by itself. This is erroneous. Schneier conveys this idea well in his book *Secrets and Lies: Digital Security in a Networked World* [7]. One way to summarize his thesis is that security measures are characterized less by the way they succeed than by the way they fail. In other words, when a *good* security measure miscarries, and it is reasonable to assume that they will all do so eventually, then each single security component failure leaves the whole as unaffected as possible. In engineering, systems that are failure-tolerant are

[2] P. Wolper. Verification: dreams and reality. Inaugural lecture of the course “The algorithmic verification of reactive systems”.

sometimes called *ductile* [8]. Therefore, one way to provide ductility is to consider how a system reacts when something bad happens. But this is not always easy in practice. In order to facilitate the rigorous analysis of such scenarios, formal methods and tools, such as model checkers, need to be used.

Therefore, formal verification techniques that were previously only used on space and nuclear technologies are currently progressively being adopted to verify critical information systems in general.

The rest of this document is organized as follows. Section 2 reviews two related projects which were previously completed: the MaliCOTS project and the JACC project. The MaliCOTS project was aimed at developing a suite of methods and tools in order to analyze Commercial-Off-The-Shelf (COTS) software with respect to a security policy. The JAva Certifying Compilation (JACC) project was aimed at developing a certifying compilation system for Java in order to statically enforce high-level security policies. Section 3 presents current work performed by fellow researchers that is related to this project: model checking, Java model checking, Abstract State Machines (ASMs), and the model checking of ASMs. The abstracts of the papers referenced in the states of the art presented in this section are integrated into this section itself rather than regrouped in an appendix. This decision was made in order to provide better legibility to this document. Section 4 discusses the project: a novel hybrid model-checking approach. This approach is *hybrid* in the sense that it combines static and dynamic analysis principles in order to provide more precise model checking.

2 Past Experience

This section reviews two previously-completed related projects: the MaliCOTS project and the JACC project.

2.1 The MaliCOTS Project

The use of COTS software is attractive for many reasons. The product is relatively inexpensive, it is available almost immediately, and, in many cases, there is already user-acceptance. By selecting a “best-of-breed” product, one also gets a program that is much more advanced and stable than anything which could be developed in-house in a similar time-frame. Everything is not perfect, however, when it comes to deploying COTS software in safety-critical infrastructures. Because COTS software is often developed for the general public, it does not usually meet the higher security and reliability standards required by the armed forces and other safety-critical infrastructures, opening the door to faulty implementations leaving

potential security holes. There is also the possibility of the presence of malicious code, voluntary or not.

To solve this problem, it is necessary to have software go through a rigorous certification process. However, most processes of this type are still largely manual in nature, with peer reviewing and testing being the most popular. The problem is exacerbated when end users want to perform the certification, because they often have no access to the source code. Therefore, automated tools are needed to perform the mechanical certification of COTS software. The ability to discover potentially malicious code in COTS software was thus the premise of the MaliCOTS project.

2.1.1 Executive Context

In 1995 at Defence Research & Development Canada (DRDC) – Valcartier, it was realized that the widespread use of COTS tools in critical military environments introduces many new security and reliability risks. To determine how to counter this emerging threat vector, a feasibility study identified three possible research paths to be further investigated: static analysis of the binary code, dynamic analysis of the binary code, and certifying the source code at compile-time.

Following this feasibility study, the MaliCOTS project decided to build a prototype for each path. The various approaches are discussed below. Over a four-year period, the project involved twelve graduate students and four professors from Université Laval and two scientists from DRDC Valcartier. The project was completed in the summer of 2001.

Each of these three paths is now detailed in the following subsections.

2.1.2 Static Analysis

Static analysis can be defined as the science of determining the behaviour of a program without running it, i.e., by examining the code, be it source or assembly. It is widely used for program optimization and quality assurance. In the SAMCode prototype, IDAPro, a commercial disassembler, is used to provide the assembly language. It is then fed into the SAMCode tool and a flowchart of the code is constructed and is used to model-check against specific security properties. Basic security attributes, such as access to the network, can be detected very rapidly. More complex security properties, expressed by automata, can also be used to specify patterns of behaviour. The model checker then tries to match these “patterns” in the model. The tool works in Microsoft Windows 2000 and monitors the use of specific API functions of this operating system. Details can be found in [9].

Among the strengths of static analysis, identification of dead code and hidden functionalities has been demonstrated in the past. More generally, static analysis has the ability to detect inappropriate logic by analyzing all execution paths in an abstract manner. Finally, program visualization and code understanding are made possible by a large number of commercial static analysis products.

However, when the source code is not available, certifying components in this manner becomes extremely difficult, if not impossible. Moreover, imprecision can become a problem since undecidability is inevitably encountered. Finally, copyright issues might also preclude the decompilation of executable code for analysis in some abstract representation.

2.1.3 Dynamic Analysis

In dynamic analysis, the program is executed and its behaviour is examined against a security policy. There are many possible approaches. The one used in the DAMon prototype is to instrument Microsoft Windows 2000 in order to intercept all relevant system calls. Since Microsoft Windows 2000 is designed so that all calls to system resources need to follow a specific path, this gives very good control. In the prototype, all calls to the file system, the registry, the communication ports, and the creation and destruction of processes are intercepted by a separate monitor. Access is controlled by a security policy that can change with time to adapt to new situations. The novel idea is that the system resource monitors collaborate with each other to provide more security. For example, if the file monitor detects an access to a sensitive file, it can tell the port monitor not to let anything out on the network. More details can be found in [10].

Dynamic analysis is a pragmatic approach that offers several short-term benefits since it makes use of all the information during execution, including user inputs and network transactions. However, formal and complete certification with this technique is difficult to achieve. This is due to the fact that dynamic analysis can only show the *presence* of errors, not their *absence*. From past experience, dynamic analysis must be focused on specific tasks to be useful. For example, surveillance of user behaviour and transactions on the network and the monitoring of concurrent processes appear to be natural application domains. In fact, in many circumstances, dynamic analysis appears to be the only practical way of managing them.

2.1.4 Certifying Compilation

Using static and dynamic analyses (see Subsections 2.1.2 and 2.1.3) allows for a relatively effective semi-automatic certification. However, the whole process is long and arduous and it is a losing battle since it needs to be repeated every time

a new update is issued – and perhaps for every single little patch that a software vendor releases. Furthermore, what can be done when a problem is found? In the case of COTS software, the only option is usually to rely on the goodwill of the vendor. Even if the vendor does comply, the burden of proof of compliance still rests on the post-compilation certification process.

Improvements can be achieved by certifying security at the compiler level, working from source code. The certification process can then be automated and it benefits from having access to more information. An example of an implementation of certifying compilation, also known as self-certified code, is the JACC project (see Subsection 2.2).

Using Java as an example, certifying compilation works in the following way. A certifying compiler first parses the source code, producing two outputs: the regular bytecode to be executed by the Java virtual machine and security annotations. These annotations, along with the bytecode, are used to create a model of the behaviour of the program. This model is then checked against a security policy by a verifier that includes a model checker. More details can be found in [11].

The main advantages of the certifying compilation approach include: certification can be sound and complete (if the security policy is verifiable) even for very large software packages; in addition to the compliance with the "verify once, execute many times" paradigm, execution time is generally not increased or only slightly so (if the security policy is relatively simple) by a short verification prior to execution; and editors can deliver a trusted component without revealing the intellectual property that the source code represents.

The main difficulty with sound and complete (where possible) model-based certification techniques is their complexity. Indeed, large software means large (if not *infinite*) state spaces that do not fit in memory. This is referred to as the *state-space explosion problem*. However, many strategies have been developed to tackle this problem: symbolic representation of state spaces, efficient memory management strategies, partial-order reduction, compositional verification, and abstract interpretation.

2.1.5 Results and Conclusions

The four-year MaliCOTS project, which ended in 2001, produced four prototypes: an x386-assembly static analysis tool, a dynamic monitor for Microsoft Windows 2000, a C certifying compiler, and a model checker for this certifying compiler.

These prototypes revealed that each technique has its own strengths and weaknesses and the MaliCOTS project gave a better understanding of where and when a specific

tool should be used.

From this hands-on experience, the most important conclusion was that certification technologies should cooperate to fulfill their objectives – but the question is how this should be done. An approach to answer this question is presented in Subsection 4.1.

2.2 The JACC Project

The Java language emerged as a multi-paradigmatic (object-oriented, parallel, distributed, secure, etc.) language that supports mobile code. It obeys the paradigm "write once, run everywhere". Indeed, programs in a compiled platform-independent form (class file or Java bytecode) can migrate from anywhere in the network. Java bytecode can be executed on any platform that runs a Java virtual machine (JVM) that emulates a processor architecture. The most popular way to achieve code mobility in Java is to put links in web pages to Java class files (usually referred to as "applets").

Mobile code in general and Java in particular pose severe, and very interesting, challenges in terms of security, reliability, and performance. The security issue is of paramount importance. The host client accepting a mobile code must ensure that it will not affect the secrecy (by leaking sensitive information), integrity (by corrupting information), authentication (by impersonating authorized principals), availability (by denying services to legal users), etc., of its system. The current trends in mobile code and Java security are: defensive (adding layers of firewalling, cryptographic protocols, network partitions, etc.), restrictive (sandbox models, rigid security policies, etc.), and ad hoc (dynamic checks, checksums, scanning, etc.).

Consequently, there was a move to develop a security architecture that is flexible, application-dependent, efficient, and practical. Indeed, this architecture must be flexible enough to enforce a wide range of application- and context-specific security guidelines and policies. Moreover, such an architecture has to be based on robust theoretical foundations. This was the premise of the JACC project.

2.2.1 Executive Context

The Java language uses multi-level mechanisms in order to ensure its protection: language, compiler, bytecode verifier, and security manager. This architecture is one of the best in ensuring safe and secure execution of Java applications. However, it lacks certain properties:

- **Flexibility:** the current security architecture of the JVM is rigid because it is

application-independent. Security policies are simply a series of access permissions for resource usage independent of the execution context.

- **Efficiency:** security property verification is performed dynamically by the JVM, which can cause performance issues.

- **Robustness:** the security architecture of the JVM is vaguely and incompletely described in the official documentation published by JavaSoft Inc. [12]. The inner workings of the bytecode verifier are not well documented. Moreover, numerous errors have been found in the bytecode verifier which shows that its development may not have been carried out using robust theoretical foundations.

The JACC project was concerned with the formal static verification of expressive security properties written for Java programs in order to enhance the current Java security architecture. To address this problem, several approaches used to ensure safe local execution of untrusted code have been studied. Among them, certifying compilation seems to be a very promising approach. It is based on programming language theory and implementation. The main idea of certifying compilation is to generate certified code that can be verified by the consumer using a verifier to check if the software complies with a pre-established safety or security policy. The end-user (or organization) can "verify once and execute many times" with assurance that there will be no policy violations.

An architecture has been developed for secure compilation and execution of Java mobile code that is based on an extension of the certifying compilation approach. This architecture aims to solve the three main problems previously enumerated: the lack of flexibility, efficiency, and robustness in the current Java security architecture. A Java certifying compiler was designed and implemented so that it inserts new type annotations into Java class files. A language for the specification of expressive security policies was also developed. Finally, a bytecode verifier that integrates a model checker was produced.

2.2.2 Overview of the JACC Approach

In the JACC architecture, the Java language stays totally unchanged and thus the Java source code that is compiled is exactly the same that Java developers are accustomed to writing. However, the Java compiler is modified. Instead of producing only the bytecode and the standard type annotations, it produces additional annotations. Instead of being directly executed by the Java virtual machine, the augmented class files are verified by a modified Java bytecode verifier, and this verifier guarantees both safety properties and high-level security policies. The safety properties (memory, type, and control flow safety) stay the same but the security

policies are now expressed with a more expressive custom language based on the modal μ-calculus. Once the bytecode is successfully verified, it can be executed by a traditional Java virtual machine. Since high-level security is already guaranteed, the Java security manager can be turned off and efficiency is therefore increased. Finally, secure execution of the bytecode is achieved, if absolute assurance can be obtained via the JACC approach. More details can be found in [11].

2.2.3 Results and Conclusions

The JACC project produced a prototype of a Java certifying compilation platform that statically detects several cases of suspicious program behaviour. The most useful feature of the security policy specification language is the possibility of using parameterized actions. An example of such action is `read("filename.ext", F)`, where `read()` is the parameterized action and `"filename.ext"` and `F` are its parameters. These parameters help in clarifying the meaning of the action. Thanks to these actions, one is now able to find the statically-defined file names and URLs used by a program. In some cases, where data is not statically known, the use of variables as action parameters has proven to be very useful as an abstraction for missing data. This means that even if one cannot certify with certainty that a particular file is used, one can at least suspect it.

The main conclusion was that the objective was achieved using certifying compilation. Indeed, the JACC platform is more flexible (the security policy language is expressive and permits specifying high-level security properties), more efficient (the verification process is exclusively static), and more robust (the security model is based on formal foundations) than the existing Java security architecture.

3 Current Related Work

This section presents the current work being done by fellow researchers on model checking, Java model checking, Abstract State Machines (ASMs), and the model checking of ASMs. ASMs will be used as a formal foundation for this project (see Section 4).

3.1 Model Checking

Applying model checking to a design consists of three main tasks:

- Modeling,
- Specifying, and

- Verifying.

Figure 1 illustrates the process of model checking.

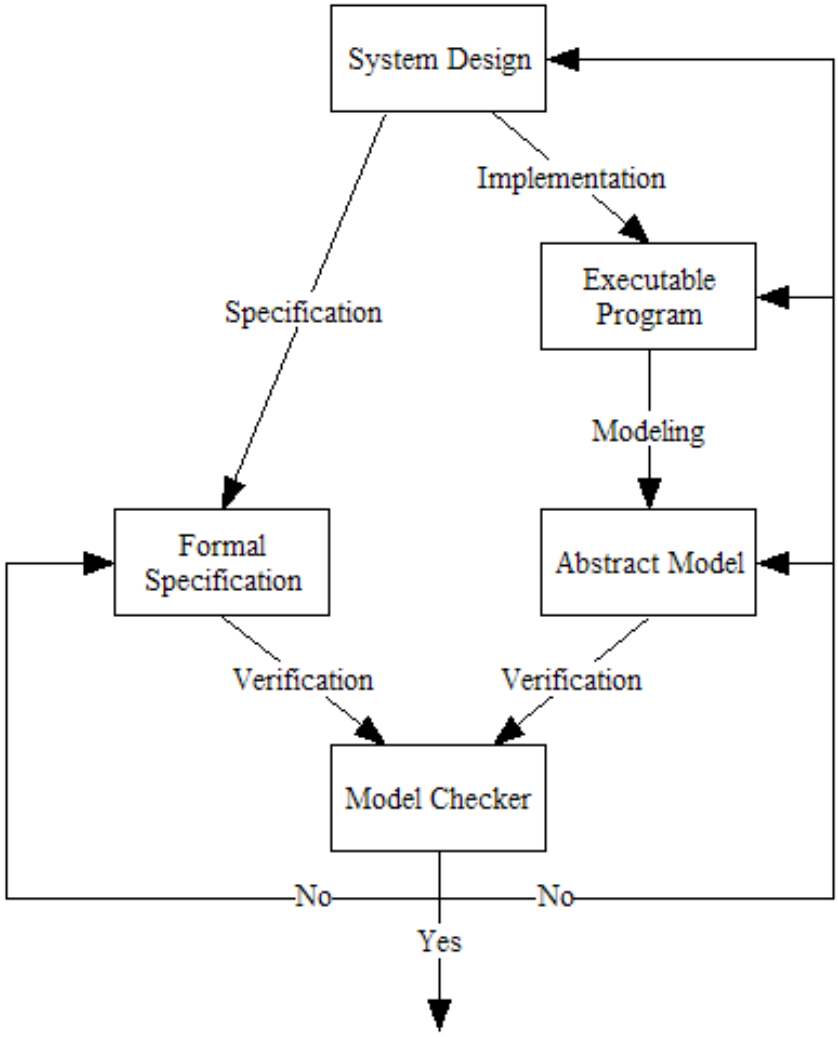

Figure 1: *The process of model checking*

The three main tasks are detailed in the following subsections. These subsections focus on concurrent systems because they are the most interesting systems that are traditionally used in model checking.

3.1.1 Modeling

Concurrent (and reactive) systems are nonterminating systems composed of different processes running in parallel, that can interact with the environment, i.e., with

the user, for instance.

The first task is to convert the design of the system into an abstract model accepted by a model-checking tool. The abstract model should eliminate irrelevant details of the design with respect to the properties to verify.

In some cases, in hardware design for example, this conversion can be done automatically. Some research work has been done in order to convert software designs automatically. In most cases, however, human guidance and assistance is needed.

Kripke structures are usually used to model the behaviour of concurrent systems. Informally, they are node-labelled graphs. The nodes of the graph model system states and are labelled with information that is true in that state. The edges of the graph represent system transitions as the result of some action of the system.

3.1.2 Specifying

Before verifying, it is necessary to state the properties that the system design must satisfy.

The specification is usually given in some logical formalism. It is common to use temporal logic, which can assert how the behaviour of the systems evolves over time.

Important issues in specification are:

- **Consistency:** Is the given specification consistent?
- **Completeness:** Does the given specification cover all the properties that the system should satisfy?
- **Validity:** Does the given specification correspond to the analyst's intention?

Most generally, these issues are not directly addressed in model checking.

3.1.3 Verifying

Ideally, verification is completely automatic. However, in practice, it often involves human assistance.

One such manual activity is the analysis of the verification results. In the case of a negative result, the analyst is provided with an error trace. This can help the designer in tracking down where the error occurred.

In this case, analyzing the error trace may require a modification of the system and reapplication of the model checking algorithm.

Unfortunately, an error trace can also result from a false negative, that is from:

- incorrect modeling of the system,
- incorrect formalization of the specification, and
- inconsistent specification.

A final possibility is that the verification task fails to terminate normally due to the size of the model, when it is too large to fit into the computer memory.

Despite the current limitations of model checking, it is a very promising verification technology which currently gains from an intense and broad research effort.

3.2 Java Model Checking – Annotated State of the Art

This subsection constitutes an annotated state of the art on Java model checking. Only the most relevant references are given. See [13] for a complete state of the art on Java model checking. Note that if you have the CD-ROM distribution of this document, you directly have access to the referenced papers by clicking on the links.

3.2.1 The Bandera Project

The following papers are related to the Bandera project, conducted at Kansas State University. Bandera is one of the major Java model checkers available today.

Slicing Multi-threaded Java Programs: A Case Study
Matthew B. Dwyer (dwyer@cis.ksu.edu)
James C. Corbett (corbett@ics.hawaii.edu)
John Hatcliff (hatcliff@cis.ksu.edu)
Stefan Sokolowski (stefan@cis.ksu.edu)
Hongjun Zheng (zheng@cis.ksu.edu)
http://www.cis.ksu.edu/santos/bandera/papers/slicing.html
Local file: papers/slicestudy.pdf
1999

From the document:

Program slicing is becoming increasingly popular as an initial step in the construction of finite-state models for automated verification. As part of a project aimed at building tools to automate the extraction of compact, sound finite-state models of concurrent Java programs, we have developed the theoretical foundations of slicing threaded programs that use Java monitors and wait/notify synchronization. In this paper, we describe how these foundations are incorporated into a tool that slices multi-threaded Java programs. We describe a simple static analysis that can be used to refine the underlying dependencies used by the slicer and illustrate the effectiveness of this refinement by describing the slicing of a realistic Java program.

Specializing Configurable Systems for Finite-state Verification
John Hatcliff (hatcliff@cis.ksu.edu)
Matthew B. Dwyer (dwyer@cis.ksu.edu)
Shawn Laubach (laubach@cs.okstate.edu)
Nanda Muhammad (nanda@cs.okstate.edu)
http://www.cis.ksu.edu/santos/bandera/papers/abstraction.html
Local file: papers/ssfv.pdf
1999

From the document:

As finite-state verification techniques and tools, such as model checkers, continue to mature, researchers and practitioners attempt to apply them in increasingly realistic software development settings. Concurrent applications, and components of those applications, are often implemented as configurable systems (i.e., where size, structure or selected behavior aspects are taken as system inputs). These systems are typically implemented using dynamically allocated data and threads of control. This use of dynamism makes it very difficult to render behavioral models of configurable systems that would be suitable as input to finite-state verification tools.

Currently, configurable systems can only be verified by performing hand-transformations of the source code that are often time-consuming, tedious, and error-prone. In this paper, we apply partial evaluation techniques to transform source code automatically into a form from which finite-state systems can be extracted. We illustrate these techniques by transforming real configurable software systems to a form that can be input to existing finite-state verification tools. This demonstrates one way that partial evaluation technology has the potential to significantly extend the applicability of finite-state verification tools for software systems.

Bandera: Extracting Finite-state Models from Java Source Code
James C. Corbett (corbett@hawaii.edu)
Matthew B. Dwyer (dwyer@cis.ksu.edu)
John Hatcliff (hatcliff@cis.ksu.edu)
Shawn Laubach (laubach@cis.ksu.edu)
Corina S. Pasareanu (pcorina@cis.ksu.edu)
Robby (robby@cis.ksu.edu)
Hongjun Zheng (zheng@cis.ksu.edu)
http://www.cis.ksu.edu/santos/bandera/
Local file: papers/bandera.pdf
2000

From the document:

> *Finite-state verification techniques, such as model checking, have shown promise as a cost-effective means for finding defects in hardware designs. To date, the application of these techniques to software has been hindered by several obstacles. Chief among these is the problem of constructing a finite-state model that approximates the executable behavior of the software system of interest. Current best-practice involves hand-construction of models which is expensive (prohibitive for all but the smallest systems), prone to errors (which can result in misleading verification results), and difficult to optimize (which is necessary to combat the exponential complexity of verification algorithms).*
>
> *In this paper, we describe an integrated collection of program analysis and transformation components, called Bandera, that enables the automatic extraction of safe, compact finite-state models from program source code. Bandera takes as input Java source code and generates a program model in the input language of one of several existing verification tools; Bandera also maps verifier outputs back to the original source code. We discuss the major components of Bandera and give an overview of how it can be used to model check correctness properties of Java programs.*

Bandera Specification Language: A Specification Language for Software Model Checking
Robby
http://www.cis.ksu.edu/santos/bandera/papers/specification.html
Local file: papers/bsl-thesis.pdf
2000

From the document:

Finite-state verification techniques, such as model checking, are a promising technique for finding defects in software systems. However, there are several obstacles that hinder the application of software model checking in practice. One of the obstacles is the property specification problem: taking the informal requirements of the system and writing their property specifications. The current practice in property specification for model checking requires analysts to translate the property to the model level. This has several disadvantages: (1) it forces the specification to be stated in term of the model's representation. This requires the understanding of these typically highly optimized representations to accurately render the specifications. (2) The model representations may change depending on which optimizations and abstractions were used when generating the model. When changed, it requires the specifications to be changed. (3) Model checker tools have different inputs (temporal logics) for property specification. If more than one model checker is used, then the specifications have to be recoded and maintained in a different language. (4) Analysts sometimes find it difficult to use temporal logics to accurately express the properties of software. Once written, these specifications are often hard to reason about, debug, and modify. Even greater difficulties are encountered when describing dynamic component properties, such as head-allocated structures.

In this thesis, we present Bandera Specification Language (BSL), a source level specification language for model checking Java programs. BSL addresses the property specification problem by leveraging the property specification to the source level, and by using temporal specification patterns to abstract away from specific temporal logics and to ease writing and maintaining the specifications. We present the syntax and informal semantics of BSL. We also present how program slicing can be driven by properties that are specified in BSL. We then present how quantifications are supported in BSL for describing dynamic component properties. Finally, we give a methodology for using BSL and show some examples of BSL applied to non-trivial programs.

Using the Bandera Tool Set to Model-check Properties of Concurrent Java Software

John Hatcliff (hatcliff@cis.ksu.edu)
Matthew Dwyer (dwyer@cis.ksu.edu)
http://www.cis.ksu.edu/~hatcliff/

Local file: papers/CONCUR01-tutorial.pdf

2001

From the document:

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin, dSpin, SMV, and JPF). Both program slicing and user extensible abstract interpretation components are applied to customize the program model to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

In this tutorial paper, we use a simple concurrent Java program to illustrate the functionality of the main components of Bandera and how to interact the tool set using its graphical user interface.

3.2.2 The Java Modeling Language Project

The following papers are related to the Java Modeling Language (JML) project, conducted at Iowa State University. JML is a behavioral interface specification language tailored to Java. It is designed to be written and read by working software engineers, and should require only modest mathematical training. It is similar to UML but specifically adapted to Java.

JML: A Notation for Detailed Design
Gary T. Leavens
Albert L. Baker
Clyde Ruby
http://www.cs.iastate.edu/~leavens/JML/Documentation/index.html
Local file: papers/jmlkluwer.pdf
1999

From the document:

JML is a behavioral interface specification language tailored to Java. It is designed to be written and read by working software engineers, and should require only modest mathematical training. It uses Eiffel-style syntax combined with model-based semantics, as in VDM and Larch. JML supports quantifiers, specification-only variables, and other enhancements that make it more expressive for specification than Eiffel and easier to use than VDM and Larch.

A Logic for the Java Modeling Language JML
Bart Jacobs (bart@cs.kun.nl)
Erik Poll (erikpoll@cs.kun.nl)
http://www.cs.iastate.edu/~leavens/JML/Relatedpapers/index.html
Local file: papers/CSI-R0018.pdf
2000

From the document:

> *This paper describes a specialised logic for proving specifications in the Java Modeling Language (JML). JML is an interface specification language for Java. It allows assertions like invariants, constraints, pre- and post-conditions, and modifiable clauses as annotations to Java classes, in a design-by-contract style. Within the LOOP project at the University of Nijmegen JML is used for specification and verification of Java programs. A special compiler has been developed which translates Java classes together with their JML annotations into logical theories for a theorem prover (PVS or Isabelle). The logic for JML that will be described here consists of tailor-made proof rules in the higher order logic of the back-end theorem prover for verifying translated JML specifications. The rules efficiently combine partial and total correctness (like in Hoare logic) for all possible termination modes in Java, in a single correctness formula.*

3.2.3 The Java PathFinder Project

The following papers are related to the Java PathFinder (JPF) project, conducted at NASA Ames Research Center. JPF is one of the major Java model checkers available today.

Java PathFinder: Second Generation of a Java Model Checker
Guillaume Brat (brat@ptolemy.arc.nasa.gov)
Klaus Havelund (havelund@ptolemy.arc.nasa.gov)
SeungJoon Park (spark@ptolemy.arc.nasa.gov)
Willem Visser (wvisser@ptolemy.arc.nasa.gov)
http://ase.arc.nasa.gov/people/visser
Local file: papers/wave00.pdf
2000

From the document:

Model checking is seldom applied to implementation programs. Furthermore, when it is applied, the usual approach is to extract relevant portions of the code, create a model of its behavior in a different notation, and then check the latter. This approach has the drawback that it requires expertise in the use of the model checking tools and hence will not, in general, allow software developers to check their own code during development. In the Automated Software Engineering group at NASA Ames we are investigating the use of model checking on actual source code in order to allow (NASA) software developers to augment their traditional testing techniques with model checking. Here we describe our work on applying model checking to Java programs.

Finding Feasible Counter-examples when Model Checking Abstracted Java Programs
Corina S. Pasareanu (pcorina@cis.ksu.edu)
Matthew B. Dwyer
Willem Visser
http://ase.arc.nasa.gov/people/visser
Local file: papers/choosefree.pdf
2001

From the document:

Despite recent advances in model checking and in adapting model checking techniques to software, the state explosion problem remains a major hurdle in applying model checking to software. It is well-accepted that automated techniques for abstracting programs will be necessary to overcome this problem. Most common abstraction techniques compute an upper approximation of the original program. Thus, when a specification is true for the abstracted program, it will also be true for the concrete program. However, if the specification is false for the abstracted program, the counter-example may be the result of some behavior in the abstracted program which is not present in the original program. We have extended an explicit-state model checker, Java PathFinder (JPF), to analyze counter-examples in the presence of abstractions, such as those introduced by the Bandera toolset. We enhanced JPF with an option to search for "feasible" (i.e. nondeterminism-free) counter-examples "on-the-fly", during model checking. Alternatively, an abstract counter-example can be used to guide the simulation of the concrete computation and thereby check feasibility of the counter-example. We demonstrate that these techniques can be effective in analyzing counter-examples from checks of several non-trivial multi-threaded Java programs.

Addressing Dynamic Issues of Program Model Checking
Flavio Lerda (flerda@riacs.edu)
Willem Visser (wvisser@riacs.edu)
http://ase.arc.nasa.gov/people/flerda/index.shtml
Local file: papers/spin01.pdf
2001

From the document:

> *Model checking real programs has recently become an active research area. Programs however exhibit two characteristics that make model checking difficult: the complexity of their state and the dynamic nature of many programs. Here we address both these issues within the context of the Java PathFinder (JPF) model checker. Firstly, we will show how the state of a Java program can be encoded efficiently and how this encoding can be exploited to improve model checking. Next we show how to use symmetry reductions to alleviate some of the problems introduced by the dynamic nature of Java programs. Lastly, we show how distributed model checking of a dynamic program can be achieved, and furthermore, how dynamic partitions of the state space can improve model checking. We support all our findings with results from applying these techniques within the JPF model checker.*

3.2.4 Other Java Model Checking-Related Papers

The following papers are not necessarily directly related to Java model checking but could possibly be used in that context or in the context of hybrid model checking (see Section 4).

Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking
Willem Visser
SeungJoon Park
John Penix
http://ase.arc.nasa.gov/people/visser
Local file: papers/fmsp00.pdf
2000

From the document:

While it is becoming more common to see model checking applied to software requirements specifications, it is seldom applied to software implementations. The Automated Software Engineering group at NASA Ames is currently investigating the use of model checking for actual source code, with the eventual goal of allowing software developers to augment traditional testing with model checking. Because model checking suffers from the state-explosion problem, one of the main hurdles for program model checking is reducing the size of the program. In this paper we investigate the use of abstraction techniques to reduce the state-space of a real-time operating system kernel written in C++. We show how informal abstraction arguments could be formalized and improved upon within the framework of predicate abstraction, a technique based on abstract interpretation. We introduce some extensions to predicate abstraction that all allow it to be used within the class-instance framework of object-oriented languages. We then demonstrate how these extensions were integrated into an abstraction tool that performs automated predicate abstraction of Java programs.

Using Runtime Analysis to Guide Model Checking of Java Programs
Klaus Havelund (havelund@ptolemy.arc.nasa.gov)
http://ase.arc.nasa.gov/havelund/
Local file: papers/runtime.pdf
2000

From the document:

This paper describes how two runtime analysis algorithms, an existing data race detection algorithm and a new deadlock detection algorithm, have been implemented to analyze Java programs. Runtime analysis is based on the idea of executing the program once, and observing the generated run to extract various kinds of information. This information can then be used to predict whether other different runs may violate some properties of interest, in addition of course to demonstrate whether the generated run itself violates such properties. These runtime analyses can be performed standalone to generate a set of warnings. It is furthermore demonstrated how these warnings can be used to guide a model checker, thereby reducing the search space. The described techniques have been implemented in the home grown Java model checker called Java PathFinder.

Model-Checking Multi-Threaded Distributed Java Programs
Scott D. Stoller (stoller@cs.sunysb.edu)

http://www.cs.sunysb.edu/~stoller/publications.html
Local file: papers/SPIN2000-STTT.pdf
2000

From the document:

> *State-space exploration is a powerful technique for verification of concurrent software systems. Applying it to software systems written in standard programming languages requires powerful abstractions (of data) and reductions (of atomicity), which focus on simplifying the data and control, respectively, by aggregation.We propose a reduction that exploits a common pattern of synchronization, namely, the use of locks to protect shared data structures. This pattern of synchronization is particularly common in concurrent Java programs, because Java provides built-in locks. We describe the design of a new tool for state-less state-space exploration of Java programs that incorporates this reduction. We also describe an implementation of the reduction in Java PathFinder, a more traditional state-space exploration tool for Java programs.*

Combining Static Analysis and Model Checking for Software Analysis
Guillaume Brat (brat@email.arc.nasa.gov)
Willem Visser (wvisser@email.arc.nasa.gov)
http://ase.arc.nasa.gov/people/visser
Local file: papers/mcsa.pdf
2001

From the document:

> *We present an iterative technique in which model checking and static analysis are combined to verify large software systems. The role of the static analysis is to compute partial order information which the model checker uses to reduce the state space. During exploration, the model checker also computes aliasing information that it gives to the static analyzer which can then refine its analysis. The result of this refined analysis is then fed back to the model checker which updates its partial order reduction. At each step of this iterative process, the static analysis computes optimistic information which results in an unsafe reduction of the state space. However, we show that the process converges to a fixed point at which time the partial order information is safe and the whole state space is explored.*

Transformations for Model Checking Distributed Java Programs
Scott D. Stoller (stoller@cs.sunysb.edu)
Yanhong A. Liu (liu@cs.sunysb.edu)
http://www.cs.sunysb.edu/~stoller/publications.html
Local file: papers/SPIN2001.pdf
2001

From the document:

> *This paper describes three program transformations that extend the scope of model checkers for Java programs to include distributed programs, i.e., multi-process programs. The transformations combine multiple processes into a single process, replace remote method invocations (RMIs) with local method invocations that simulate RMIs, and replace cryptographic operations with symbolic counterparts.*

3.3 Abstract State Machines

To introduce Abstract State Machines (ASMs), the following are a few citations from the literature that give the main motivations for using them.

The Abstract State Machines online bibliography, compiled by Jim Huggins at the University of Michigan, is arguably the most comprehensive and complete online reference on ASMs [14]. Here is what is given on its presentation page as an introduction to ASMs:

> The Abstract State Machine (ASM) project (formerly known as the Evolving Algebras project) was started by Yuri Gurevich as an attempt to bridge the gap between formal models of computation and practical specification methods.
>
> The ASM thesis is that any algorithm can be modeled at its natural abstraction level by an appropriate ASM. Based upon this thesis, members of the ASM community have sought to develop a methodology based upon mathematics which would allow algorithms to be modeled naturally; that is, described at their natural abstraction levels. The result is a simple methodology for describing simple abstract machines which correspond to algorithms. Plentiful examples exist in the literature of ASMs applied to different types of algorithms.

In the same online reference, one may find arguments in favour of ASMs being *precise*, *faithful*, *understandable*, *executable*, *scalable*, and *general*.

Moreover, Egon Börger made the following statements in a tutorial he wrote [15]:

> The method built around the notion of Abstract State Machine (ASM) has been proven to be a scientifically well founded and an industrially viable method for the design and analysis of complex systems, which has been applied successfully to programming languages, protocols, embedded systems, architectures, requirements engineering, etc. The analysis covers both verification and validation, using mathematical reasoning (possibly theorem-prover-verified or model-checked) or experimental simulation (by running the executable models).

and also:

> Although the definition of ASMs is surprisingly simple, Abstract State Machines offer a certain number of theoretically well-founded and industrially useful methods that support the entire software development cycle. These include rigorous modelling and analysis methods (both mathematical verification and experimental validation) for the requirements, during the early phases of software development, supporting their elicitation, specification, inspection, and testing through so-called ground models, and the refinement of the high-level models, through a design process which reliably connects the requirements to the code, supporting a practical documentation discipline for code maintenance and reuse.

For a formal presentation of ASMs, please consult Annexes A and B.

3.4 Abstract State Machines – Annotated State of the Art

This subsection constitutes an annotated state of the art on Abstract State Machines. Only the most relevant references are given. See [16] for a complete state of the art on ASMs. Note that if you have the CD-ROM distribution of this document, you directly have access to the referenced papers by clicking on the links.

3.4.1 The First Step

The following two papers can be considered as the first step in defining Abstract State Machines, formerly known as Evolving Algebras. This former name was chosen because it reflects the fact that Abstract State Machines are algebras (Tarski's structures, without relations in that past context) with rules to make them evolve. However, this name confused logicians who were already familiar with Dynamic Algebras. Therefore, the name was changed to Abstract State Machines around 1995.

Evolving Algebras
Yuri Gurevich (gurevich@microsoft.com)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/introifip.pdf
1991

From the document:

> *When you use evolving algebras, you develop a strong feeling that ealgebras can model nicely any algorithmic process. The EA thesis can be seen as an attempt to express that feeling in terms of the EA computation model. The EA thesis was also the initial goal of the EA project.*

Evolving Algebras: An Attempt to Discover Semantics
Yuri Gurevich (gurevich@microsoft.com)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/tutorial.pdf
1993

From the document:

> *This tutorial is based on lecture notes from the Fall 1990 course on Principles of Programming Languages at the University of Michigan. (My young friend Quisani did not attend the lectures.) The present version incorporates some changes provoked by the necessity to update the bibliography. The main part of the paper is still the same, however, and the examples are unchanged even though many things happened in the meantime. In particular, we (the collective we) have learned how to build evolving algebras by the method of successive refinements, and the current evolving algebra description of the C programming language in [GH] doesn't look much like*

the strcpy example anymore. Now, we understand better how to compose evolving algebras and how to prove things with evolving algebras. A typical misconception is that the operational approach is necessarily too detailed. Some people think that an approach suited for complexity analysis does not give a good high-level specification language. I believe in a high-level specification language based on evolving algebras; the successive refinement method is then one tool to prove implementation correctness. But this and various other issues (how to incorporate real time into evolving algebras for example) will have to be addressed elsewhere.

3.4.2 THE Reference

This paper is certainly the most referenced one among all ASM-related papers. It is obviously of great interest, as it gives the formal semantics of ASMs, but it is outdated. Gurevich has promised to publish an updated one someday.

Evolving Algebras 1993: Lipari Guide
Yuri Gurevich (gurevich@microsoft.com)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/guide.pdf
1995

From the document:

Computation models and specification methods seem to be worlds apart. The evolving algebra project started as an attempt to bridge the gap by improving on Turing's thesis [G1, G2]. We sought more versatile machines which would be able to simulate arbitrary algorithms in a direct and essentially coding-free way. Here the term algorithm is taken in a broad sense including programming languages, architectures, distributed and real-time protocols, etc. The simulator is not supposed to implement the algorithm on a lower abstraction level; the simulation should be performed on the natural abstraction level of the algorithm.

The evolving algebra thesis asserts that evolving algebras are such versatile machines. The thesis suggests an approach to the notorious correctness problem that arises in mathematical modelling of non-mathematical reality: How can one establish that a model is faithful to reality? The approach is to construct an evolving algebra A that reflects the given computer system so closely that the correctness can be established by observation and experimentation. (There are tools for running evolving algebras.) A can then

be refined or coarsened and used for numerous purposes. An instructive example is described in [B] by Egon Börger who championed this approach and termed A the ground model of the system. The use of the successive refinement method is facilitated by the ability of evolving algebras to reflect arbitrary abstraction levels.

3.4.3 Philosophy of Abstract State Machines

The following papers cover ASMs from a philosophical point-of-view. Subjects range from *What is computation?* to *What are ASMs good for?* and *formalware development technologies based on ASMs*, that is software developed by using formal methods from design to implementation. They are of great interest for the computer science enthusiast.

Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code
Egon Börger (boerger@di.unipi.it)
Christoph Beierle
Igor Durdanovic (igord@research.nj.nec.com)
Uwe Glässer (glaesser@upb.de)
Elvinia Riccobene (riccobene@dmi.unict.it)
http://www.di.unipi.it/~boerger/Papers/SwEngg
Local file: papers/steamboiler.pdf
1995

From the document:

The steam boiler control specification problem is used to illustrate how the evolving algebra approach to the specification and the verification of complex systems can be exploited for a reliable and well documented development of executable, but formally inspectable and systematically modifiable code.

A hierarchy of stepwise refined abstract machine models is developed, the ground version of which can be checked for whether it faithfully reflects the informally given problem. The sequence of machine models yields various abstract views of the system, making the various design decisions transparent, and leads to a C++ program. This program has been demonstrated during the Dagstuhl-Meeting on Methods for Semantics and Specification, in June 1995, to control the Karlsruhe steam boiler simulator satisfactorily.

Why Use Evolving Algebras for Hardware and Software Engineering ?
Egon Börger (boerger@di.unipi.it)
http://www.eecs.umich.edu/gasm
Local file: papers/whyuse.pdf
1995

From the document:

> *In this paper I answer the question how evolving algebras can be used for the design and analysis of complex hardware and software systems. I present the salient features of this new method and illustrate them through several examples from my work on specification and verification of programming languages, compilers, protocols and architectures. The definition of a mathematical model for Hennessy and Patterson's RISC architecture DLX serves as a running example; this model is used in [24] to prove the correctness of instruction pipelining. I will point out the yet unexplored potential of the evolving algebra method for large-scale industrial applications.*

High Level System Design and Analysis Using Abstract State Machines
Egon Börger (boerger@di.unipi.it)
ftp://ftp.di.unipi.it/pub/Papers/boerger
Local file: papers/asmintro.pdf
1999

From the document:

> *We provide an introduction to a practical method for rigorous system development which has been used successfully, under industrial constraints, for design and analysis of complex hardware/software systems. The method allows one to start system development with a trustworthy high level system specification and to link such a "ground model" in a well documented and inspectable way through intermediate design steps to its implementation. The method enhances traditional operational modelling and analysis techniques by incorporating the most general abstraction, decomposition and refinement mechanisms which have become available through Gurevich's Abstract State Machines. Through its versatility the ASM approach is non-monolithic and integratable at any development level into current design and analysis environments. We also collect experimental evidence for the ASM thesis, a generalization of Turing's thesis.*

Abstract State Machines at the Cusp of the Millenium
Egon Börger (boerger@di.unipi.it)
http://www.di.unipi.it/~boerger/Papers/Methodology
Local file: papers/asm2000intro.pdf
2000

From the document:

> *We went a long way since the Spring of 1987 when Yuri Gurevich visited Pisa and, in a series of lectures on the fundamental problem of semantics of programming languages, presented the world premiere of the concept of ASMs. He gave the main motivation: reconsider Turing's thesis in the light of the problem of semantics of programs. He illustrated his ideas with examples, in particular specifications of Turing machines, stack machines and some Pascal programs. He gave also proofs of simple properties of these programs. This material appeared a year later in [22]. It was preceded by the first appearance of the ASM Thesis, in embryo in a 1984 technical report [20], and fully spelled out in a notice presented on May 13 of 1985 to the American Mathematical Society [21]. It was accompanied by the first real-world application, namely the dynamic semantics of MODULA-2 [26], and shortly afterwards followed by the ASM treatment of concurrency used to define the semantics of OCCAM [27], which was presented by Gurevich in another series of lectures in Pisa in May 1990. Since then the concept of Abstract State Machines essentially remained stable [23, 24] and triggered hundreds of publications in various domains including finite model theory, complexity theory and numerous areas of applied computer science, in particular programming languages, database query languages, protocols, architectures and embedded control software [1].*

Logician in the Land of OS: Abstract State Machines in Microsoft
Yuri Gurevich (gurevich@microsoft.com)
http://research.microsoft.com/~gurevich/Opera
Local file: papers/151.pdf
2001

From the document:

> *Analysis of foundational problems like "What is computation?" leads to a sketch of the paradigm of abstract state machines (ASMs). This is followed by a brief discussion on ASMs applications. Then we present some theoretical problems that bridge between the traditional LICS themes and abstract state machines.*

3.4.4 Sequential Abstract State Machines Thesis

This paper presents the Sequential Abstract State Machines thesis: *every sequential algorithm can be step-for-step simulated by an appropriate sequential ASM*. This is a fundamental result for the usability of ASMs as a specification language.

Sequential Abstract State Machines Capture Sequential Algorithms
Yuri Gurevich (gurevich@microsoft.com)
http://delivery.acm.org/10.1145/350000/343384
Local file: papers/p77-gurevich.pdf
2000

From the document:

> *We examine sequential algorithms and formulate a sequential-time postulate, an abstract-state postulate, and a bounded-exploration postulate. Analysis of the postulates leads us to the notion of sequential abstract state machine and to the theorem in the title. First we treat sequential algorithms that are deterministic and noninteractive. Then we consider sequential algorithms that may be nondeterministic and that may interact with their environments. Categories and Subject Descriptors: F.1.1 [Computation by abstract devices]: Models of Computation; I.6.5 [Simulation and Modelling]: Model Development – Modelling methodologies.*

3.4.5 Parallel Abstract State Machines Thesis

This paper presents the Parallel Abstract State Machines thesis: *every parallel algorithm can be step-for-step simulated by an appropriate parallel ASM*. This is a fundamental result for the usability of ASMs as a specification language.

Abstract State Machines Capture Parallel Algorithms
Yuri Gurevich (gurevich@microsoft.com)
Andreas Blass (ablass@umich.edu)
http://research.microsoft.com/~gurevich/annotated.html#157
Local file: papers/157.pdf
2002

From the document:

> *We give an axiomatic description of parallel, synchronous algorithms. Our main result is that every such algorithm can be simulated, step for step, by an abstract state machine with a background that provides for multisets.*

3.4.6 Complex Systems and Abstract State Machines

These papers are very good examples of ASMs being used as a specification language for complex systems. The authors define these systems in a very complete yet clear way. It shows the undeniable advantage of using formal methods to specify complex systems in order to find inconsistencies before they are deployed.

Modelling and Analysis of Distributed and Reactive Systems using Evolving Algebras
Egon Börger (boerger@di.unipi.it)
Uwe Glässer (glaesser@upb.de)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/pvm.pdf
1995

From the document:

> *This is a tutorial introduction into the evolving algebra approach to design and verification of complex computing systems. It is written to be used by the working computer scientist. We explain the salient features of the methodology by showing how one can develop from scratch an easily understandable and transparent evolving algebra model for PVM, the widespread virtual architecture for heterogeneous distributed computing.*

A Description of the Tableau Method Using Abstract State Machines
Egon Börger (boerger@di.unipi.it)
Peter H. Schmitt (pschmitt@ira.uka.de)
http://www.eecs.umich.edu/gasm
Local file: papers/tableau.pdf
1997

From the document:

> *Starting from the formulation of the tableau calculus as it is presented in the textbooks we give an operational description of the tableau method in terms of abstract state machines at various levels of refinement ending after four stages at a specification that is very close to the leanTAP implementation of the tableau calculus in PROLOG. Proofs of correctness and completeness of the refinement steps are given.*

Initialization Problems for Java
Egon Börger (boerger@di.unipi.it)
Wolframm Schulte (wolfram@informatik.uni-ulm.de)
http://www.kettering.edu/~jhuggins/papers
Local file: papers/javainit.pdf
1999

From the document:

> *We exhibit a grey area in the specification of Java and of its implementation through the Java Virtual Machine (JVM): the treatment of initialization of classes and interfaces. We report the result of our experiments with different implementations of Java, which confirm the theoretical prediction of our work on mathematical models for Java [4] and the JVM [3], namely that the designers of Java and the JVM have used notions of initialization which do not match and which aflict the portability of Java programs. We show also that concurrent initialization may deadlock and that various current Java compilers violate the initialization semantics through standard optimization techniques.*

Investigating Java Concurrency Using Abstract State Machines
Yuri Gurevich (gurevich@microsoft.com)
Wolfram Schulte (wolfram@informatik.uni-ulm.de)
Charles Wallace (wallace@mtu.edu)
http://research.microsoft.com/users/schulte
Local file: papers/Investigating-Java-Concurrency.pdf
2000

From the document:

> *We present a mathematically precise, platform-independent model of Java concurrency using the Abstract State Machine method. We cover all aspects of Java threads and synchronization, gradually adding details to the model in a series of steps. We motivate and explain each concurrency feature, and point out subtleties, inconsistencies and ambiguities in the official, informal Java specification.*

An Abstract Communication Model
Yuri Gurevich (gurevich@microsoft.com)

Uwe Glässer (glaesser@upb.de)
Margus Veanes (veanes@microsoft.com)
http://research.microsoft.com/~gurevich/Opera
Local file: papers/tr-2002-55.pdf
2002

From the document:

> *We present an abstract communication model. The model is quite general even though it was developed in the process of specifying a particular network architecture, namely the Universal Plug and Play (UPnP) architecture. The generality of the model has been confirmed by its reuse for different architectures. The model is based on distributed abstract state machines and implemented in the specification language AsmL.*

3.4.7 Decidability and Abstract State Machines

These two papers are on a technique to generate Finite State Machines from Abstract State Machines. Of course, this result, even though it is currently limited, is of great interest for anyone who wants to verify properties on ASMs using finite state verification technologies like model checking.

Generating Finite State Machines from Abstract State Machines
Yuri Gurevich (gurevich@microsoft.com)
Wolfram Schulte (wolfram@informatik.uni-ulm.de)
Margus Veanes (margus@microsoft.com)
Wolfgang Grieskamp (wrwg@microsoft.com)
ftp://ftp.research.microsoft.com/pub/tr
Local file: papers/tr-2001-97.pdf
2001

From the document:

> *We give an algorithm that derives a finite state machine (FSM) from a given abstract state machine (ASM) specification. This allows us to integrate ASM specs with the existing tools for test case generation from FSMs. ASM specs are executable but have typically too many, often infinitely many states. We group ASM states into finitely many hyperstates which are the nodes of the FSM. The links of the FSM are induced by the ASM state transitions.*

Testing with Abstract State Machines
Yuri Gurevich (gurevich@microsoft.com)
Wolfram Schulte (wolfram@informatik.uni-ulm.de)
Margus Veanes (margus@microsoft.com)
Wolfgang Grieskamp (wrwg@microsoft.com)
ftp://ftp.research.microsoft.com/pub/tr
Local file: papers/extabstract.pdf
2001

From the document:

> *Recently Egon Börger wrote: "If we succeed to exploit ASMs for defining and implementing methods for generating test suites from high level specifications, this will turn a dark and at present overwhelming part of software development into an intellectually challenging and methodologically well supported task of enormous practical value" [1, New Frontiers]. We had to start working on this challenge to solve a problem that arose at Microsoft.*
>
> *In general, testing is used in the software industry to find bugs. Here we are concerned with conformance testing. A bug occurs when the implementation does not conform to the specification, assuming that the specification is correct. In our case, the specification is a nondeterministic ASM program with a fixed initial state. The program may have infinitely many states. To this end, we group the ASM states into finitely many hyperstates. This way we create a finite automaton which is used to generate a test suite. In general, you don't have the time to explore all possible runs of the ASM program. But full branch coverage is ensured in most cases: the suite involves some invocation of every command in the specification program (in general, the problem of determining if the guard of a specification branch will become true is hard).*
>
> *As far as the implementation is concerned, we require only that it provides enough observables to judge the conformance of the current implementation state with the current specification state. We assume a reliable "status method" that is used to retrieve the observable at any state of the device.*

3.4.8 Model Checking and Abstract State Machines

These papers are on model checking Abstract State Machines. They constitute unavoidable references in the current research context since they apply ASM theory to the automatic verification of security properties (see Annex C, Subsection 3.4.10, and Section 4).

Abstract State Machines: Verification Problems and Complexity
Marc Spielmann (spielmann@i7.informatik.rwth-aachen.de)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/diss00.pdf
2000

From the document:

> *Abstract state machines (ASMs) provide the formal foundation for a successful methodology for specification and verification of complex dynamic systems. In addition, ASMs induce a computation model on structures, which, in some sense, is more powerful and universal than the standard computation models in theoretical computer science. An investigation of ASMs is therefore interesting from both the point of view of applied computer science and the point of view of theoretical computer science. In the present thesis, practically relevant as well as theoretically motivated questions concerning ASMs are investigated. Subject of the first part of the thesis is the automatic verifiability of ASM specifications. In the second part, the ASM computation model itself and choiceless complexity classes, which have recently been defined by means of ASMs, are discussed.*

Model Checking Abstract State Machines and Beyond
Marc Spielmann (spielmann@i7.informatik.rwth-aachen.de)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/autoverif.pdf
2000

From the document:

> *We propose a systematic investigation of the (semi-)automatic verifiability of ASMs. As a first step, we put forward two verification problems concerning the correctness of ASMs and investigate the decidability and complexity of both problems.*

Model Checking Abstract State Machines
Kirsten Winter (kirsten@svrc.uq.edu.au)
http://edocs.tu-berlin.de/diss/2001
Local file: papers/winterkirsten.pdf
2001

From the document:

> *Abstract State Machines (ASM) constitute a high-level specification language for a wide range of applications. The existing tool support for ASM includes editors, type-checkers, and simulators. Moreover, ASM are encoded into the logic of two theorem provers that provide support for mechanical interactive proofs. This thesis aims to extend this tool support by means of a fully algorithmic approach, namely model checking. We use the ASM Workbench as a core tool framework for ASM and provide a general interface into an intermediate language. This general interface can be exploited for interfacing various tools that allow the treatment of simple transition systems. With this transformation step we bridge a gap between the high-level modelling language ASM and the low-level input language of model checkers. Based on this general interface, we develop two interfaces: from the ASM Workbench to the model checker SMV and to the MDG-Package. The former is a widely used model checker that implements symbolic model checking for CTL formulas. We implement the interface and investigate two case studies by means of our transformation and the SMV tool.*
>
> *The results illustrate the practicability of this approach as well as its limitations. These limitations motivate the second interface to the MDG-Package. This package comprises the basic functionality for symbolic model checking based on multiway decision graphs (MDGs). This graph structure allows the representation of possibly infinite system through abstract data types and functions. Thus, it provides a simple means for supporting abstraction of complex systems. Moreover, in this framework assumptions on the system environment can be modelled as temporal logic formulas which are conjoined with the system model. This extends the limits of this fully automatic analysis provided by model checking.*

3.4.9 Abstract State Machine Extensions

The following papers define extensions to standard ASMs. They empower ASMs with the capability to model recursive, object-oriented, concurrent, distributed, and reactive algorithms, for instance.

Recursive Abstract State Machines
Yuri Gurevich (gurevich@microsoft.com)
Marc Spielmann (spielmann@i7.informatik.rwth-aachen.de)
http://www.eecs.umich.edu/gasm
Local file: papers/recursion.pdf

1997

From the document:

> *According to the ASM thesis, any algorithm is essentially a Gurevich abstract state machine. The only objection to this thesis, at least in its sequential version, has been that ASMs do not capture recursion properly. To this end, we suggest recursive ASMs.*

Background, Reserve, and Gandy Machines
Yuri Gurevich (gurevich@microsoft.com)
Andreas Blass (ablass@umich.edu)
http://research.microsoft.com/~gurevich/Opera
Local file: papers/143.pdf
2000

From the document:

> *Algorithms often need to increase their working space, and it may be convenient to pretend that the additional space was really there all along but was not previously used. In particular, abstract state machines have, by definition [103], an infinite reserve. Although the reserve is a naked set, it is often desirable to have some external structure over it. For example, in [120] every state was required to include all finite sets of its atoms, all finite sets of these, etc. In this connection, we define the notion of a background class of structures. Such a class of structures specifies the constructions (like finite sets or lists) available as background for algorithms.*
>
> *The importation of reserve elements must be non-deterministic, since an algorithm has no way to distinguish one reserve element from another. But this sort of non-determinism is much more benign than general non-determinism. We capture this intuition with the notion of inessential non-determinism. Alternatively, one could insist on specifying a particular one of the available reserve elements to be imported. This is the approach used in [Robin Gandy, Church's thesis and principles for mechanisms in: The Kleene Symposium (Ed. Jon Barwise et al.), North-Holland, 1980, 123–148.]. The price of this insistence is that the specification cannot be algorithmic. We show how to turn a Gandy-style deterministic, non-algorithmic process into a non-deterministic algorithm of the sort described above, and we prove that Gandy's notion of "structural" for his processes corresponds to our notion of inessential non-determinism.*

Rich Sequential-Time ASMs
Yuri Gurevich (gurevich@microsoft.com)
Wolfram Schulte (wolfram@informatik.uni-ulm.de)
Margus Veanes (margus@microsoft.com)
http://research.microsoft.com/foundations/ASM2001
Local file: papers/gsv.pdf
2000

From the document:

> *Software industry can use a good specification language. This specification language should be executable, and so ASMs become relevant. The language should allow one to integrate specifications with existing technologies. To meet this need, our group in Microsoft Research builds a powerful extension of the original ASMs. We call it ASML (for ASM Language). ASML has a rich type system, is object oriented, and is being integrated with Microsoft programming environment. Here we are not concerned with the precise syntax of ASML. We view the original ASMs as mathematical objects and we extend the theory of original ASMs to provide a solid semantic foundation for ASML. The explicitly distributed version of ASML is yet to be implemented; accordingly we deal only with sequential-time computing in this paper.*

Composition and Submachine Concepts for Sequential ASMs
Egon Börger (boerger@di.unipi.it)
Joachim Schmid (Joachim.Schmid@tydo.de)
ftp://www.eecs.umich.edu/groups/gasm
Local file: papers/structuring.pdf
2000

From the document:

> *We define three composition and structuring concepts which reflect frequently used refinements of ASMs and integrate standard structuring constructs into the global state based parallel ASM view of computations. First we provide an operator which combines the atomic update view of ASMs with sequential machine execution and naturally incorporates classical iteration constructs into ASMs. For structuring large machines we define their parameterization, leading to a notion of possibly recursive submachine calls which sticks to the bare logical minimum needed for sequential*

ASMs, namely consistency of simultaneous machine operations. For encapsulation and state hiding we provide ASMs with local state, return values and error handling.

Some of these structuring constructs have been implemented in ASM-Gofer. We provide also a proof-theoretic definition which supports the use of common structured proof principles for proving properties for complex machines in terms of properties of their components.

Toward Industrial Strength Abstract State Machines
Yuri Gurevich (gurevich@microsoft.com)
Wolfram Schulte (wolfram@informatik.uni-ulm.de)
Margus Veanes (margus@microsoft.com)
ftp://ftp.research.microsoft.com/pub/tr
Local file: papers/tr-2001-98.pdf
2001

From the document:

A powerful practical ASM language called AsmL is being developed in Microsoft Research by the group on Foundations of Software Engineering. AsmL extends the language of original ASMs in a number of directions. This paper describes some of those directions.

Partial Updates: Exploration
Yuri Gurevich (gurevich@microsoft.com)
Nikolai Tillmann
http://research.microsoft.com/~gurevich/Opera
Local file: papers/156.pdf
2002

From the document:

The partial update problem for parallel abstract state machines has manifested itself in the cases of counters, sets and maps. We propose a solution of the problem that lends itself to a sufficient implementation and covers the three cases mentioned above. There are other cases of the problem that require a more general framework.

Partial Updates Exploration II
Yuri Gurevich (gurevich@microsoft.com)
Nikolai Tillmann
http://research.microsoft.com/~gurevich/annotated.html#161
Local file: papers/161.pdf
2003

From the document:

> *During one step of a powerful programming, specification or query language, the same mathematical object, like a set or map or sequence, may be modified – in parallel – by different parts of the program. Such partial updates need to be efficiently checked for consistency and integrated. This is the problem of partial updates in a nutshell. In our first paper on the subject we proposed a general solution of the partial-update problem for abstract state machines (ASMs) where the problem is exacerbated by the use of nested submachines; in particular we solved the problem in the cases of counters, sets and maps. Here we propose a more general framework that allows us to solve the problem for sequences and labeled ordered trees. The partial update problem for sequences is related to the problem of collaborative editing.*

3.4.10 A Few Abstract State Machine Applications

Abstract State Machines are not only useful in theory – they also have many practical applications. There is no better way to be convinced of this statement than to visit *the* online reference on ASMs: the Abstract State Machines online bibliography [14]. One can also read [16] for a complete state of the art on ASMs. Among others, there are papers on hardware and software architectures specification and verification, compiler correctness analysis, database modeling, distributed system modeling, interpreters, Java language, compiler and virtual machine formalization, programming language semantics, protocol verification, and software engineering.

The Foundations of Software Engineering Group at Microsoft Research has also developed an implementation of ASM called AsmL. Gurevich, the inventor of ASM, is the leader of this group. Further information can be found at http://research.microsoft.com/foundations/.

Robert Stärk, Joachim Schmid, and Egon Börger are the authors of a very interesting book on the formalization of Java and the Java virtual machine using ASMs [17]. They have formally defined the operational semantics of the Java language

and the Java virtual machine. They have also very elegantly modeled a Java compiler and a virtual machine (including the bytecode verifier) that have been proven to be correct. Many theorems have demonstrated the correctness of their models. Finally, these models are all executable due to the dynamic nature of ASMs.

Another interesting project is the eXtensible Abstract State Machines (XASM) project. Its main objective is to develop a modern programming language whose semantics is based on ASMs. For further details, please consult http://www.xasm.org/.

Finally, an international workshop on ASMs has been in existence since 1994. Information from the 2003 edition can be found at http://www.dmi.unict.it/asm2003/.

Therefore, since many practical and industrial ASM applications have been developed, it is possible to say that ASMs are not only useful in theory but also in practice. Siemens was the first large company to use ASMs, and since 1998, Microsoft Research is pursuing an intensive effort towards the adoption of ASMs in .NET applications development [18]. Many bugs have already been (manually) found in different technologies by formalizing them using ASMs.

For further information on model-checking ASMs, please read Annex C.

4 Future Work – A Novel Hybrid Model-Checking Approach

This section describes the project: a novel hybrid model-checking approach. As it has already been mentioned, this approach is *hybrid* in the sense that it combines static and dynamic analysis principles in order to provide more precise model checking. Models of this new model-checking approach will be based on ASM theory.

4.1 Context

The hybrid model-checking approach is to be integrated into a broader security architecture [19] (see Figure 2) that has been developed from the experience gained in the MaliCOTS project (see Subsection 2.1). Indeed, one result of the MaliCOTS project was the raising of an interesting and fundamental question: “How can verification technologies cooperate so as to provide more efficient, flexible, and robust security?”.

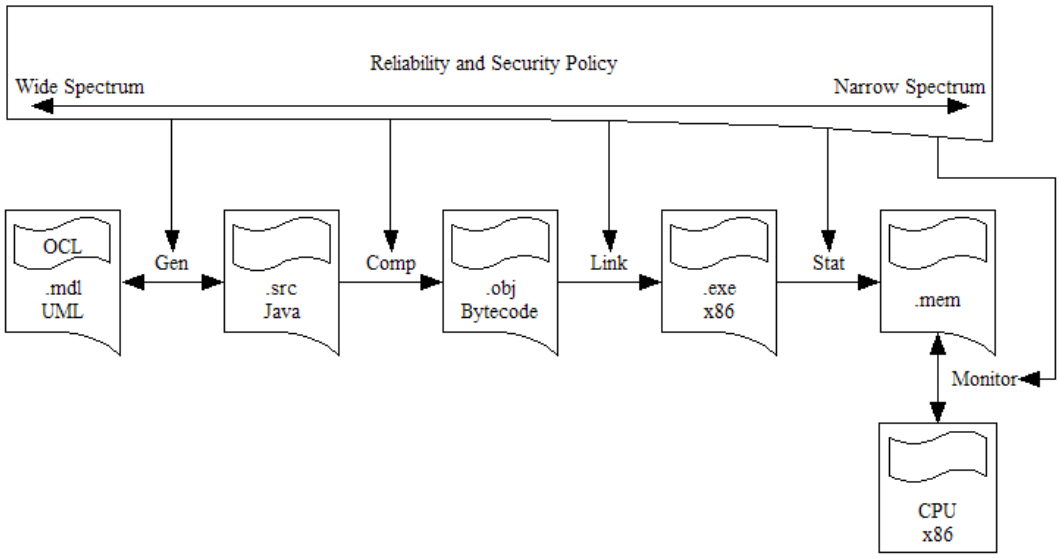

Figure 2: *Security architecture*

In the MaliCOTS project, dynamic analysis was considered as a front-line security certifier. In actual fact, dynamic analysis should be at the end of the certification chain. Therefore, the solution to the verification technology cooperation challenge is to establish a secure development chain from a secure Unified Modelling Language (UML) design to secure monitored binary execution [19] (as illustrated in Figure 2). All computational models used in this secure development chain can be based on ASM theory.

This ambitious research project thus aims at developing a UML design certification technique [20] that uses the Object Constraint Language (OCL) as a property specification language. A UML/OCL model checker is currently in development. It will ensure that security concerns are already considered at the design phase of the development cycle of secure applications. This is of paramount importance in order to start managing risk at the highest possible level of abstraction so that errors can be discovered as early as possible in the software development cycle.

These certified UML diagrams will then form the basis of a Java implementation skeleton associated with certificates containing the security properties that could not be verified within the UML models. These certificates thus contain the residual risk that must be managed at the implementation level (or perhaps even later, at the execution level). The implementation's source code and possible additional properties will then be processed by a Java certifying compiler. Once again, this certifying compiler will delegate the residual risk to the next line of certification: the Java hybrid model checker, which is the subject of this project. This model checker will

be a traditional model checker except for its capacity to integrate new information in the software package's model in order to make the analysis more precise. Ideas about how this integration could be performed are presented in Subsection 4.2.

Finally, residual risks will be delegated to the final line of defence: a Java execution monitor. Since most of the properties should have already been verified at higher levels in the secure development cycle, this monitor should simply have to perform the remaining checks that can only be handled dynamically, thus reducing complexity and overhead.

Of course, this research project is not without prospective challenges. Some interesting difficulties have already been identified: "Is it possible to automatically identify and delegate unverified properties and, if so, how? Ideally, the specification language should be the same for every certification level, but then, what is the right choice for such a specification language? How could modularization and composition be managed in the certification chain? Can one make the different tools optional? For example, if somebody is not interested in certifying compilation, could all the residual risk emerging from the design phase be delegated to the hybrid model checker?", etc.

4.2 A Few Hybrid Model-Checking Approaches

Currently, there are a few potential methods of integrating additional dynamic information in the software package's model during model checking, which give a few different hybrid model-checking approaches. They have been given the following names: *interactive*, *parameterized*, *test-based*, and *worst-case hybrid model checking*. For the sake of completeness, a *statically-supported dynamic analysis* has also been hypothesized. The ideas behind all these approaches are given in the following subsections.

4.2.1 Interactive Hybrid Model Checking

Interactive hybrid model checking aims at posing appropriate questions to the analyst during model checking. This could also be referred to as *semi-automatic* hybrid model checking. These questions would focus on getting information that is not statically present in the model.

For example, suppose that a file open is modeled but the file name and location are not known at model-checking time. The traditional way of coping with this lack of information is to abstract it by some means, such as using variables, for instance. However, it has been observed, during past projects (see Section 2), that

this solution rapidly becomes highly complex and very difficult to manage. Moreover, it leads to imprecise results in many situations. For instance, a property could be stated to be `false` when in fact, it is `true` most of the time. These are called *conservative* results. They are inevitable in general but techniques to improve their precision are always welcome. Therefore, interactive hybrid model checking would cope with this situation by asking the analyst to enter the name and location of a file of interest in order to more precisely verify what happens to it afterwards. For example, the analyst could choose a file that he knows to be confidential in order to determine if the information contained in this file is dealt with in a non-secure manner (with respect to a predefined security policy, i.e., a predefined property). The hybrid model checker could memorize the various choices of the analyst and create a verification script that would be useful for a future verification.

By "posing appropriate questions to the analyst", it is not meant that questions should be systematically asked each time an action is statically incomplete. In fact, an analysis should be performed to determine which incomplete actions should be completed as a priority, depending on the benefits of completing them. It is believed, at this stage, that this analysis could be based on traditional dataflow analysis techniques.

In other words, questions should be asked optimally, in terms of the anticipated benefits for the analysis and the minimal annoyance for the analyst. This makes an interesting challenge to this approach.

In order to investigate this new concept, models of verified programs could firstly be based on automata theory. This theory is well understood and thus, ideal to get a deeper understanding of new analyses. ASMs could then be integrated, once more advanced practical implementations based on the theoretical results are achieved. To build upon past projects (refer to Section 2), verified properties would be expressed in one of the branching temporal logics (CTL, CTL*, FBTL, . . .).

It could be interesting to find solutions for the following inter-related problems:

1. How can statically incomplete actions be prioritized in order to optimize which ones are needed to be completed first with respect to the property to be verified?
2. How can we identify the smallest set of incomplete actions for which completion would lead to the verification of the property?
3. How could information be added into the model without interrupting or disturbing the running model checking process?
4. How could the model be abstracted after getting more information in order to facilitate the rest of the current verification process with respect to the verified property or the class of this property?

4.2.2 Parameterized Hybrid Model Checking

Parameterized hybrid model checking targets a more automatic hybrid model-checking approach than interactive hybrid model checking. Indeed, in this approach, the analyst is expected to define resource domains (for example, confidential files, prohibited network usage, etc.) that the hybrid model checker will use to populate missing information in appropriate incomplete actions (hence the name parameterized hybrid model checking).

Therefore, this approach is essentially based on interactive hybrid model checking except that it has a higher degree of independence from the analyst. It could be interesting for analysts with less background in formal program verification or in contexts where the analyst who uses the model checker is not the same as the one who determines what has to be verified, i.e., the properties and the critical resources. The same problems would thus also have to find solutions.

4.2.3 Test-Based Hybrid Model Checking

A slight addition to parameterized hybrid model checking would be *test-based* hybrid model checking. Indeed, the latter would consist in making the analyst produce a set of test cases that would be automatically verified by the model checker. In this context, resource domains would also be defined (like for parameterized hybrid model checking) but in addition, expected results with respect to particular inputs in the resource domains would also be specified. The goal of the model checker would then be to verify that the properties are valid or not but also to ensure that the results are the expected ones. This, of course, would necessitate very precise information from the analyst. However, it could become a test strategy based on formal methods. This should be very interesting.

4.2.4 Worst-Case Hybrid Model Checking

Worst-case hybrid model checking would be an effort towards asking as few inputs as possible of the analyst. Therefore, the analyst would not have to answer interactive questions or define resource domains and test cases in addition to the properties to verify. The model checking process would rather use a powerful static analysis in order to determine the worst-case scenarios, i.e., the most conservative and critical ones, and would then inform the analyst of its results and conclusions in a way that he could assess the potential damage of the verified program on his system with respect to the properties he has defined. Such verification would certainly be interesting and appropriate in situations when a preliminary investigation must be conducted on a small to medium-sized piece of software.

4.2.5 Statically-Supported Dynamic Analysis

The hybrid model checking approaches presented in the previous subsections were all backed by static analysis (usually, dataflow analysis was anticipated) in order to go a little further into the proof of the satisfiability of the properties to be verified. The following dynamic analysis approach is also intensively supported by static analysis, that is, its dynamic analysis algorithm becomes intertwined with static analysis (once again, the static analysis used is anticipated to predominantly be dataflow analysis). This approach has been given the name *statically-supported dynamic analysis*.

It essentially consists of the following:

1. Perform a pre-execution static analysis and then, during execution:
   (a) Loop
      i. Detect and memorize information that can make the static analysis progress
      ii. Repeat the static analysis with the additional information

It would be interesting to investigate how the results of the static analysis could be represented in order to facilitate the dynamic analysis at runtime.

Moreover, and most importantly, such a statically-supported dynamic analysis could potentially detect that a property is necessarily going to be violated and react earlier than it would have been possible without statically-collected information. Therefore, statically-supported dynamic analysis would hypothetically not suffer from too-late diagnosis, like it can be the case for traditional dynamic analysis.

This last approach is thus another way of thinking about a “hybrid” analysis. Indeed, on one hand, the approaches based on model checking are static analysis augmented by dynamic analysis principles. On the other hand, this last approach is dynamic analysis augmented by static analysis principles. This last approach is currently investigated by, at least, one fellow researcher: Klaus Havelund from the Automated Software Engineering Group at NASA Ames Research Center (for a list of his publications, see http://ase.arc.nasa.gov/havelund/). However, we are not currently aware of any researcher working in the first four model-checking approaches.

Because both philosophies imply high complexity, we are going to cover only one of them in this effort, that is, the first one, the one related to the augmented model-checking approaches.

4.3 Architecture

The architecture of the hybrid model checker is envisioned to follow these guidelines (refer to Figure 3).

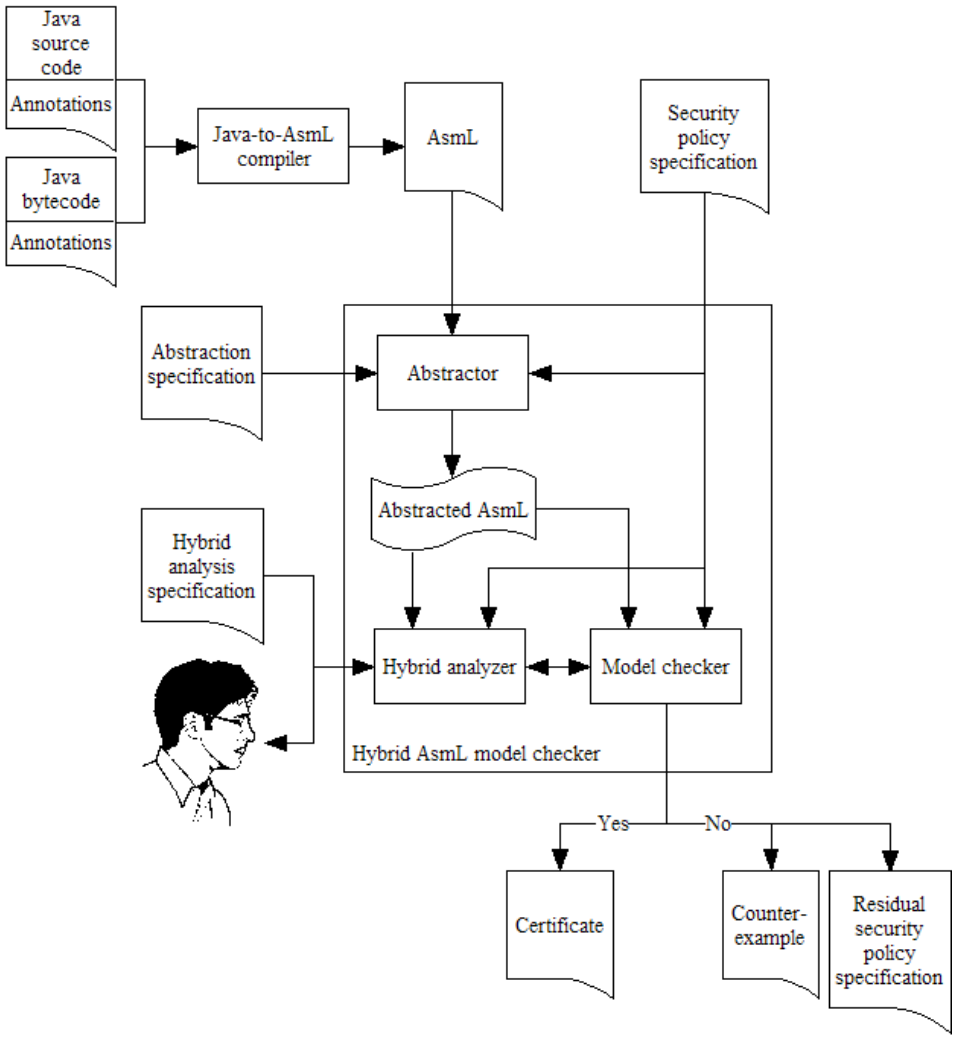

Figure 3: *Architecture of the proposal*

First, the system modeling language used will be AsmL (for ASM Language) [21]. AsmL is a general-purpose specification language based on ASMs and developed at Microsoft Research. It is used for creating human-readable, machine-executable models of a system's operation in a way that is minimal and complete with respect to any given user-defined level of abstraction. It has already been used to model large-sized and complex systems like, for instance, Microsoft's Universal Plug and Play. Also, ASMs have been used to specify the semantics of the Java language, the Java compiler, and the Java Virtual Machine [17].

Moreover, specifications written in AsmL are formal and executable, which is a plus when dealing with dynamic analysis principles. Like informal specifications, formal executable specifications are descriptions of how software components work. Unlike informal specifications, formal specifications have a single, unambiguous meaning. In AsmL, this meaning comes in the form of an abstract state machine (ASM), which is a mathematical model of the system's evolving, runtime state. Being executable, AsmL specifications may be run as a program, for instance, to simulate how a particular system will behave or to check the behavior of an implementation against its specification. However, unlike traditional programs or implementations, executable specifications are intended to be minimal. In other words, although they are faithful in describing, without omission, everything that is part of the chosen level of detail, they are equally faithful in leaving unspecified what is outside that level of detail. Thus, unlike programs, executable specifications restrict themselves to the constraints and behavior that all correct implementations of the system should have in common. In other words, an executable specification must be as clear about the freedom given to correct implementations of the system it describes as it is about constraints. For example, executable specifications do not constrain the order of operations unless it is significant, whereas current-day programs realize a sequential order of operation as an implementation decision.

Also, an e-mail communication with Dr Yuri Gurevich, the AsmL group leader at Microsoft Research, indicated that he is particularly interested in model-checking AsmL. This could lead to an interesting collaboration.

Second, a security policy specification language will have to be chosen or developed. Because of the hands-on experience gained during the MaliCOTS and JACC projects, a security policy specification language based on the modal μ-calculus will probably be used. It can also be based on FBTL (see Subsection C.2) or another branching temporal logic, such as CTL*. In all cases, an extension to the logic will have to be developed in order to consider data in logic formulae. For example, the capacity to specify or abstract file names is needed. Such an extension to the modal μ-calculus has been developed in the JACC project, so it may also be reused.

Third, a compiler from Java source code or Java bytecode to AsmL will be developed and used in order to model check Java with the AsmL model checker. Translating Java source code instead of Java bytecode would be simpler because Java source code is more similar to AsmL than Java bytecode and also because Java source code contains more control flow-related information (i.e., explicit loops, explicit exception handling, etc.). However, translating Java bytecode instead of Java source code would be more practical because it is the format used to transfer Java programs. It is therefore always available. A design decision will thus have to be taken on this matter. The Java source code or Java bytecode files may be augmented by annotations produced by the UML design certifier or the Java certifying

compiler, respectively (please refer to Subsection 4.1). The translation from Java to AsmL should be as faithful as possible to the original Java program because this first model is most probably going to be abstracted, so it needs to be accurate.

Finally, at least one hybrid analysis approach will be formalized and integrated in this model checker.

5 Conclusion

In this document, there has been an overview of a novel hybrid model-checking approach based on ASM theory that is integrated into a continuous certification chain within the software development cycle. It has been mentioned that this approach is *hybrid* in the sense that it combines static and dynamic analysis principles in order to provide more precise model checking. This certification chain avoids the limitations of the MaliCOTS project by dividing risk management into modular certification tools. By delegating unverified properties to the next tool down the chain, certification is efficient, flexible, and robust and, equally importantly, residual risks can be assessed at the end of the certification chain.

Annex A
Abstract State Machines Overview

Reliance has been placed on the ASM computation model as defined in [22], which is based on the definitions given in [23], [24], and [25]. Spielmann's formalization was chosen because he has defined automatic verifiability theorems based on his ASM theory. A discussion of the theoretical foundations on which his theorems are based is presented in Subsection C.2. The interested reader is referred to [22] for more information.

A.1 Syntax

Before going into ASM syntax, a few definitions must be given.

Definition A.1 (Vocabulary) A *vocabulary* Υ (pronounced "capital upsilon") is a non-empty set of relation[3] and function[4] symbols (names). Each symbol in Υ is associated with a natural number, called the *arity* of the symbol.

Definition A.2 (Structure) Let Υ be a vocabulary. A *structure* $\mathcal{A}$ over Υ consists of a non-empty set A, called the *universe* of $\mathcal{A}$, an interpretation $R^{\mathcal{A}} \subseteq A^k$ for every k-ary relation symbol $R \in \Upsilon$, and an interpretation $f^{\mathcal{A}} : A^k \rightarrow A$ for every k-ary function symbol $f \in \Upsilon$. A *finite structure* is a structure whose universe is finite.

Definition A.3 (ASM Vocabulary) An *ASM vocabulary* Υ is a tuple $(\Upsilon_{in}, \Upsilon_{stat}, \Upsilon_{dyn}, \Upsilon_{out})$ of pairwise disjoint vocabularies. The symbols in Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{out} are called *input*, *static*, *dynamic*, and *output symbols*, respectively.

[3] A k-ary relation R between k sets $A_1, A_2, \ldots, A_k$ is a subset of $A_1 \times A_2 \times \ldots \times A_k$. It is noted $R \subseteq A_1 \times A_2 \times \ldots \times A_k$. $A_1 \times A_2 \times \ldots \times A_k$ can also be abbreviated by A_i^k, where $1 \leq i \leq k$. For example, if $A = \{1, 2\}$ and $B = \{1, 2, 3\}$, a possible *binary* relation R between A and B, noted $R \subseteq A \times B$, is $\{(1, 1), (1, 2), (2, 3)\}$.

[4] A k-ary function f is a $(k + 1)$-ary relation $f \subset A_1 \times A_2 \times \ldots \times A_k \times B$ such that *every* element of $A_1 \times A_2 \times \ldots \times A_k$ is paired with *exactly one* element of B. Formally, this constraint is specified as:

1. $(a_1, a_2, \ldots, a_k, b_1) \in f$ and $(a_1, a_2, \ldots, a_k, b_2) \in f$ implies that $b_1 = b_2$, and
2. for each $a_1 \in A_1, a_2 \in A_2, \ldots, a_k \in A_k$ there is some $b \in B$ such that $(a_1, a_2, \ldots, a_k, b) \in f$.

The k-ary function f is noted $f : A_1 \times A_2 \times \ldots \times A_k \rightarrow B$. $A_1 \times A_2 \times \ldots \times A_k$ can also be abbreviated by A_i^k, where $1 \leq i \leq k$. For example, if $A = \{1, 2\}$ and $B = \{1, 2, 3\}$, a possible *unary* function f, noted $f : A \rightarrow B$, is $\{(1, 1), (2, 3)\}$.

Definition A.4 (State) Let Υ be an ASM vocabulary. A *state* $\mathcal{S}$ over Υ is a structure over the vocabulary $\Upsilon = \Upsilon_{in} \cup \Upsilon_{stat} \cup \Upsilon_{dyn} \cup \Upsilon_{out}$.

The ASM syntax can now be defined.

Definition A.5 (ASM Programs – Syntax) Let Υ be an ASM vocabulary. A *guard* φ is a first-order formula over $\Upsilon - \Upsilon_{out}$. *ASM programs* are inductively defined as follows:

•**Update:** For every relation symbol $R \in \Upsilon_{dyn} \cup \Upsilon_{out}$, every function symbol $f \in \Upsilon_{dyn} \cup \Upsilon_{out}$, and all terms $\bar{t}, t_0$ over $\Upsilon - \Upsilon_{out}$, each of the following is an ASM program:

$$R(\bar{t}),$$

$$\neg R(\bar{t}), \text{and}$$

$$f(\bar{t}) := t_0.$$

Such ASM programs are also called *atomic ASM programs*. The notation $R(\bar{t})$ means that $\mathcal{S} \models R(\bar{t})$ after applying the update $R(\bar{t})$ on some state $\mathcal{S}$. The notation $\neg R(\bar{t})$ is similar except that $\mathcal{S} \models \neg R(\bar{t})$. The notation $f(\bar{t}) := t_0$ means that the location $\bar{t}$ of f is updated to t_0. In other words, if, for example, $f(\bar{t}) = 1$ and $t_0 = 2$, then $f(\bar{t}) = 2$ after applying the update $f(\bar{t}) := t_0$ on some state.

•**Conditional:** If Π is an ASM program and φ is a guard, then

$$(\texttt{if } \varphi \texttt{ then } \Pi)$$

is also an ASM program.

•**Parallel composition:** If Π_0 and Π_1 are ASM programs, then

$$(\Pi_0 \parallel \Pi_1)$$

is also an ASM program.

•**Parameterized parallel composition:** If Π is an ASM program, $\bar{x}$ is a tuple of pairwise distinct variables, and φ is a guard, then

$$(\texttt{do-for-all } \bar{x} : \varphi(\bar{x}), \Pi)$$

is also an ASM program. This rule is the generalization of the parallel composition rule. Variables are dynamic nullary functions in Υ.

•**Non-deterministic choice:** If Π is an ASM program, $\bar{x}$ is a tuple of pairwise distinct variables, and φ is a guard, then

$$(\texttt{choose } \bar{x} : \varphi(\bar{x}), \Pi)$$

is also an ASM program. Variables are dynamic nullary functions in Υ.

An ASM program is called *deterministic* if it does not contain a `choose`.

It is interesting to note that, in their basic definition, ASMs do not have *sequential* composition, but only *parallel* composition. There are two reasons behind this unconventional design decision:

1. Yuri Gurevich, the original designer of ASMs, wanted to keep his theory as compact and simple as possible. This is usually an objective of paramount importance when designing formal computational models.

2. Sequential composition can be achieved by using parallel composition and mutually exclusive guards. Therefore, theoretically speaking, sequential composition is not necessary when parallel composition and guards are available, which is the case in ASMs. In practice, however, achieving sequential composition with parallel composition and mutually exclusive guards is not convenient. This is why ASM is a theory and not a specification language. An example of an industrial specification language based on the ASM theory is AsmL, developed by Yuri Gurevich and his research team at Microsoft Research (see Subsections 3.4.10 and 4.3).

A.2 Semantics

Before going into ASM semantics, the following definition must be given.

Definition A.6 (Transition) An *update set* U over some ASM vocabulary Υ is a set of atomic ASM programs over Υ. Let $\mathcal{S}$ and $\mathcal{S}'$ be two states and let U be an update set, each over Υ. $\mathcal{S}'$ is called a successor state of $\mathcal{S}$ with respect to U if $\mathcal{S}'$ is identical to $\mathcal{S}$, except for the following modifications:

•if $R(\bar{t}) \in U$ and there is no update $\neg R(\bar{s}) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s})$, then $\mathcal{S}' \models R(\bar{t})$,

•if $\neg R(\bar{t}) \in U$ and there is no update $R(\bar{s}) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s})$, then $\mathcal{S}' \models \neg R(\bar{t})$, and

•if $(f(\bar{t}) := t_0) \in U$ and there is no update $(f(\bar{s}) := s_0) \in U$ such that $\mathcal{S} \models (\bar{t} = \bar{s}) \wedge (t_0 \neq s_0)$, then $\mathcal{S}' \models (f(\bar{t}) = t_0)$.

These rules ensure that the update set is *consistent*.

The ASM semantics can now be defined.

Definition A.7 (ASM Programs – Semantics) A state $\mathcal{S}$ is *appropriate* for Π if Π is an ASM program over the vocabulary of $\mathcal{S}$. Simultaneously for all states $\mathcal{S}$ appropriate for Π, a set $\text{Den}(\Pi, \mathcal{S})$ of update sets is defined by induction on the construction of Π as follows:

•**Update:** If Π is an atomic ASM program, then let

$$\text{Den}(\Pi, \mathcal{S}) := \{\{\Pi[\mathcal{S}]\}\},$$

where $\Pi[\mathcal{S}]$ is identical to Π except that the arguments of functions and relations in Π are replaced by their values evaluated in $\mathcal{S}$.

•**Conditional:** If

$$\Pi = (\texttt{if } \varphi \texttt{ then } \Pi_0),$$

then let

$$\text{Den}(\Pi, \mathcal{S}) := \begin{cases} \text{Den}(\Pi_0, \mathcal{S}) & \text{if } \mathcal{S} \models \varphi \\ \{\emptyset\} & \text{otherwise.} \end{cases}$$

•**Parallel composition:** If

$$\Pi = (\Pi_0 \parallel \Pi_1),$$

then let

$$\text{Den}(\Pi, \mathcal{S}) := \left\{ U_0 \cup U_1 : \begin{array}{l} U_0 \in \text{Den}(\Pi_0, \mathcal{S}), \\ U_1 \in \text{Den}(\Pi_1, \mathcal{S}) \end{array} \right\}.$$

•**Parameterized parallel composition:** If

$$\Pi = (\texttt{do-for-all } \bar{x} : \varphi(\bar{x}), \Pi_0),$$

then choose an index set I such that there exists a bijective mapping m from I to $\{\bar{a} : \mathcal{S} \models \varphi[\bar{a}]\}$. For each $i \in I$, set $\bar{a}_i = m(i)$ and let

$$\text{Den}(\Pi, \mathcal{S}) := \left\{ (\bigcup_{i \in I} U_i) : (\forall i : U_i \in \text{Den}(\Pi_0, (\mathcal{S}, \bar{a}_i))) \right\},$$

where $(\mathcal{S}, \bar{a}_i)$ denotes the state obtained from $\mathcal{S}$ by adding the j-th element in $\bar{a}_i$ as an interpretation of the j-th variable in $\bar{x}$, for all j. Remember that $\bar{x}$ is a tuple of pairwise distinct variables and that variables are dynamic nullary functions in Υ.

•**Non-deterministic choice:** If

$$\Pi = (\texttt{choose } \bar{x} : \varphi(\bar{x}), \Pi_0),$$

then let

$$\text{Den}(\Pi, \mathcal{S}) := \begin{cases} \bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \text{Den}(\Pi_0, (\mathcal{S}, \bar{a})) & \text{if } \varphi^{\mathcal{S}} \neq \emptyset \\ \{\emptyset\} & \text{otherwise,} \end{cases}$$

where $(\mathcal{S}, \bar{a})$ denotes the state obtained from $\mathcal{S}$ by adding the j-th element in $\bar{a}$ as an interpretation of the j-th variable in $\bar{x}$, for all j. Remember that $\bar{x}$ is a tuple of pairwise distinct variables and that variables are dynamic nullary functions in Υ. Note that every $\varphi(\bar{x})$ defines a k-ary query on $\text{Fin}(\Upsilon)$, the class of finite structures over Υ. This query maps any $\mathcal{S} \in \text{Fin}(\Upsilon)$ to the k-ary relation $\varphi^{\mathcal{S}} := \{\bar{a} \in A^k : \mathcal{S} \models \varphi[\bar{a}]\}$.

It is worth noticing that if Π is deterministic, then $\text{Den}(\Pi, \mathcal{S})$ is a singleton set.

It is important to note that ASMs stop when they reach a fixpoint, if and only if a fixpoint is eventually reached. Therefore, any ASM is conceptually integrated into a "loop until fixpoint" statement, i.e., any ASM program is continuously re-evaluated until there is no legal transition (see Definition A.6) from some state $\mathcal{S}$ to a successor state $\mathcal{S}'$.

Definition A.8 (ASM Program as a Transition Relation) The *transition relation* $Trans_\Pi$, implicitly defined by Π itself, is a binary relation between states. It contains a pair $(\mathcal{S}, \mathcal{S}')$ of states iff:

•both $\mathcal{S}$ and $\mathcal{S}'$ are appropriate for Π, and

•there exists an update set $U \in \text{Den}(\Pi, \mathcal{S})$ such that $\mathcal{S}'$ is a successor state of $\mathcal{S}$ with respect to U, as defined above (see Definition A.6).

Therefore, if $(\mathcal{S}, \mathcal{S}') \in Trans_\Pi$, then $\mathcal{S}'$ is also called a *successor state* of $\mathcal{S}$ with respect to Π. Once again, if Π is deterministic, then $Trans_\Pi$ is deterministic.

A.3 Abstract State Machine

Before a formal definition of an ASM can be given, a few other definitions are needed.

Definition A.9 (ASM Program Equivalence) Two ASM programs are *equivalent* if they define the same transition relation, i.e., $\Pi_0 \equiv \Pi_1$ iff $Trans_{\Pi_0} = Trans_{\Pi_1}$.

Definition A.10 (Input) An *input* $\mathcal{I}$ over an ASM vocabulary Υ is a finite structure over Υ_{in}.

Definition A.11 (Initialization Mapping) Let Υ be an ASM vocabulary. An *initialization mapping* over Υ is a function that maps every input $\mathcal{I}$ over Υ to a state $\mathcal{S}_\mathcal{I}$ over Υ and that satisfies the following conditions:

- the universe of $\mathcal{I}$ is a subset of the universe of $\mathcal{S}_\mathcal{I}$,
- for every relation symbol $R \in \Upsilon_{in}$, every k-ary function symbol $f \in \Upsilon_{in}$, and every k-tuple $\bar{a}$ of elements of $\mathcal{S}_\mathcal{I}$:

$$R^{\mathcal{S}_\mathcal{I}} = R^\mathcal{I}$$

and

$$f^{\mathcal{S}_\mathcal{I}}(\bar{a}) = \begin{cases} f^\mathcal{I}(\bar{a}) & \text{if } \bar{a} \text{ consists of elements of } \mathcal{I} \\ 0^\mathcal{I} & \text{otherwise,} \end{cases}$$

and

- for every input $\mathcal{I}'$ over Υ, if $\mathcal{I}'$ and $\mathcal{I}$ are isomorphic, $\mathcal{S}_{\mathcal{I}'}$ and $\mathcal{S}_\mathcal{I}$ are also isomorphic.

The formal definition of an ASM can now be given.

Definition A.12 (Abstract State Machine) An *abstract state machine* M is a triple $(\Upsilon, initial, \Pi)$, where Υ is an ASM vocabulary, $initial$ is an initialization mapping over Υ, and Π is an ASM program over Υ.

Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{out} are the *input*, *static*, *dynamic*, and *output vocabulary* of M, respectively. An *input* appropriate for M is an input over Υ. Finally, M is deterministic if Π is deterministic.

Definition A.13 (Run) Let $M = (\Upsilon, initial, \Pi)$ be an ASM and let $\mathcal{I}$ be an input appropriate for M. A *run* ρ of M on $\mathcal{I}$ is an infinite sequence $(\mathcal{S}_i)_{i \in \omega}$ of states over Υ such that $\mathcal{S}_0 = initial(\mathcal{I})$ and for every $i \in \omega$, $(\mathcal{S}_i, \mathcal{S}_{i+1}) \in Trans_\Pi$.

Annex B
Abstract State Machines Examples

This annex presents a few examples of ASM programs in order to illustrate their syntax and detail their semantics. These examples are written using no particular ASM language (for instance, AsmL (see Subsections 3.4.10 and 4.3) or ASM-SL ([26] and [27])). They are simply mathematically stated in order to faithfully illustrate the explanations given in Annex A.

B.1 Example #1 – In-Place Sorting

This ASM program models an in-place sorting algorithm, that is, a sorting algorithm that directly works in the collection to be sorted. This example illustrates that ASM semantics is highly parallel in nature and that ASMs can be used to model algorithms at a very high level of abstraction.

State

ELEMENTS	$=$	$\{\ldots\}$
INDICES	$=$	$\{0, 1, \ldots, n\}$, where $n \in \mathbb{N}$
get	:	INDICES $\rightarrow$ ELEMENTS
less_than_or_equal	:	ELEMENTS $\times$ ELEMENTS $\rightarrow$ Boolean
i	:	INDICES
j	:	INDICES

Program

(choose (i, j)	:	$i < j$
	$\wedge$	$\neg$less_than_or_equal(get(i), get(j))
	,	get(i) := get(j) $\parallel$ get(j) := get(i))

This algorithm is meant to be generic, that is, to specify an in-place sorting algorithm that is not directly dependant on the elements to be sorted. Therefore, it can conceptually sort naturals just as well as strings, for instance. There must only exists an order on the elements to be sorted. This is why the set ELEMENTS is not specified here (it must only be *finite*, see below). It can contain anything on which an order can be specified. This order is specified by the function named

less_than_or_equal, which takes two elements of ELEMENTS and returns `true` or `false` depending on whether the first element is less than or equal to the second.

INDICES is the set of indices that are used to index the elements of ELEMENTS. Therefore, the set INDICES contains naturals that range from 0 to $|\text{ELEMENTS}| - 1$, and thus, $|\text{ELEMENTS}| = |\text{INDICES}| = n + 1$. To ensure that the program terminates, ELEMENTS must be finite. The function named get takes an element of INDICES and returns the element of ELEMENTS that corresponds to the given index. It is therefore the indexing function of the set ELEMENTS. i and j are simply variables, i.e., dynamic nullary functions, of type INDICES used by the program.

The program is quite simple. Intuitively, it *non-deterministically* chooses *every* two indices i and j, for which $i < j$ and $\neg$less_than_or_equal$(\text{get}(i), \text{get}(j))$, that is, for which $i < j$ and $\text{get}(i) > \text{get}(j)$. It then swaps both values, in *only one* step, in the indexing function get. It results a sorted collection, based on the defined order less_than_or_equal, because when the program terminates (reaches a fixpoint), there is no i and j such that $i < j$ and $\neg$less_than_or_equal$(\text{get}(i), \text{get}(j))$ anymore.

In order to detail the formal semantics of this generic algorithm, it must be instantiated into a specific algorithm that works on a specified set ELEMENTS. The following simple integer-specific algorithm will be used:

State

ELEMENTS	$=$	$\{3, 1, 2\}$
INDICES	$=$	$\{0, 1, 2\}$
get	$=$	$\{(0, 3), (1, 1), (2, 2)\}$
less_than_or_equal	$=$	$\{(1, 1, \texttt{true}), (1, 2, \texttt{true}), (1, 3, \texttt{true}),$
		$(2, 1, \texttt{false}), (2, 2, \texttt{true}), (2, 3, \texttt{true}),$
		$(3, 1, \texttt{false}), (3, 2, \texttt{false}), (3, 3, \texttt{true})\}$
i	:	INDICES
j	:	INDICES

Program

$(\texttt{choose}\ (i, j)$	:	$i < j$
	$\wedge$	$\neg$less_than_or_equal$(\text{get}(i), \text{get}(j))$
	,	$\text{get}(i) := \text{get}(j) \parallel \text{get}(j) := \text{get}(i))$

It is important to remember that ASMs stop when they reach a fixpoint, if and only if a fixpoint is eventually reached. Therefore, any ASM is conceptually integrated into a "loop until fixpoint" statement. This is why the following detailed semantics of the example program includes such a conceptual loop.

1. Loop until fixpoint

   (a) Since the program is a `choose`, the non-deterministic choice rule is used to detail its semantics. This rule is stated as follows:

   Non-deterministic choice: If

   $$\Pi = (\texttt{choose } \bar{x} : \varphi(\bar{x}), \Pi_0),$$

   then let

   $$\text{Den}(\Pi, \mathcal{S}) := \begin{cases} \bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \text{Den}(\Pi_0, (\mathcal{S}, \bar{a})) & \text{if } \varphi^{\mathcal{S}} \neq \emptyset \\ \{\emptyset\} & \text{otherwise,} \end{cases}$$

   where $(\mathcal{S}, \bar{a})$ denotes the state obtained from $\mathcal{S}$ by adding the j-th element in $\bar{a}$ as an interpretation of the j-th variable in $\bar{x}$, for all j. Remember that $\bar{x}$ is a tuple of pairwise distinct variables and that variables are dynamic nullary functions in Υ. Note that every $\varphi(\bar{x})$, where $\bar{x} = (x_0, x_1, \ldots, x_k)$, defines a k-ary query on Fin(Υ), the class of finite structures over Υ. This query maps any $\mathcal{S} \in \text{Fin}(\Upsilon)$ to the k-ary relation $\varphi^{\mathcal{S}} := \{\bar{a} \in A^k : \mathcal{S} \models \varphi[\bar{a}]\}$.

   i. In order to detail the semantics of the example program, the binary query $\varphi^{\mathcal{S}}$ must first be calculated. In the example program, $\varphi = i < j \wedge \neg$less_than_or_equal(get$(i)$, get$(j)$). Therefore, $\varphi^{\mathcal{S}} = \{(0,1), (0,2)\}$.

   ii. Since $\varphi^{\mathcal{S}} \neq \emptyset$, $\bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \text{Den}(\Pi_0, (\mathcal{S}, \bar{a}))$, where

   $$\Pi_0 = (\text{get}(i) := \text{get}(j) \parallel \text{get}(j) := \text{get}(i)),$$

   must now be calculated. In fact, it consists in calculating

   $$\text{Den}(\Pi_0, (\mathcal{S}, (0,1))) \cup \text{Den}(\Pi_0, (\mathcal{S}, (0,2))).$$

   A. Let's start with $\text{Den}(\Pi_0, (\mathcal{S}, (0,1)))$. Since Π_0 is a parallel composition, the parallel composition rule is used to detail its semantics. This rule is stated as follows:

   Parallel composition: If

   $$\Pi = (\Pi_0 \parallel \Pi_1),$$

   then let

   $$\text{Den}(\Pi, \mathcal{S}) := \left\{ U_0 \cup U_1 : \begin{array}{l} U_0 \in \text{Den}(\Pi_0, \mathcal{S}), \\ U_1 \in \text{Den}(\Pi_1, \mathcal{S}) \end{array} \right\}.$$

Thus,

$$\begin{aligned}
&\text{Den}(\Pi_0, (\mathcal{S}, (0,1))) \\
&\quad = \text{Den}((\Pi_1 \parallel \Pi_2), (\mathcal{S}, (0,1))) \\
&\quad = \left\{ U_1 \cup U_2 : \begin{array}{l} U_1 \in \text{Den}(\Pi_1, (\mathcal{S}, (0,1))), \\ U_2 \in \text{Den}(\Pi_2, (\mathcal{S}, (0,1))) \end{array} \right\},
\end{aligned}$$

where $\Pi_1 = (\text{get}(i) := \text{get}(j))$ and $\Pi_2 = (\text{get}(j) := \text{get}(i))$.

- It follows that $\text{Den}(\Pi_1, (\mathcal{S}, (0,1)))$ and $\text{Den}(\Pi_2, (\mathcal{S}, (0,1)))$ must be calculated. Since Π_1 and Π_2 are atomic ASM programs, the update rule is used to detail their semantics. This rule is stated as follows:

  Update: If Π is an atomic ASM program, then let

  $$\text{Den}(\Pi, \mathcal{S}) := \big\{\{\Pi[\mathcal{S}]\}\big\},$$

  where $\Pi[\mathcal{S}]$ is identical to Π except that the arguments of functions and relations in Π are replaced by their values evaluated in $\mathcal{S}$.

  Therefore, $\text{Den}(\Pi_1, (\mathcal{S}, (0,1))) = \big\{\{\text{get}(0) := \text{get}(1)\}\big\}$ and $\text{Den}(\Pi_2, (\mathcal{S}, (0,1))) = \big\{\{\text{get}(1) := \text{get}(0)\}\big\}$.

B. Going back to the semantics of the parallel composition $(\Pi_1 \parallel \Pi_2)$, it follows that

$$\begin{aligned}
&\text{Den}(\Pi_0, (\mathcal{S}, (0,1))) \\
&\quad = \text{Den}((\Pi_1 \parallel \Pi_2), (\mathcal{S}, (0,1))) \\
&\quad = \left\{ U_1 \cup U_2 : \begin{array}{l} U_1 \in \text{Den}(\Pi_1, (\mathcal{S}, (0,1))), \\ U_2 \in \text{Den}(\Pi_2, (\mathcal{S}, (0,1))) \end{array} \right\} \\
&\quad = \big\{\{\text{get}(0) := \text{get}(1), \text{get}(1) := \text{get}(0)\}\big\}.
\end{aligned}$$

C. $\text{Den}(\Pi_0, (\mathcal{S}, (0,2)))$ must now be calculated. By following the same steps as for $\text{Den}(\Pi_0, (\mathcal{S}, (0,1)))$, it can be seen that

$$\begin{aligned}
&\text{Den}(\Pi_0, (\mathcal{S}, (0,2))) \\
&\quad = \text{Den}((\Pi_1 \parallel \Pi_2), (\mathcal{S}, (0,2))) \\
&\quad = \big\{\{\text{get}(0) := \text{get}(2), \text{get}(2) := \text{get}(0)\}\big\}.
\end{aligned}$$

iii. Therefore,

$$\begin{aligned}
&\text{Den}(\Pi, \mathcal{S}) \\
&\quad = \bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \text{Den}(\Pi_0, (\mathcal{S}, \bar{a})) \\
&\quad = \text{Den}(\Pi_0, (\mathcal{S}, (0,1))) \cup \text{Den}(\Pi_0, (\mathcal{S}, (0,2))) \\
&\quad = \left\{ \begin{array}{l} \{\text{get}(0) := \text{get}(1), \text{get}(1) := \text{get}(0)\}, \\ \{\text{get}(0) := \text{get}(2), \text{get}(2) := \text{get}(0)\} \end{array} \right\}.
\end{aligned}$$

(b) Since $\text{Den}(\Pi, \mathcal{S})$ is not a singleton set, the example program is non-deterministic. This is normal because it uses a `choose`.

Now that $\text{Den}(\Pi, \mathcal{S})$ has been calculated for the initial state, it is time to use the result of this calculation to fire a transition (see Definition A.6) and obtain a successor state different from the previous one. If no new successor state can be obtained by firing transitions, a fixpoint is reached and the ASM terminates. Otherwise, ASMs do not terminate.

In order to fire a transition, an update set is non-deterministically chosen among the elements of $\text{Den}(\Pi, \mathcal{S})$ and the rules of Definition A.6 are used to calculate a successor state. These rules ensure that the chosen update set is *consistent*.

Let's suppose that the first update set in $\text{Den}(\Pi, \mathcal{S})$ is chosen ($\{\text{get}(0) := \text{get}(1), \text{get}(1) := \text{get}(0)\}$). After updating the current state with this update set, the example ASM program looks like the following:

State

ELEMENTS	$=$	$\{3, 1, 2\}$
INDICES	$=$	$\{0, 1, 2\}$
get	$=$	$\{(0,1), (1,3), (2,2)\}$
less_than_or_equal	$=$	$\{(1,1,\texttt{true}), (1,2,\texttt{true}), (1,3,\texttt{true}),$
		$(2,1,\texttt{false}), (2,2,\texttt{true}), (2,3,\texttt{true}),$
		$(3,1,\texttt{false}), (3,2,\texttt{false}), (3,3,\texttt{true})\}$
i	:	INDICES
j	:	INDICES

Program

$(\texttt{choose}\ (i,j)$	:	$i < j$
	$\wedge$	$\neg$less_than_or_equal$(\text{get}(i), \text{get}(j))$
	,	$\text{get}(i) := \text{get}(j) \parallel \text{get}(j) := \text{get}(i))$

Note that both atomic ASM programs in the chosen update set are *simultaneously* executed and *not sequentially*, which leads to

$$\text{get} = \{(0,1),(1,3),(2,2)\}$$

and *not*

$$\text{get} = \{(0,1),(1,1),(2,2)\},$$

which, of course, would be erroneous.

The example program is then re-evaluated ($\text{Den}(\Pi, \mathcal{S})$ is re-calculated) on the new successor state (i.e., the new updated state). This is represented here by the "loop until fixpoint" statement.

Without giving the details,

$$\begin{aligned}
&\text{Den}(\Pi, \mathcal{S}) \\
&\quad = \bigcup_{\bar{a} \in \varphi^{\mathcal{S}}} \text{Den}(\Pi_0, (\mathcal{S}, \bar{a})) \\
&\quad = \text{Den}(\Pi_0, (\mathcal{S}, (1,2))) \\
&\quad = \big\{\{\text{get}(1) := \text{get}(2), \text{get}(2) := \text{get}(1)\}\big\}
\end{aligned}$$

on the successor state. After updating this state with the only update set available, the example ASM program looks like the following:

State

ELEMENTS	$=$	$\{3,1,2\}$
INDICES	$=$	$\{0,1,2\}$
get	$=$	$\{(0,1),(1,2),(2,3)\}$
less_than_or_equal	$=$	$\{(1,1,\texttt{true}),(1,2,\texttt{true}),(1,3,\texttt{true}),$
		$(2,1,\texttt{false}),(2,2,\texttt{true}),(2,3,\texttt{true}),$
		$(3,1,\texttt{false}),(3,2,\texttt{false}),(3,3,\texttt{true})\}$
i	:	INDICES
j	:	INDICES

Program

$(\texttt{choose}\ (i,j)$	:	$i < j$
	$\wedge$	$\neg\text{less\_than\_or\_equal}(\text{get}(i), \text{get}(j))$
	,	$\text{get}(i) := \text{get}(j) \parallel \text{get}(j) := \text{get}(i))$

When $\text{Den}(\Pi, \mathcal{S})$ is re-calculated on this new state, it results in a singleton set containing only the empty update set. A fixpoint has therefore been reached and the execution terminates.

The indexing function get now represents the collection $[1, 2, 3]$, which is, indeed, sorted.

B.2 Example #2 – Synchronized Clocks

This ASM program models three synchronized clocks. These clocks tick at the exact same rate and always show the exact same time. This example is used to illustrate the conditional and parameterized parallel composition rules detailed in Annex A. It could easily be extended to any number of clocks. For the sake of simplicity, it is assumed that the current time of each clock is represented by a natural number. This natural number can represent any unit of time, for instance, milliseconds.

State

CLOCKS	$=$	$\{C_1, C_2, C_3\}$
time	:	CLOCKS $\rightarrow \mathbb{N}$
ticking	:	Boolean
reset	:	Boolean

Program

((`if` *ticking* $\vee \neg$*reset*		
`then` (`do-for-all` i	:	$C_i \in$ CLOCKS
	,	time$(C_i) :=$ time$(C_i) + 1))$
	$\|\|$	
(`if` $\neg$*ticking* $\wedge$ *reset*		
`then` (`do-for-all` i	:	$C_i \in$ CLOCKS
	,	time$(C_i) := 0)))$

The set CLOCKS contains the three modeled clocks. The function time takes a clock in CLOCKS and returns its current time. The Boolean variables, i.e., nullary dynamic relations, *ticking* and *reset* determine if the clocks are ticking, that is, if

they are working (or enabled), and if they need to be reset, respectively. ASMs have the concept of external functions and relations, that is, functions and relations that act as oracles, in other words, for which their behavior is not specified; they simply return some value, according to their signature, anytime they are applied. They are called external because they are considered external to the given model. Therefore, they add flexibility to the modeling task because they let the designer interface his design with the model's environment without importing much complexity in it. His model can thus consider its environment which usually results in better design. In this example, *ticking* and *reset* could be considered external. For instance, they could model the end-user pressing on/off and reset buttons.

Once again, the program is rather simple. Intuitively, if the clocks are working, the program repeatedly increments their time. This increment is performed *simultaneously* for all clocks. This is why they are perfectly synchronized. If the clocks are not working and they need to be reset, the program simply resets their time to 0. Once again, they are all reset *simultaneously*. It is worth noticing that this program does not terminate. It will never reach a fixpoint.

Some initializations must be done before being able to detail the semantics of this example program. The initialized program looks like the following:

State

$$
\begin{array}{rcl}
\text{CLOCKS} & = & \{C_1, C_2, C_3\} \\
\text{time} & = & \{(C_1, 0), (C_2, 0), (C_3, 0)\} \\
\textit{ticking} & = & \texttt{true} \\
\textit{reset} & = & \texttt{false}
\end{array}
$$

Program

$$
\begin{array}{rcl}
((\texttt{if } \textit{ticking} \vee \neg \textit{reset} & & \\
\texttt{then } (\texttt{do-for-all } i & : & C_i \in \text{CLOCKS} \\
& , & \text{time}(C_i) := \text{time}(C_i) + 1)) \\
& \| & \\
(\texttt{if } \neg \textit{ticking} \wedge \textit{reset} & & \\
\texttt{then } (\texttt{do-for-all } i & : & C_i \in \text{CLOCKS} \\
& , & \text{time}(C_i) := 0)))
\end{array}
$$

Since it has already been studied in the previous example, the main parallel composition rule used by this program will not be detailed here, for the sake of brevity.

Moreover, because the two conditional rules are very similar, only the first one will be covered. In fact, the two conditional rules are executed in parallel but only one of the two conditions is `true` at any moment because they are exclusive. The detailed semantics of this first conditional rule follows:

1. Since it is an `if then`, the conditional rule is used to detail its semantics. This rule is stated as follows:

   Conditional: If

   $$\Pi = (\texttt{if } \varphi \texttt{ then } \Pi_0),$$

   then let

   $$\text{Den}(\Pi, \mathcal{S}) := \begin{cases} \text{Den}(\Pi_0, \mathcal{S}) & \text{if } \mathcal{S} \models \varphi \\ \{\emptyset\} & \text{otherwise.} \end{cases}$$

   (a) In order to detail the semantics of this rule, it must first be determined if $\mathcal{S} \models \varphi$, where $\varphi = \textit{ticking} \vee \neg \textit{reset}$. Because $\textit{ticking} = \texttt{true}$, $\varphi = \texttt{true}$ in $\mathcal{S}$, and thus, $\mathcal{S} \models \varphi$.

   (b) Since $\mathcal{S} \models \varphi$, $\text{Den}(\Pi_0, \mathcal{S})$, where

   $$\Pi_0 = (\texttt{do-for-all } i : C_i \in \text{CLOCKS}, \text{time}(C_i) := \text{time}(C_i) + 1),$$

   must now be calculated.

   i. Since Π_0 is a parameterized parallel composition, the parameterized parallel composition rule is used to detail its semantics. This rule is stated as follows:

   Parameterized parallel composition: If

   $$\Pi = (\texttt{do-for-all } \bar{x} : \varphi(\bar{x}), \Pi_0),$$

   then choose an index set I such that there exists a bijective mapping m from I to $\{\bar{a} : \mathcal{S} \models \varphi[\bar{a}]\}$. For each $i \in I$, set $\bar{a}_i = m(i)$ and let

   $$\text{Den}(\Pi, \mathcal{S}) := \left\{ (\bigcup_{i \in I} U_i) : (\forall i : U_i \in \text{Den}(\Pi_0, (\mathcal{S}, \bar{a}_i))) \right\},$$

   where $(\mathcal{S}, \bar{a}_i)$ denotes the state obtained from $\mathcal{S}$ by adding the j-th element in $\bar{a}_i$ as an interpretation of the j-th variable in $\bar{x}$, for all j. Remember that $\bar{x}$ is a tuple of pairwise distinct variables and that variables are dynamic nullary functions in Υ.

   A. The bijective mapping m must therefore first be calculated. In order to calculate m, $\{\bar{a} : \mathcal{S} \models \varphi[\bar{a}]\}$, where $\varphi = C_i \in \text{CLOCKS}$, is needed. Of course,

   $$\{\bar{a} : \mathcal{S} \models \varphi[\bar{a}]\} = \{(1), (2), (3)\}$$

is obtained. Therefore, $m = \{(1,(1)),(2,(2)),(3,(3))\}$, if $I = \{1,2,3\}$ is used, and finally, $\bar{a}_1 = (1), \bar{a}_2 = (2)$, and $\bar{a}_3 = (3)$.

B. Thus,

$$\begin{aligned}
&\text{Den}(\Pi_0, \mathcal{S}) \\
&= \text{Den}((\texttt{do-for-all } i : C_i \in \text{CLOCKS}, \\
&\quad \text{time}(C_i) := \text{time}(C_i) + 1), \mathcal{S}) \\
&= \left\{ (\bigcup_{i \in I} U_i) : (\forall i : U_i \in \text{Den}(\Pi_1, (\mathcal{S}, \bar{a}_i))) \right\},
\end{aligned}$$

where $\Pi_1 = (\text{time}(C_i) := \text{time}(C_i) + 1)$. In fact, it boils down to calculating

$$\text{Den}(\Pi_1, (\mathcal{S}, (1))) \cup \text{Den}(\Pi_1, (\mathcal{S}, (2))) \cup \text{Den}(\Pi_1, (\mathcal{S}, (3))).$$

- Let's start with $\text{Den}(\Pi_1, (\mathcal{S}, (1)))$, where

  $$\Pi_1 = (\text{time}(C_i) := \text{time}(C_i) + 1),$$

  which is an atomic ASM program. The update rule is thus used to detail its semantics. This rule is stated as follows:

  Update: If Π is an atomic ASM program, then let

  $$\text{Den}(\Pi, \mathcal{S}) := \big\{\{\Pi[\mathcal{S}]\}\big\},$$

  where $\Pi[\mathcal{S}]$ is identical to Π except that the arguments of functions and relations in Π are replaced by their values evaluated in $\mathcal{S}$.

  Therefore, $\text{Den}(\Pi_1, (\mathcal{S}, (1))) = \big\{\{\text{time}(C_1) := \text{time}(C_1)+1\}\big\}$.
- Similarly, $\text{Den}(\Pi_1, (\mathcal{S}, (2))) = \big\{\{\text{time}(C_2) := \text{time}(C_2) + 1\}\big\}$ and $\text{Den}(\Pi_1, (\mathcal{S}, (3))) = \big\{\{\text{time}(C_3) := \text{time}(C_3) + 1\}\big\}$.

C. Going back to the semantics of the parameterized parallel composition,

$$\begin{aligned}
&\text{Den}(\Pi_0, \mathcal{S}) \\
&= \text{Den}((\texttt{do-for-all } i : C_i \in \text{CLOCKS}, \\
&\quad \text{time}(C_i) := \text{time}(C_i) + 1), \mathcal{S}) \\
&= \left\{ (\bigcup_{i \in I} U_i) : (\forall i : U_i \in \text{Den}(\Pi_1, (\mathcal{S}, \bar{a}_i))) \right\} \\
&= \big\{\{\text{time}(C_1) := \text{time}(C_1) + 1, \\
&\quad \text{time}(C_2) := \text{time}(C_2) + 1, \\
&\quad \text{time}(C_3) := \text{time}(C_3) + 1\}\big\}.
\end{aligned}$$

(c) Finally,

$$\begin{aligned}
&\text{Den}(\Pi, \mathcal{S}) \\
&\quad = \text{Den}(\Pi_0, \mathcal{S}) \\
&\quad = \big\{\{\text{time}(C_1) := \text{time}(C_1) + 1, \\
&\qquad \text{time}(C_2) := \text{time}(C_2) + 1, \\
&\qquad \text{time}(C_3) := \text{time}(C_3) + 1\}\big\}.
\end{aligned}$$

2. Since $\text{Den}(\Pi, \mathcal{S})$ is a singleton set, the example program (or more precisely, the first `if then`) is deterministic. This is normal because it does not use a `choose` (in fact, the same is true for the entire example program).

   Now that $\text{Den}(\Pi, \mathcal{S})$ has been calculated for the initial state, it is time to use the result of this calculation to fire a transition (see Definition A.6) and obtain a successor state different from the previous one. If no new successor state can be obtained by firing transitions, a fixpoint is reached and the ASM terminates. Otherwise, ASMs do not terminate.

   In order to fire a transition, an update set is non-deterministically chosen among the elements of $\text{Den}(\Pi, \mathcal{S})$ and the rules of Definition A.6 are used to calculate a successor state. These rules ensure that the chosen update set is *consistent*. In this case, $\text{Den}(\Pi, \mathcal{S})$ is a singleton set and therefore, its unique update set is automatically chosen.

   After updating the current state with this update set, the example ASM program looks like the following:

State

CLOCKS	$=$	$\{C_1, C_2, C_3\}$
time	$=$	$\{(C_1, 1), (C_2, 1), (C_3, 1)\}$
ticking	$=$	`true`
reset	$=$	`false`

Program

((`if` *ticking* $\vee \neg$*reset*		
`then (do-for-all` i	:	$C_i \in$ CLOCKS
	,	$\text{time}(C_i) := \text{time}(C_i) + 1))$
	$\|\|$	
(`if` $\neg$*ticking* $\wedge$ *reset*		
`then (do-for-all` i	:	$C_i \in$ CLOCKS
	,	$\text{time}(C_i) := 0)))$

The example program is then re-evaluated (Den$(\Pi, \mathcal{S})$ is re-calculated) on the new successor state (i.e., the new updated state).

Without giving the details, Den$(\Pi, \mathcal{S})$ is the same set on the successor state. Therefore, after updating this state with the only update set available, the example ASM program looks like the following:

State

CLOCKS	$=$	$\{C_1, C_2, C_3\}$
time	$=$	$\{(C_1, 2), (C_2, 2), (C_3, 2)\}$
ticking	$=$	`true`
reset	$=$	`false`

Program

((`if` *ticking* $\vee \neg$*reset*		
`then (do-for-all` i	:	$C_i \in$ CLOCKS
	,	time$(C_i) :=$ time$(C_i) + 1))$
	$\|\|$	
(`if` $\neg$*ticking* $\wedge$ *reset*		
`then (do-for-all` i	:	$C_i \in$ CLOCKS
	,	time$(C_i) := 0)))$

When Den$(\Pi, \mathcal{S})$ is re-calculated on this new state, it results the same singleton set again. Without giving a formal proof (which is very simple anyway; it can easily be presented as a simple truth table for *ticking* and *reset*), it can be seen that a fixpoint is never reached when executing this program. This ASM never terminates. Thus, the three clocks are incremented indefinitely unless *ticking* $=$ `false` and *reset* $=$ `true`, when they are reset to 0.

B.3 Example #3 – Rock, Paper & Scissors!

This ASM program models the game of "Rock, Paper & Scissors!". Its semantics will not be detailed. It is only used to demonstrate the modeling power of ASMs. The following terminology is taken from http://www.worldrps.com/.

State

THROWS	$=$	$\{\text{Rock}, \text{Paper}, \text{Scissors}\}$
PLAYERS	$=$	$\{P_1, P_2\}$
WINNER	$=$	PLAYERS $\cup \{\text{Stalemate}\}$
approach	:	PLAYERS $\rightarrow$ THROWS
win	:	THROWS $\times$ THROWS $\rightarrow$ WINNER
current_winner	:	WINNER

Program

current_winner	$:=$	$\text{win}(\text{approach}(P_1), \text{approach}(P_2))$

The function named approach should be external. Therefore, it does not have to be defined. However, the function named win needs more specification. It is given in the following ASM program:

State

THROWS	$=$	$\{\text{Rock}, \text{Paper}, \text{Scissors}\}$
PLAYERS	$=$	$\{P_1, P_2\}$
WINNER	$=$	PLAYERS $\cup \{\text{Stalemate}\}$
approach	:	PLAYERS $\rightarrow$ THROWS
win	$=$	$\{(\text{Rock}, \text{Rock}, \text{Stalemate}), (\text{Rock}, \text{Paper}, P_2),$
		$(\text{Rock}, \text{Scissors}, P_1), (\text{Paper}, \text{Rock}, P_1),$
		$(\text{Paper}, \text{Paper}, \text{Stalemate}), (\text{Paper}, \text{Scissors}, P_2),$
		$(\text{Scissors}, \text{Rock}, P_2), (\text{Scissors}, \text{Paper}, P_1),$
		$(\text{Scissors}, \text{Scissors}, \text{Stalemate})\}$
current_winner	:	WINNER

Program

current_winner	$:=$	$\text{win}(\text{approach}(P_1), \text{approach}(P_2))$

Simple! Don't you think?

B.4 Example #4 – Take a Guess

A little quiz for the theoretical computer science enthusiast: what does this ASM program model?

State

$$
\begin{array}{rcl}
\mathrm{S} & = & \{s_1, s_2, \ldots, s_n\}, \text{where } n \in \mathbb{N} \\
\Sigma & = & \{a_1, a_2, \ldots, a_i\}, \text{where } i \in \mathbb{N} \\
\Gamma & = & \{t_1, t_2, \ldots, t_j\} \cup \Sigma \cup \{\triangle\}, \text{where } j \in \mathbb{N} \\
\delta & : & (\mathrm{S} - \{h\}) \times \Gamma \rightarrow \mathrm{S} \times (\Gamma \cup \{\mathrm{L}, \mathrm{R}\}) \\
\iota & : & \mathrm{S} \\
h & : & \mathrm{S}
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{STEPS} & = & \{\mathit{read}, \mathit{transit}, \mathit{do\_action}, \mathit{halt}\} \\
\mathit{step} & : & \mathrm{STEPS}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{state} & : & \mathrm{S} \times (\Gamma \cup \{\mathrm{L}, \mathrm{R}\}) \\
\mathrm{first} & : & \mathrm{S} \times (\Gamma \cup \{\mathrm{L}, \mathrm{R}\}) \rightarrow \mathrm{S} \\
\mathrm{second} & : & \mathrm{S} \times (\Gamma \cup \{\mathrm{L}, \mathrm{R}\}) \rightarrow (\Gamma \cup \{\mathrm{L}, \mathrm{R}\}) \\
\mathit{symbol} & : & \Gamma
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{head} & : & \mathbb{N} \\
\mathrm{tape} & : & \mathbb{N} \rightarrow \Gamma
\end{array}
$$

Program

$$
\begin{array}{rcl}
((\texttt{if } \mathit{step} = \mathit{read} \wedge & & \\
\mathrm{first}(\mathit{state}) \neq h & & \\
\texttt{then } (\mathit{symbol} := \mathit{tape}(\mathit{head}) & \| & \mathit{step} := \mathit{transit})) \\
\| & & \\
(\texttt{if } \mathit{step} = \mathit{transit} \wedge & & \\
\mathrm{first}(\mathit{state}) \neq h & & \\
\texttt{then } (\mathit{state} := \delta(\mathrm{first}(\mathit{state}), \mathit{symbol}) & \| & \mathit{step} := \mathit{do\_action})) \\
\| & & \\
(\texttt{if } \mathit{step} = \mathit{do\_action} \wedge & & \\
\mathrm{second}(\mathit{state}) = \mathrm{L} \wedge & &
\end{array}
$$

$$
\begin{array}{rcl}
\text{first}(\textit{state}) \neq h & & \\
\texttt{then } (\textit{head} := \textit{head} - 1 & \| & \textit{step} := \textit{read})) \\
& \| & \\
(\texttt{if } \textit{step} = \textit{do\_action} \wedge & & \\
\text{second}(\textit{state}) = \text{R} \wedge & & \\
\text{first}(\textit{state}) \neq h & & \\
\texttt{then } (\textit{head} := \textit{head} + 1 & \| & \textit{step} := \textit{read})) \\
& \| & \\
(\texttt{if } \textit{step} = \textit{do\_action} \wedge & & \\
\text{second}(\textit{state}) \neq \text{L} \wedge & & \\
\text{second}(\textit{state}) \neq \text{R} \wedge & & \\
\text{first}(\textit{state}) \neq h & & \\
\texttt{then } (\text{tape}(\textit{head}) := \text{second}(\textit{state}) & \| & \textit{step} := \textit{read})) \\
& \| & \\
(\texttt{if } \textit{step} \neq \textit{halt} \wedge & & \\
\text{first}(\textit{state}) = h & & \\
\texttt{then } \textit{step} := \textit{halt})) & &
\end{array}
$$

See [5] for the answer.

Isn't it interesting?

[5] It models a Turing machine.

Annex C
Model-Checking Abstract State Machines

Now that Abstract State Machines have been portrayed, it is appropriate to ask the question: "Can we automatically verify Abstract State Machines?", i.e., given an ASM M and a property φ expressed in some appropriate logic, is it possible to decide whether $M \models \varphi$ for all inputs? Since ASMs are computationally complete [28] [25] [22], this decision problem is, in its full generality, undecidable. Therefore, there is interest in determining if this problem is decidable for a restrained subclass of ASMs.

Fortunately, this is not the first time that this question has been raised. At least two researchers have worked extensively on this subject: Kirsten Winter and Marc Spielmann. It is reasonable to say that Winter concentrated her efforts mostly on practical concerns and Spielmann, mostly on the theoretical aspects.

C.1 Kirsten Winter's Work on Model-Checking ASMs

In her PhD thesis [29], Kirsten Winter showed that it is possible to translate ASMs to more traditional Kripke structures, which are the standard transition systems used in model-checking theory. In fact, ASMs were translated to model specifications appropriate for two existing tools: the Symbolic Model Verifier (SMV) model checker and the Multiway Decision Graph (MDG) package.

The resultant models needed to be kept small in order to be model-checkable in a reasonable time or simply to fit in memory. One way to achieve this objective was to support *abstraction*. The good news is that ASMs were designed with abstraction in mind in order to model any algorithm at its natural abstraction level.

Moreover, state spaces had to stay fixed. The state space of an ASM model is said to stay *fixed* if the universe of its structure is not extended during a run. Universe extension is introduced by the `import` rule of Gurevich's original ASM theory [23]. However, the theory introduced by Spielmann, and presented in Annex A, avoids this `import` rule. Rather, it adopts a more recent and very elegant method for object invention, the reason why universes have to be extended, from [30] and [31]. This method uses hereditarily finite sets built from the elements of the input domain $\Upsilon_{in} \cup \Upsilon_{stat}$. Since Winter did not use Spielmann's theory, she had to ignore `import` rules in order to model-check ASMs.

By building an ASM front-end to SMV and MDG and with these restrictions, Winter concluded that the model-checking approach for ASMs works not only in principle, but in practice too, as two case studies show (refer to [29] for further details).

C.2 Marc Spielmann's Work on Model-Checking ASMs

In his PhD thesis [22], Marc Spielmann investigated theoretical concerns about ASMs. His theory has already been presented in Annex A. The theoretical foundations on which his theorems are based ([22]) are outlined in this subsection.

C.2.1 First-Order Branching Temporal Logic

Spielmann's specification language for ASMs is First-Order Branching Temporal Logic (FBTL). Before presenting the FBTL syntax and semantics, the following definition must be given.

Definition C.1 (Computation Graph) Let $M = (\Upsilon, initial, \Pi)$ be an ASM and let $\mathcal{I}$ be an input appropriate for M. The *computation graph* of M on $\mathcal{I}$, denoted $C_M(\mathcal{I})$, is a triple $(States, Trans, \mathcal{S}_0)$, where:

- $States$ is the set of those states over Υ whose universe is identical with the universe of $initial(\mathcal{I})$,
- $Trans \subseteq States \times States$ is the restriction of $Trans_\Pi$ to $States$, and
- $\mathcal{S}_0 = initial(\mathcal{I})$.

The infinite paths in $C_M(\mathcal{I})$ starting at $\mathcal{S}_0$ are precisely the runs of M on $\mathcal{I}$. Thus, $C_M(\mathcal{I})$ defines the structure on which FBTL's formulas are to be interpreted.

The FBTL syntax can now be defined.

Definition C.2 (FBTL – Syntax) FBTL being a straightforward extension of the well-known propositional branching-time logic Computation Tree Logic* (CTL*) [32] by first-order reasoning, its syntax is composed of state and path formulas of first-order branching temporal logic, inductively defined as follows:

- **(State1)** Every atomic first-order formula is an atomic state formula.
- **(State2)** If φ is a path formula, then $\mathbf{E}\varphi$ is a state formula.
- **(Path1)** Every state formula is a path formula.
- **(Path2)** If φ and ψ are path formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are path formulas.

•**(StatePath1)** If φ and ψ are state (respectively path) formulas, then $\neg\varphi$ and $\varphi \vee \psi$ are state (respectively path) formulas.

•**(StatePath2)** If x is a variable and φ is a state (respectively path) formula, then $\exists x\varphi$ is a state (respectively path) formula.

FBTL denotes the set of state formulas.

Definition C.3 (FBTL – Semantics) Let $C_M(\mathcal{I}) = (States, Trans, \mathcal{S}_0)$ be a computation graph of an ASM of vocabulary Υ, and let $\rho = (\mathcal{S}'_i)_{i\in\omega}$ be an infinite path in $C_M(\mathcal{I})$, not necessarily starting at the initial state $\mathcal{S}_0$. For any $j \in \omega$, ρ^j denotes the infinite path $(\mathcal{S}'_{i+j})_{i\in\omega}$, i.e., the suffix $\mathcal{S}'_j, \mathcal{S}'_{j+1}, \ldots$ of ρ. Let φ (respectively ψ) be a state (respectively path) formula of FBTL over Υ with $\bar{x}$ being the tuple of free variables in φ (respectively ψ). Simultaneously for every state $\mathcal{S}$ in $C_M(\mathcal{I})$, every infinite path ρ in $C_M(\mathcal{I})$, and all interpretations $\bar{a}$ of the variables $\bar{x}$ (chosen from the universe of $\mathcal{S}_0$), the two traditional satisfaction relations $(C_M(\mathcal{I}), \mathcal{S}, \bar{a}) \models \varphi$ and $(C_M(\mathcal{I}), \rho, \bar{a}) \models \psi$ are defined by induction on the construction of φ and ψ, as follows:

•**(State1)** $(C_M(\mathcal{I}), \mathcal{S}, \bar{a}) \models \varphi$ iff $\mathcal{S} \models \varphi[\bar{a}]$ and φ is an atomic state formula.

•**(State2)** $(C_M(\mathcal{I}), \mathcal{S}, \bar{a}) \models \mathbf{E}\varphi$ iff there is an infinite path ρ' in $C_M(\mathcal{I})$ starting at $\mathcal{S}$ such that $(C_M(\mathcal{I}), \rho', \bar{a}) \models \varphi$.

•**(Path1)** $(C_M(\mathcal{I}), \rho, \bar{a}) \models \varphi$ iff $(C_M(\mathcal{I}), \mathcal{S}'_0, \bar{a}) \models \varphi$, where $\mathcal{S}'_0$ is the first state in ρ.

•**(Path2)** $(C_M(\mathcal{I}), \rho, \bar{a}) \models \mathbf{X}\varphi$ iff $(C_M(\mathcal{I}), \rho^1, \bar{a}) \models \varphi$.

$(C_M(\mathcal{I}), \rho, \bar{a}) \models \varphi\mathbf{U}\psi$ iff there is an $i \in \omega$ such that $(C_M(\mathcal{I}), \rho^i, \bar{a}) \models \psi$ and for all $0 \leq j < i$, $(C_M(\mathcal{I}), \rho^j, \bar{a}) \models \varphi$.

$(C_M(\mathcal{I}), \rho, \bar{a}) \models \varphi\mathbf{B}\psi$ iff for every $i \in \omega$, if $(C_M(\mathcal{I}), \rho^i, \bar{a}) \models \psi$, then there is a $0 \leq j < i$ with $(C_M(\mathcal{I}), \rho^j, \bar{a}) \models \varphi$.

•**(StatePath1)** Let σ stand for either a state $\mathcal{S}$ or an infinite path ρ in $C_M(\mathcal{I})$, depending on whether φ and ψ are state or path formulas.

$(C_M(\mathcal{I}), \sigma, \bar{a}) \models \neg\varphi$ iff not$((C_M(\mathcal{I}), \sigma, \bar{a}) \models \varphi)$.

$(C_M(\mathcal{I}), \sigma, \bar{a}) \models \varphi \vee \psi$ iff $(C_M(\mathcal{I}), \sigma, \bar{a}) \models \varphi$ or $(C_M(\mathcal{I}), \sigma, \bar{a}) \models \psi$.

•**(StatePath2)** Let σ stand for either a state $\mathcal{S}$ or an infinite path ρ in $C_M(\mathcal{I})$, depending on whether φ is a state or a path formula.

$(C_M(\mathcal{I}), \sigma, \bar{a}) \models \exists y \varphi(\bar{x}, y)$ iff there is an element b in the universe of $\mathcal{S}_0$ such that $(C_M(\mathcal{I}), \sigma, (\bar{a}, b)) \models \varphi(\bar{x}, y)$.

The following abbreviations from CTL* are also defined in FBTL:

$$\mathbf{A}\varphi := \neg\mathbf{E}\neg\varphi,$$

$$\mathbf{F}\varphi := \mathit{true}\mathbf{U}\varphi, \text{ and}$$

$$\mathbf{G}\varphi := \mathit{false}\mathbf{B}\varphi.$$

Finally, note that $\varphi\mathbf{B}\psi \equiv \neg(\neg\varphi\mathbf{U}\psi)$.

C.2.2 Sequential Nullary ASMs

Sequential nullary ASMs are now briefly introduced, after defining finitely initialized ASMs.

Definition C.4 (Reduction) For every $\Upsilon' \subseteq \Upsilon$, $\mathcal{A}|\Upsilon'$ denotes the *reduction* of $\mathcal{A}$ to Υ', i.e., the structure over Υ' obtained from $\mathcal{A}$ by removing the interpretations of the symbols in $\Upsilon - \Upsilon'$.

Definition C.5 (Finitely Initialized ASM) An ASM $M = (\Upsilon, initial, \Pi)$ is *finitely initialized* if for every input $\mathcal{I}$ appropriate for M, $initial(\mathcal{I})|\Upsilon_{in} = \mathcal{I}$ and in $initial(\mathcal{I})|(\Upsilon_{stat} \cup \Upsilon_{dyn} \cup \Upsilon_{out})$, all relations are empty and all functions have range $\{0\}$.

Definition C.6 (Sequential Nullary ASM) With these definitions in mind, *sequential nullary ASMs* are basically finitely-initialized ASMs whose Υ and Π have the following constraints:

- all dynamic symbols are nullary,
- all guards are quantifier-free, and
- `do-for-all` does not occur.

These ASMs are *sequential* because of the last two constraints and *nullary* because of the first one.

The syntax and semantics of *sequential nullary ASM programs* are easily defined using the classic ASM syntax and semantics given in Annex A. They are therefore not formally defined in this document. Please consult [22] for more information.

Spielmann's theorems are based on sequential nullary ASMs and a decidable fragment of FBTL (not presented in this document; refer to [22]). He shows that sequential nullary ASMs and the decidable fragment of FBTL are expressive and proves that the verification problem is decidable for these two mathematical entities, i.e., that if M is a sequential nullary ASM and φ is a property expressed in the decidable fragment of FBTL, then whether $M \models \varphi$ for every input $\mathcal{I}$ appropriate for M is decidable.

References

1. Young, Michal and Taylor, Richard N. (1991). Rethinking the Taxonomy of Fault Detection Techniques. Technical Report. Purdue University. 1

2. Federal Networking and Information Technology Research and Development (NITRD) (2001). Workshop on New Visions for Software Design and Productivity, Federal Networking and Information Technology Research and Development (NITRD). Software Design and Productivity Coordinating Group, Interagency Working Group on Information Technology, Research and Development. http://www.hpcc.gov/pubs/sdp_wrkshp_final.pdf. 1

3. Katoen, Joost-Pieter (1998-1999. Local file: papers/avvs.pdf). Concepts, Algorithms, and Tools for Model Checking. Lecture Notes of the Course "System Validation (using Model Checking)". 2, 3

4. RTI (2002). The Economic Impacts of Inadequate Infrastructure for Software Testing. Planning Report 02-03. National Institute of Standards and Technology, US Department of Commerce. 2

5. Schefer, Rob (2002). How safe is your business? Perception/Reality Discontent. Enterprise Data Center Strategies (EDCS) 1060. META Group. 2

6. Inc., Géo Alliance International (2003). Certifying Critical Software: JACC Market Study. Summary and Final Report. 2

7. Schneier, Bruce (2000). Secrets and Lies: Digital Security in a Networked World, John Wiley & Sons. 3

8. Mann, Charles C. (2002). Special Report: Homeland Insecurity. *The Atlantic Monthly*, pp. 81–102. 4

9. Bergeron, Jean, Debbabi, Mourad, Desharnais, Jules, Erhioui, Mourad M., Lavoie, Yvan, and Tawbi, Nadia (2001). Static Detection of Malicious Code in Executable Programs. In *Symposium on Requirements Engineering for Information Security (SREIS)*, Purdue University, Indianapolis, ID, USA. Local file: papers/SREIS2001.pdf. 5

10. Debbabi, Mourad, Girard, Marc, Poulin, Luc, and Salois, Martin (2001). Monitoring of Malicious Activity in Software Systems. In *Symposium on Requirements Engineering for Information Security (SREIS)*, Purdue University, Indianapolis, ID, USA. Local file: papers/MonitoringOfMaliciousActivityInSoftwareSystems.pdf. 6

11. Debbabi, Mourad, Desharnais, Jules, Fourati, Myriam, Menif, Emna, Painchaud, Frédéric, and Tawbi, Nadia (2002). Secure Self-Certified Code for Java. In *Formal Aspects of Security (FASec)*, London, UK. Local file: papers/paperFinal.pdf. 7, 10

12. Lindholm, Tim and Yellin, Frank (1999). The Java Virtual Machine Specification, Second ed. The Java Series. Addison Wesley. 9

13. Painchaud, Frédéric (2003). The State of the Art in Java Model Checking. Technical Report. Defence Research & Development Canada – Valcartier. Local file: papers/SOTAJavaMC.pdf. 13

14. Huggins, Jim. Abstract State Machines: A Formal Method for Specification and Verification. Internet website. `http://www.eecs.umich.edu/gasm/`. 23, 40

15. Börger, Egon. Abstract State Machine Tutorial. Internet website. `http://www.di.unipi.it/~boerger/ASMTutorialEtaps.html`. 24

16. Daigle, Sylvain and Painchaud, Frédéric (2003). The State of the Art in Abstract State Machines (ASMs). Technical Report. Defence Research & Development Canada – Valcartier. Local file: papers/SOTAASM.pdf. 24, 40

17. Stärk, Robert, Schmid, Joachim, and Börger, Egon (2001). Java and the Java Virtual Machine: Definition, Verification, Validation, Springer. 40, 47

18. Gurevich, Yuri (To appear). Abstract State Machines: An Overview of the Project. *Lecture Notes in Computer Science*. Local file: papers/165.pdf. 41

19. Charpentier, Robert and Salois, Martin (2003). Secure Software from Design to Binary. In *High Confidence Software and Systems Conference*, NSA. Baltimore, Maryland, United States. Local file: papers/SecureSoftwareFromDesignToBinary.pdf. 41, 42

20. Jürjens, Jan (2002). Principles for Secure Systems Design. Ph.D. thesis. Wolfson College. Local file: papers/JurjensThesis.pdf. 42

21. of Software Engineering, Foundations (2002). AsmL: The Abstract State Machine Language. Microsoft Research. Local file: papers/AsmL2Reference.pdf. 47

22. Spielmann, Marc (2000). Abstract State Machines: Verification Problems and Complexity. Ph.D. thesis. Rheinisch-Westfälische Technische Hochschule Aachen. Local file: papers/diss00.pdf. 50, 71, 72, 75

23. Gurevich, Yuri (1995). Evolving Algebras 1993: Lipari Guide. In Börger, Egon, (Ed.), *Specification and Validation Methods*, pp. 231–243. Oxford University Press. Local file: papers/guide.pdf. 50, 71

24. Gurevich, Yuri (1997). May 1997 Draft of the ASM Guide. Local file: papers/guide97.pdf. 50

25. Gurevich, Yuri (1999). Sequential Abstract State Machines Capture Sequential Algorithms. (Technical Report 99-65). Microsoft Research. Redmond, Washington. Local file: papers/p77-gurevich.pdf. 50, 71

26. (2003). The ASM Workbench – ASM-SL. Internet website. `http://www.uni-paderborn.de/fachbereich/AG/rammig/UK/gruppe/giusp/workbench/asm-sl.html`. 56

27. Castillo, Giuseppe Del (2000). The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. Ph.D. thesis. Informatik und Heinz Nixdorf Institut. Universität Paderborn. Local file: papers/asm-sl.pdf. 56

28. Blass, Andreas and Gurevich, Yuri (2002). Abstract State Machines Capture Parallel Algorithms. (Technical Report 2001-117). Microsoft Research. Redmond, Washington. Local file: papers/157.pdf. 71

29. Winter, Kirsten (2001). Model Checking Abstract State Machines. Ph.D. thesis. School of Electrical Engineering and Computer Sciences. Technical University of Berlin. Local file: papers/winterkirsten.pdf. 71

30. Blass, A., Gurevich, Yuri, and Shelah, S. (1999). Choiceless Polynomial Time. (Technical Report 99-08). Microsoft Research. Redmond, Washington. Local file: papers/choiceless.pdf. 71

31. Blass, Andreas and Gurevich, Yuri (2000). Background, Reserve, and Gandy Machines. In Clote, Peter and Schwichtenberg, Helmut, (Eds.), *Lecture Notes in Computer Science*, Vol. 1862, pp. 1–17. Springer. Local file: papers/143.pdf. 71

32. Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science*, Vol. B of *Formal Models and Semantics*, pp. 995–1072. Elsevier Science Publishers B.V. 72

List of acronyms

API	Application Programming Interface
ASM	Abstract State Machine
C2IS	Command and Control Information Systems
COTS	Commercial-Off-the-Shelf
DRDC	Defence Research and Development Canada
FBTL	First-Order Branching Temporal Logic
JACC	Java Certifying Compilation System
JBV	Java Bytecode Verifier
JML	Java Modeling Language
JPF	Java PathFinder
JVM	Java Virtual Machine
MaliCOTS	Malicious COTS
UAV	Unmanned Aerial Vehicle
UML	Unified Modelling Language

Glossary

Certifying compilation	A compilation process that produces annotations along with compiled object code. The annotations provide information on the compiled code such as types, for example.
Dynamic analysis	A set of techniques to analyze software during execution.
Formal methods	A set of hardware and software analysis methods that are based on mathematical formalism, symbolism, and logic.
Java Bytecode Verifier	A component of the Java Virtual Machine that is responsible to ensure low level security (safety).
Java Platform Architecture	The architecture of the Java Platform.
Java Security Architecture	The security architecture of the Java Platform.
Java Security Manager	A component of the Java Virtual Machine that is responsible to ensure high level security.
Java Virtual Machine	The main component of the Java Platform that is responsible to safely and securely execute Java programs.
Model checking	A technique used to verify that a property holds on every possible state of a hardware or software system.
Static analysis	A set of techniques to analyze software prior to execution.
Validation	A process that is used to assure that the design requirements are met in a product being developed.
Verification	A process that is used to assure that a product being developed meets some predefined quality standards.

Distribution list

Internal Distribution
DRDC Valcartier TM 2004 - 001

1 - Director General
3 - Document Library
1 - Head/Information and Knowledge Management
1 - Head/Systems of Systems
1 - F. Painchaud (author)
1 - D. Demers
1 - R. Charpentier
1 - M. Salois
1 - F. Michaud
1 - M. Lizotte
1 - M. Gauvin

External Distribution
DRDC Valcartier TM 2004 - 001

1 - Directorate R & D – Knowledge and Information Management
1 - Defence R & D Canada – Headquarters
2 - Director General Joint Force Development
1 - Director Sciences and Technology – Command and Control Information Systems
2 - Directorate Sciences and Technology – Command and Control Information Systems
2 - Directorate Distributed Computing Engineering and Integration
1 - Directorate Land Command Systems Program Management
2 - Directorate Land Requirements
1 - Assistant Deputy Minister (Information Management)
1 - Canadian Forces Experimentation Center
1 - Canadian Forces Command and Staff College Toronto

Attn: Commanding Officer

1 - Canadian Forces Maritime Warfare School
Canadian Forces Base Halifax
Halifax, Nova Scotia
Attn: Commanding Officer

1 - Communications Security Establishment
Attn: Louis Bélanger

1 - Communications Security Establishment
Attn: Sylvain Bazinet

1 - Communications Security Establishment
Attn: Simon Arcand

3 - Defence R & D Canada – Ottawa
Attn: Mark McIntyre

1 - Royal Military College of Canada
Kingston
Attn: Dr Scott Knight

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1. ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.)

Defence Research and Development Canada – Valcartier
2459 Pie-XI Blvd North, Val-Bélair, Québec, Canada, G3J 1X5

2. SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable).

UNCLASSIFIED

3. TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title).

Combining Static and Dynamic Analysis for Advanced Certification of Java™ C2IS

4. AUTHORS
(Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)

Painchaud, Frédéric

5. DATE OF PUBLICATION (month and year of publication of document)	6a. NO. OF PAGES (total containing information. Include Annexes, Appendices, etc).	6b. NO. OF REFS (total cited in document)
March 2004	92	32

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered).

Technical Memorandum

8. SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address).

Defence Research and Development Canada – Valcartier
2459 Pie-XI Blvd North, Val-Bélair, Québec, Canada, G3J 1X5

9a. PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Specify whether project or grant).	9b. CONTRACT NO. (if appropriate, the applicable number under which the document was written).
15BF34	

10a. ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.)	10b. OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC Valcartier TM 2004 - 001	

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

(X) Unlimited distribution
( ) Defence departments and defence contractors; further distribution only as approved
( ) Defence departments and Canadian defence contractors; further distribution only as approved
( ) Government departments and agencies; further distribution only as approved
( ) Defence departments; further distribution only as approved
( ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

Since information-processing systems are becoming omnipresent in our daily lives and in safety-critical infrastructures, there is a strong move to validate and verify them, in order to make them more fault-tolerant and secure. To meet this challenging objective, it is believed that formal methods must be included within the hardware and software engineering processes, which presently suffer from having insufficient (although growing) tool-support with a sound mathematical foundation. Static and dynamic analysis, certifying compilation, and model checking are good examples of such formal methods for hardware and software validation and verification.

This document first presents a summary of two successfully completed projects on malicious code detection and Java certifying compilation, which used formal methods to certify software packages for security purposes. This is followed by an introduction to model checking and Abstract State Machines as a basis for this project. This new project aims at defining and implementing a novel hybrid model-checking approach for Java. This approach is *hybrid* in the sense that it combines static and dynamic analysis principles in order to provide more precise model checking. Ultimately, this project is therefore about using Abstract State Machines (more precisely, the Abstract State Machine Language) as a modeling formalism for Java programs, developing a model checker for this formalism, and parameterizing this model checker in the hope of being able to decide more security properties.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

Software Certification
Java
Java Security
Static Analysis
Dynamic Analysis
Abstract State Machines
Model Checking

Defence R&D Canada

Canada's leader in defence
and national security R&D

R & D pour la défense Canada

Chef de file au Canada en R & D
pour la défense et la sécurité nationale

WWW.drdc-rddc.gc.ca